## Contents

1 Preface 1

2 Part I: first steps with SageMath 3

<table>
<thead>
<tr>
<th>2.1 Lecture 1: Welcome to MCS 320</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1 What is Computer Algebra?</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 What is SageMath?</td>
<td>4</td>
</tr>
<tr>
<td>2.1.3 Running SageMath</td>
<td>4</td>
</tr>
<tr>
<td>2.1.4 Which systems in SageMath are used?</td>
<td>5</td>
</tr>
<tr>
<td>2.1.5 Assignments</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.2 Lecture 2: Sage as a Calculator – getting Help</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1 Getting Started</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Getting Help</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 The Packages in Sage</td>
<td>7</td>
</tr>
<tr>
<td>2.2.4 Assignments</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.3 Lecture 3: Exact and Floating-Point Numbers</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1 Integer and Rational Numbers</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Floating-Point Numbers</td>
<td>10</td>
</tr>
<tr>
<td>2.3.3 Assignments</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.4 Lecture 4: Complex and Algebraic Numbers</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1 Complex Numbers</td>
<td>12</td>
</tr>
<tr>
<td>2.4.2 Algebraic Numbers</td>
<td>13</td>
</tr>
<tr>
<td>2.4.3 Assignments</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.5 Lecture 5: Symbols, Variables, and References</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1 Expressions and Names</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2 Verification of Solutions</td>
<td>15</td>
</tr>
<tr>
<td>2.5.3 Evaluation of Expressions</td>
<td>16</td>
</tr>
<tr>
<td>2.5.4 References and Shared Values</td>
<td>17</td>
</tr>
<tr>
<td>2.5.5 Assignments</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.6 Lecture 6: Data Types and Data Structures</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.1 Coercing to the Basic Number Types</td>
<td>18</td>
</tr>
<tr>
<td>2.6.2 Random Numbers</td>
<td>19</td>
</tr>
<tr>
<td>2.6.3 Components of Expressions</td>
<td>21</td>
</tr>
<tr>
<td>2.6.4 Storing Data with Functions</td>
<td>22</td>
</tr>
<tr>
<td>2.6.5 Assignments</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.7 Lecture 7: Evaluation and Execution</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.1 Addition and Multiplication Tables</td>
<td>23</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.3</td>
<td>Lecture 19: Working with Functions</td>
</tr>
<tr>
<td>4.3.1</td>
<td>List Comprehensions</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Composing Functions</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Functions Returning Functions</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Assignments</td>
</tr>
<tr>
<td>4.4</td>
<td>Lecture 20: Symbolic and Numeric Differentiation</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Symbolic Differentiation</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Numerical Differentiation</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Implicit Differentiation</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Plotting the Tangent Line</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Assignments</td>
</tr>
<tr>
<td>4.5</td>
<td>Lecture 21: Integration and Summation</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Indefinite and Definite Integrals</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Assisting the Integrator</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Symbolic Summation</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Assignments</td>
</tr>
<tr>
<td>4.6</td>
<td>Lecture 22: Series, Approximations, and Limits</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Taylor Series</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Taylor Series in SymPy</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Power Series</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Approximations</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Limits</td>
</tr>
<tr>
<td>4.6.6</td>
<td>Assignments</td>
</tr>
</tbody>
</table>

5  Indices and tables  

Index  

85  

87
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, the second edition, published by Springer in 1996. From 2001 till 2014, the course was offered, using Maple, about once every academic year. Since Spring 2015, Maple was replaced by SageMath.

The course provides an introduction to computer algebra via practical experimentation in a computer lab. The class meets three times a week, on Monday, Wednesday, and Friday for 50 minutes. The sections in these notes correspond to SageMath notebook sessions executed during those 50 minutes class meetings.

Exercises and quizzes are vital. The last 20 minutes of every Friday lecture (except during exam weeks) are spent on a quiz. The graded quiz is returned and briefly discussed at the start of the class meeting on Monday. Each Friday, a selection of the assignments is announced which will be collected as homework the following Monday. The graded homework is returned on Wednesday. Many of the assignments listed at the end of each lecture are former exam questions.
The first part of the course consists of the first 9 lectures during first three weeks of classes. 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.

2.1 Lecture 1: Welcome to MCS 320

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 $\pi$. The lecture ends with finding out which system performs a particular calculation.

2.1.1 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
2.1.2 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. 2.1.

Fig. 2.1: The motto of SageMath: building the car instead of reinventing the wheel.


2.1.3 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 
\( \exp(-x^2)\sin(8\times x) \).

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 \( \pi \) with 20 decimal places:

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

Unlike the exact value for \( \pi \), 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.

### 2.1.4 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: print sage.misc.citation.systems.keys()
```

### 2.1.5 Assignments

1. Go to [http://www.sagemath.org/tour.html](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](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 Sage 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)`?

### 2.2 Lecture 2: Sage as a Calculator – getting Help

In this lecture we show how to use the Sage 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 $\pi$, accurate up to 30 decimal places. We explore the help system of Sage, priming the continued fractions of the next lecture.

### 2.2.1 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
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)
```
With $\log_{10}(2)$ we compute the binary logarithm of 10, because the argument of \texttt{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 $\pi_{30}$. The conversion to the string representation $\texttt{str}(\pi_{30})$ is used to check that indeed we do have 30 decimal places, as confirmed by the output produced by the cell.

As simple computation of the difference between pi and $\pi_{30}$ can be done as

\begin{verbatim}
R(pi - pi30)
\end{verbatim}

which returns $0.00000000000000000000000000000000$

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

\begin{verbatim}
\texttt{RealField(40*log(10,2))}(\pi - \pi_{30})
\end{verbatim}

yields $1.69568553528388823010177990083205496387e-31$. Thus we verified that $\pi_{30}$ has the desired accuracy of 30 decimal places.

Next we illustrate the difference between precision and accuracy:

\begin{verbatim}
\texttt{pi40 = RealField(40*log(10,2))}(\pi_{30})
\end{verbatim}

we have made a variable $\pi_{40}$ with a working precision of 40 decimal places. But because it takes the value of $\pi_{30}$, the accuracy of $\pi_{40}$ is no more than 30 decimal places.

### 2.2.2 Getting Help

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

\begin{verbatim}
help(\texttt{RealField})
\end{verbatim}

To look for a specific topic, we can use the \texttt{search_doc} as illustrated below:

\begin{verbatim}
\texttt{search_doc("numerical approximation")}
\end{verbatim}

Then the user gets redirected to the extensive online documentation, shown in Fig. 2.2 below.

### 2.2.3 The Packages in Sage

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

\begin{verbatim}
\texttt{print type(\pi)}
\texttt{print type(math.pi)}
\texttt{import sympy}
\texttt{print type(sympy.pi)}
\texttt{import numpy}
\texttt{print type(numpy.pi)}
\end{verbatim}

The second and fourth type is the ordinary \texttt{float} whereas the first and third type are symbolic.

To work with the pi as defined by \texttt{sympy}:
Fig. 2.2: The documentation in Sage consists of tutorials, the reference manual, constructions (how do I construct .. in Sage?), and PREP tutorials aimed at undergraduate students.

```
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.

### 2.2.4 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 Sage. Illustrate the differences with the appropriate commands.

2. Verify that Sage 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 Sage 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 $5 \times 13 \times 31$. Use the help system of Sage 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.
2.3 Lecture 3: 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 \( \pi \) 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 \( \pi \). 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 \( x^2 - 2 \).

2.3.1 Integer and Rational Numbers

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

```python
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:

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

which prints \([1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, \ldots] \)

and the output of \( \text{show(c)} \) is displayed in Fig. 2.3.

We can verify that

\[
1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \cdots}}}}}}
\]

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

```python
f = 1 + 1/(2 + 1/(2 + 1/(2 + 1/(2 + 1/2))))
print f
print float(f), float(x)
p = [convergents(c)[k] for k in range(5)]
```

Each convergent is a rational approximation. The last command is a list comprehension and prints \([1, 3/2, 7/5, 17/12, 41/29] \) which contains 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

```python
sqrt(2).n(digits=2).exact_rational()
```

With a list comprehension, we can then compute the sequence of increasingly more accurate rational approximations for \( \sqrt{2} \).
Fig. 2.3: The continued fraction expansion of $\sqrt{2}$.

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

Note that this sequence is different from the convergents.

### 2.3.2 Floating-Point Numbers

The default floating-point type is `float` as in Python. The machine precision is defined as the largest 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.

```python
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

```python
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
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 $\sqrt{2}$.

### 2.3.3 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 $\ell$) 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 $\pi$ are 3, 22/7, 333/106, 355/113, ... In this sequence, what is the next more accurate rational approximation for $\pi$? 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.

---

**2.3. Lecture 3: Exact and Floating-Point Numbers 11**
2.4 Lecture 4: 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.

2.4.1 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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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 $\pm \sqrt{-27}$. With the option all we get to see all square roots:

\[
\text{sqrt}(a, \text{all}=\text{True})
\]

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

\[
\text{sqrt(\text{SR}(4), \text{hold}=\text{True})}
\]

### 2.4.2 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

\[
\text{sqrt(2).minpoly()}
\]

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

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

\[
\text{print factor(z^2 - 1)}
\]

\[
\text{print factor(z^2 - 2)}
\]

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 $c$.

\[
K.<c> = \text{NumberField}(x^2 - 2); \text{print } K
\]

\[
\text{print } c^2 - 2
\]

\[
\text{print } c^3
\]

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.

\[
R.<x> = \text{PolynomialRing}(K)
\]

\[
p = x^2 - 2
\]

\[
\text{factor}(p)
\]

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

In coding and cryptography we often work with finite fields.

\[
k = \text{GF}(8,'c'); \text{print } k
\]

\[
e = [a \text{ for } a \text{ in } k]; \text{print } e
\]

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.

2.4. Lecture 4: Complex and Algebraic Numbers
The multiplication table is

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0, c^2 & c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c \\
0, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c, c^2 \\
0, c^2 + c, c^2 + c + 1, c^2 + 1, 1, c, c^2, c + 1 \\
0, c^2 + c + 1, 1, c, c^2, c + 1, c^2 + c + 1 \\
0, 1, c, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1 \\
0, c, c^2, c + 1, c^2 + c, c^2 + c + 1, c^2 + 1, 1 \\
\end{array}
\]

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.

```python
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:

```python
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.

```python
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

```python
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

```python
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.

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

### 2.4.3 Assignments

1. The complex number \( z \) in polar representation is given by the radius (absolute value) \( r = 3 \) and angle (argument) \( \theta = \pi/3 \). Use Sage to find the exact (no floating-point) value of \( z \) in the form \( a + bi \).

2. Execute `solve(x^3 - 5 == 0, x)` and show that all solutions of \( x^3 - 5 = 0 \) are of the form \( 5^{\frac{1}{3}}e^{i\theta} \), with \( \theta \) being either 0 or \( \pm \frac{\pi}{2} \).
3. Take the complex number \( z = 1 + i \) and let Sage 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 Sage instructions.

4. Use Sage 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).

5. Compute all irreducible polynomials of degree 2 over a field with 3 elements.

### 2.5 Lecture 5: Symbols, Variables, and References

While Sage works with dynamic typing in a similar fashion as Python, sometimes we must declare variables explicitly with \texttt{var} as in \texttt{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.

#### 2.5.1 Expressions and Names

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

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

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

With \texttt{restore} (we could also call the restore the unassign operation) we can get back the original value of \texttt{pi}.

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

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

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

#### 2.5.2 Verification of Solutions

By default, with \texttt{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.

\[z = 3; \text{print} \ 'z = ', z\]
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.

### 2.5.3 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.

\[
\text{print } p(x=x, a=a, b=b, c=c)
\]
\[
s0 = s[0][x](a=a, b=b, c=c); \text{ print } s0
\]

Now we see the polynomial \( x^3 + x^2 + 2.000000000000000*x + 3.000000000000000 \) and a numerical value for the solution.

We can then evaluate the expression \( p \) at \( s0 \).

\[
\text{print } p(x=s0, a=a, b=b, c=c)
\]

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

\[
\text{print } \text{complex}(p(x=s0, a=a, b=b, c=c))
\]

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

### 2.5.4 References and Shared Values

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

```python
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. 2.4.

![Fig. 2.4: names of variables as references to other variables or values](image-url)
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.

```python
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` shows
Observe that the order of the assignments matters.

```python
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.

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

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

### 2.5.5 Assignments

1. Execute the statements `reset(); 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.

2. Execute the statements `reset(); var('x, y'); x = 3` in a cell. What is the next statement so `print x, y` shows `3, x` as `y` refers to `x`, as `x` refers to `3`.

### 2.6 Lecture 6: Data Types and Data Structures

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.

#### 2.6.1 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.
We can convert a string in hexadecimal format or octal format into decimal notation.

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

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

```python
c = ZZ(range(10), base=10); print c
d = ZZ(range(10), base=3); print d
```

The number `c` lists the decimal digits 9876543210 and 250959 uses the same coefficients to make the number

\[0 \cdot 3^0 + 1 \cdot 3^1 + 2 \cdot 3^2 + 3 \cdot 3^3 + 4 \cdot 3^4 + 5 \cdot 3^5 + 6 \cdot 3^6 + 7 \cdot 3^7 + 8 \cdot 3^8 + 9 \cdot 3^9.\]

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

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

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

```python
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.

```python
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 number we can add to 1.0 and still make a difference. We can compute the machine precision as follows:

```python
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:

```python
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`.

### 2.6.2 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:
Random real numbers are generated as follows:

```python
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 $\mathbb{CC}$ and the Complex Double Field $\mathbb{CDF}$.

```python
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:

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

Then the output of `bar_chart(L)` is shown in Fig. 2.5.

![Bar chart of 100 random numbers](image)

**Fig. 2.5:** The bar chart of 100 random numbers.

If we sort the numbers, then we can see that the distribution tends to be uniform.
The sorted numbers are visualized in Fig. 2.6.

![Bar chart of 100 sorted random numbers.](image)

Fig. 2.6: The bar chart of 100 sorted random numbers.

### 2.6.3 Components of Expressions

We often work with equations.

```python
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.

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

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

```python
print eqn.lhs()
p
```
Alternatives to \texttt{lhs()} are \texttt{left()} and \texttt{left\_hand\_side()} and instead of \texttt{rhs()} we may also use \texttt{right()} and \texttt{right\_hand\_side()}.

### 2.6.4 Storing Data with Functions

The example in this section is taken from the book \textit{Sage for Power Users} by William Stein.

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

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

Let us now execute the function a couple times.

```python
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 \texttt{L} each time. Because we will need to use the address later, we return \texttt{id(L)}.

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

We run this function \texttt{our\_append2} also three times.

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

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 \texttt{our\_append2} without giving an argument for the list \texttt{L}, the arguments were evaluated. The effect of \texttt{L = []} is that an empty list is created and placed somewhere in memory. Each time the function is called with the default argument of \texttt{L}, the same memory location is used. Note that the name \texttt{L} does not exist outside the function. Just to check, \texttt{print L} will result in a \texttt{NameError}.

With \texttt{ctypes} we can retrieve the object an address refers to.

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

Executing the cell shows
## 2.6.5 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.

4. 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.

## 2.7 Lecture 7: 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 Sage 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.

### 2.7.1 Addition and Multiplication Tables

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

```python
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`.

```python
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`.

```python
print Z3.multiplication_table()
```
and then we see

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

Does this work if we extend \( \mathbb{Z}_3 \) with an algebraic number? We define an irreducible polynomial with coefficients in \( \mathbb{Z}_3 \).

```python
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 \( \mathbb{Z}_3 \).

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

Then Sage 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.

```python
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.

```python
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 + 1, 2, a, a + 2, 2*a, 2, a]
[0, a + 2, 2*a + 1, a + 1, 2*a, 2, 2*a + 2, 1, a]
[0, a + 1, 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]
[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 \( \mathbb{Z}_3 \).

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

```python
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  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  
```

### 2.7.2 Expression Trees

We are interested in the internal structure of expressions.

```python
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 $x \times y$

print $x / y$

and we see $\text{sub}(v_0, v_1), \text{mul}(v_0, v_1),$ and $\text{div}(v_0, v_1)$.

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

```python
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 $y$

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

This prints $\text{add}(\text{sub}(\text{add}(\text{ipow}(v_0, 3), \text{mul}(\text{mul}(4, v_0), \text{ipow}(v_1, 2))), \text{mul}(7, v_1)), 2)$. 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
  |  |  |  |  --- 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.
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.

```
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 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 $4 \times x \times y^2$ in Fig. 2.7.

```
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')
```

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

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

We then see the expression tree for $x^3 + 4 \times x \times y^2$ in Fig. 2.8.

```
N2345 = LabelledBinaryTree([N23, N45], label='mul')
```

Then we subtract $7 \times y$, as defined in node `N67`. 

```
N012345 = LabelledBinaryTree([N01, N2345], label='add')
```
We then see the expression tree for $x^3 + 4*x*y^2 - 7*y$ in Fig. 2.9.

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

The final expression tree is shown in Fig. 2.10.

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

The output is a list
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 \( \text{ipow} \) 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.

### 2.7.3 Assignments

1. Consider computing modulo 5. This corresponds to computing with numbers in \( \mathbb{Z}_5 = \{0, 1, 2, 3, 4\} \). 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. 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 \( p \).

### 2.8 Lecture 8: Input/Output Formats – Saving 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.

#### 2.8.1 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 \( \pi \).

```python
x = numerical_approx(pi, digits=20)
print 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`.

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

If we omit the `/tmp` in the file name, which is necessary on Windows computers, then the file is written in the current directory, provided that the current directory is writable to the user. 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.
We clear everything to make sure $x$ is gone.

```python
reset()
print x
```

The `print` just shows the symbol $x$ the reference of $x$ to the 20-digit approximation of $\pi$ is gone.

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

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

```python
x = preparse(s)
print type(x), x
y = eval(x)
print type(y), 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 3.141592653589793239.

### 2.8.2 Saving and Loading SageMath Objects

The `save` and `load` are much easier than working directly with Python files. In this section we continue with the $y$ from the previous section. If the $y$ is lost, just do $y = \text{numerical_approx}(\pi, \text{digits}=20)$. 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('/tmp/sageynum')
```

The result of executing the `save` is that the directory `/tmp/` contains the file `sageynum.sobj`. We execute `reset('y')` to remove the reference to $y$.

```
reset('y')
print y
```

Because $y$ is after the `reset` no longer defined, we get a `NameError`.

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

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

On return in $z$ is 3.141592653589793239 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`. 
2.8.3 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 \( z = \text{numerical\_approx}(\pi, \text{digits}=20) \).

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

The pickled object is the string with multiple lines.

```plaintext
csage.rings.real_mpfr
__create__RealNumber_version0
p0
(csage.rings.real_mpfr
__create__RealField_version0
p1
(I67
I00
S'RNDN'
p2
tp3
Rp4
S'3.4gvml245kc4d80@0'
p5
I32
tp6
Rp7
.
```

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

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

We delete the string s and reset z.

```python
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.

```python
file = open('/tmp/sageznum.txt', 'r')
lines = file.readlines()
print lines
```

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

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

The output of `print s` shows the pickled representation of the real number. Now we can reconstruct the number z from the pickled object.
\[
\begin{align*}
z & = \text{pickle.loads}(s) \\
\text{print type}(z), z
\end{align*}
\]

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

### 2.8.4 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 approximation 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?

### 2.9 Lecture 9: Code Generation with 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 \( \pi \). The fastest version applies numpy vectorization, although this approach requires a reformulation of the original script.

#### 2.9.1 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`. 

---

32 Chapter 2. Part I: first steps with SageMath
def python_sum_symbolic(n):
    return float(sum(sqrt(1-(k/n)**2) for k in xrange(1, n+1)))/n
4*python_sum_symbolic(1000)

We see 3.139554669110264 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.

2.9.2 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 a floating-point number.

def python_sum(n):
    from math import sqrt
    return sum( sqrt(1-(k/n)**2) for k in xrange(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.

2.9.3 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.

2.9. Lecture 9: Code Generation with Cython
The vectorization of code is the replacement of a Python for loop in `for k in xrange(1, n+1)` by a loop executed by numpy functions.

```python
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.

```python
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.

```python
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.

```python
print 'speedup :', ct3/ct4
```

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

### 2.9.4 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 xrange(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.

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

and then we benchmark the code with `timeit`.

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

We obtain 25 loops, best of 3: 26.2 ms per loop. Running a larger example with `time`
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)
def cython_sum_typed(long n):
    cdef long k
    return sum( sqrt(1-(k/float(n))**2) for k in xrange(1, n+1) )/n
```

We first check the correctness.

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

and now we will see improved timing results.

```python
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.

```python
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.

```python
print 'speedup :', ct5/ct6
```

which prints speedup : 23.6068401182.

### 2.9.5 Assignments

1. Perform all computations on the computers in the lab. 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.
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.

### 3.1 Lecture 10: 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.

#### 3.1.1 Polynomials as Expressions

Let us start with a definition in plain words. A univariate polynomial is a sequence 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 \( x \).

```python
p = x^4 - 4*x^2 - 7*x + 9
print p, 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?

```python
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.

```python
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.

```python
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!

```python
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.

```python
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.

### 3.1.2 Univariate Polynomials

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

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

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

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

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

```python
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.

```python
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.

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

and build a field extension to add a root of $q$ to $\mathbb{Q}$

```python
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 \( q \).

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

See if \( q_2 \) factors or not.

```python
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 \( K \).

```python
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.

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

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

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

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

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

Now make our last field extension!

```python
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 \( c \).

```python
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.

```python
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.

```python
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.
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.

3.1.3 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.

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

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

```python
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.

```python
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.

```python
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.

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

3.1.4 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 \( x = \text{polygen}(\mathbb{Q}). \)
   (a) 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 \( q = 2x^5 + 5x^4 + 7x^3 + 8x^2 + 5x + 3. \)
(b) How can Sage compute the cofactors \( k \) and \( \ell \) so that \( \gcd(p, q) = kp + \ell q \)?

(c) Finally, give the Sage commands to verify the relation \( \gcd(p, q) = kp + \ell q \) for the \( k \) and the \( \ell \) that were found.

Hint: do `help(xgcd)`.

3. Do 
\[
x = \text{polygen}(\text{RR}); \quad \text{print factor}(x^2 - 2.25) \quad \text{and} \quad x = \text{polygen}(\text{QQ}); \quad \text{print factor}(x^2 - 9/4).
\]
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:

(a) symbolically, by adding sufficiently many formal roots; and

(b) 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) \( 2z^4 x + z^3 x + 2z^2 y \),

(b) \( 2xz^4 + xz^3 + 2z^2 y \), 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 \( \text{var}('x') \) and \( \text{QQ}['x'] \).

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 \( \text{var}('x') \), when \( \text{QQ}['x'] \)?

What can you do after \( \text{QQ}['x'] \) but not after \( \text{var}('x') \)?

### 3.2 Lecture 11: Rational Functions and Conversions

One of the main problems in computer algebra is expression swell.

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.

#### 3.2.1 Rational Expressions

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

```python
x = polygen(QQ)
p = x^3 - 1; q = x^2 - 1; r = p/q
print r, 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 \( r \).
This automatic normalization may lead to expression swell.

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.

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

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

The same for the numerator, with or without normalization

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

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

If we really do not want to remove common factors, then we must explicitly convert to strings.

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

### 3.2.2 Conversions

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

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

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

### 3.2.3 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 \( k = 0, 1, \ldots, 8 \).

### 3.3 Lecture 12: 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.

#### 3.3.1 Expression Trees

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

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

```python
x, y, z = var('x,y,z')
sq = p(x=x,y=y,z=z)
print sq, 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.

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

What is printed is \([x \cdot y^2 \cdot z^4, 1/5 \cdot x^5 \cdot y, 2 \cdot x \cdot y^2 \cdot z^2, x^3]\) and \(<\text{function add_vararg at 0x1192011b8}>\). 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.

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

The top level of the expression tree is shown in Fig. 3.1.

![Expression Tree](image)

**Fig. 3.1:** The top level of the expression tree of a multivariate polynomial.

We can now apply the same operations to the leaves.

```python
ostr Aer = [op.operands() for op in sq.operands()]
ostr Aer = [op.operands() for op in sq.operands()]
print ostr Aer
print ostr Aer
```

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.

```python
stroperands = [[str(x) for x in L] for L in ostr Aer]
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. 3.2.

```
  / \ / /
 x  y^2 z^4
```

```
  / \
 x^5 y 1/5
```

```
  / \ / /
 x  y^2 z^2 2
```

```
  / \ /  \
 x   3
```

Fig. 3.2: 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. 3.3.

```
  / \ / /
 x  y^2 z^4
```

```
  / \
 x^5 y 1/5
```

```
  / \ / /
 x  y^2 z^2 2
```

```
  / \ /  \
 x   3
```

Fig. 3.3: 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.
Now we can define the nodes which represent the powers.

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

The other nodes are defined similarly.

```python
nodexpow5 = LabelledOrderedTree([leafx, leafpow5], label='^')
nodexpow4 = LabelledOrderedTree([leafy, leafpow4], label='^')
nodexpow4 = 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.

```python
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.

```python
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.

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

The complete expression tree is shown in Fig. 3.4.

### 3.3.2 Evaluation of Expressions

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

```python
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,
Fig. 3.4: The complete expression tree of a multivariate polynomial.

- with $p$ we do use the variable names as key words.

```python
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 397.4000000000000. To time the evaluation, we use `timeit`.

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

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

```python
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.

```python
f.op_list()
```

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

```python
[('load_arg', 0),
 ('load_arg', 1),
 ('ipow', 2),
 'mul',
 ('load_arg', 2),
 ('ipow', 4),
 'mul',
 ('load_arg', 0),
 ('ipow', 5),
 ('load_arg', 1),
 'mul',
 ('load_const', 1/5),
 'mul',
 'add',
 ('load_arg', 0),
 ('load_arg', 1),
 ('ipow', 2),
 'mul',
 ('load_arg', 2),
 ('ipow', 2),
 (continues on next page)
```
3.3.3 Assignments

1. Consider \( p = 3x^4 - 6x^3 + 5x^2 + 9x - 7 \). Draw the expression tree for \( p \). Also give all Sage commands with their output used to make your drawing.

2. Consider \( p = 3x^4 - 6x^3 + 5x^2 + 9x - 7 \). 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 \( p = 3x^4 - 6x^3 + 5x^2 + 9x - 7 \) and its evaluation at \( \text{math.pi} \). Compare with \texttt{timeit} the efficiency of the original expression, the Horner \( q \) form of \( p \), and the fast callable objects of \( p \) and \( q \).

3.4 Lecture 13: Substitution, Expansion, and Factorization

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

3.4.1 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 \((x+y)^2 + 1/((x+y)^2)\) into \((x+y)^4 + 1/((x+y)^2)\).

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

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

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

```python
p.factor()
```

The output is \((x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 + 1)/(x + y)^2\). The \texttt{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 \(x + y\).

```python
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 \texttt{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. 3.5.

![Simultaneous substitution](image)

Fig. 3.5: Simultaneous substitution of the variables in an expression as done by \( e(a=b, b=c, c=a) \) for \( e = a + 2*b + 3*c \).

The sequential substitution of the variables in an expression is illustrated in Fig. 3.6.

![Sequential substitution](image)

Fig. 3.6: Sequential substitution of the variables in an expression, as done by \( e.subs(a=b).subs(b=c).subs(c=a) \) for \( e = a + 2*b + 3*c \).

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

```
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,a=c)
```
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.

```python
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.

```python
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)`.

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

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

### 3.4.2 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.

```python
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.

```python
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.

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

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

```python
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`.  

50 Chapter 3. Part II: polynomials and expressions
There is an alternative and shorter way to obtain the above result. We view \( p \) as a polynomial in \( x \) with coefficients in \( \mathbb{Q}[a,b,c] \). Converting \( p \) into the ring \( \mathbb{Q}[a,b,c][x] \) is straightforward:

\[
q = \mathbb{Q}[a,b,c][x](p)
\]

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.

### 3.4.3 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.

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

and we see the expression \((x + 1) \cdot (x - 1)\). 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 \( \mathbb{Q} \) with a root of an irreducible polynomial, using the symbol \( a \) in the extended number field \( K \).

```python
y = polygen(QQ)
q = y^2 + 2
print factor(q)
polyren
q.is_irreducible()
K.<a> = QQ.extension(q)
kq = (K[y])(q)
print factor(kq)
```

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

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

and the numerical (approximate) factorization is \((x - 1.41421356237310\, \text{i}) \cdot (x + 1.41421356237310\, \text{i})\).

### 3.4.4 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 \( \frac{(x+1)^3+1}{(x+1)^2} \) 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 \(xy - x(z^2 + 1) + (z^2 + 1)y - (z^2 + 1)^2\).

### 3.5 Lecture 14: 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.
3.5.1 Normal and Canonical Form

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

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

```python
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.

```python
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?

```python
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.

```python
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).

3.5.2 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)
print 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

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, type(q)

We see that multivariate polynomials in Sage are implemented via libSINGULAR. SINGULAR is a computer algebra package 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.

3.5.3 A Numerical Test on Equality

There is a numerical probability-one test on 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.

3.5.4 Assignments

1. Consider the polynomial \( p = (x^2 + xy + x + y)(x + y) \).
(a) Order the monomials in \( p \) in total degree order, using \( x > y \).

(b) 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 \( (x^2 - 1)/(x - 1) \).

   (a) Are these expressions symbolically the same? Give the Sage commands to illustrate your answer.

   (b) Verify the equality numerically. Show how a numerical equality test can go wrong.

### 3.6 Lecture 15: Review of the First 14 Lectures

In the first 14 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

   (a) to compute a rational approximation for \( 2\sqrt{2} \) accurate with 5 decimal places;

   (b) to show that the accuracy of this approximation is indeed 5 decimal places;

   (c) 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 \( x = \text{polygen}(\mathbb{Q}) \) and \( \text{var}('x') \).

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. Consider the polynomial \( p = 25x^3 + 12x^2 + 26x + 4 \) over a finite field of size 29, similar to working modulo 29.

   (a) Does the polynomial factor over this field?

   (b) Give the Sage commands to add sufficiently many formal roots to the field so \( p \) has three roots over the extended number field.

   (c) Write the symbolic factorization of \( p \) in linear factors below.

6. Explain how rational expressions are normalized. Illustrate with an example.

7. Type \( \text{var} ('x', 'y'); \ q = (x^2 - y) / (y^2 - x) \) and draw the expression three of \( q \).

8. Give the Sage commands to transform \( (x+(z^2+1))(y-(z^2+1)) \) into \( xy+(-z^2-1)x+(z^2+1)y-z^4-2z^2-1 \).

### 3.7 Lecture 16: the First Midterm Exam

The first midterm exam covers the first two parts of the course.
3.7.1 Questions on the Spring 2017 Midterm Exam

The list of questions on the Spring 2017 midterm 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 \( \exp(\pi) \).
   (a) Write a rational approximation \( A \) for \( N \), accurate with 3 decimal places.
   (b) Give the Sage command(s) to verify that \( A \) approximates \( N \) indeed with 3 decimal places.
   (c) 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.
   (a) Write the Sage commands to define \( p \) and to show that \( p \) does not factor over this finite field.
   (b) 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 \( z \).
   (a) Define one normal form of such a polynomial.
   (b) Illustrate your definition with an example of a random polynomial.
   (c) 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
   (a) transform \( p \) into \( q \); and
   (b) transform \( q \) into \( p \).

3.7.2 Questions on the Fall 2018 Midterm Exam

The list of questions on the Fall 2018 midterm 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.
   (a) Give a floating-point approximation \( A \) of \( N \), accurate with 5 decimal places.
   (b) 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.
(c) 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 $x, y = \text{var}('x, y')$.
   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?
   (a) Give all Sage commands to convert $p$ into an expression best suited for fast evaluation. Do not write the output of the Sage commands.
   (b) 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.
   (a) Is $p$ irreducible? If not, what are the factors of $p$?
   (b) 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 $x, y = \text{var}('x, y'); q = (x-y+1)/(x^2+3*y)$ 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 $\frac{(x-y)^2}{(x+y)^2} + 1$ into $\frac{(x-y)^2 + (x+y)^2}{(x+y)^2}$?
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.

### 4.1 Lecture 17: Defining Mathematical Functions

Expressions in Sage 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.

#### 4.1.1 Functions in Sage and in Python

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

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

The `print f` shows \((x, y) \mapsto e^{\cos(y)} + \sin(x)\) and we see that a function is an expression.

```python
print f(2,pi)
sage: print f.integrate(x)
```

This shows \(e^{-1} + \sin(2)\) and \((x, y) \mapsto x e^{\cos(y)} - \cos(x)\). Observe thus that, as we integrate a function, the result is again displayed as a function.

Let us explore the difference with Python functions.

```python
def g(x,y):
    return sin(x) + e^cos(y)
print g, type(g)
```
Introduction to Symbolic Computation, Release 0.5.5

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

\[
g \text{.integrate}(x)
\]

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

\[
e = g(x,y) \\
\text{print } e, \text{ type}(e)
\]

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

So we can turn an expression into a function.

\[
h(x,y) = e \\
\text{print } h, \text{ type}(h) \\
\text{print } h \text{.integrate}(x)
\]

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

### 4.1.2 Step Functions

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

\[
\text{print } \text{unit_step, type(unit_step)} \\
\text{print } '\text{at -3 :}', \text{ unit_step(-3)} \\
\text{print } '\text{at 4 :'}, \text{ unit_step(4)}
\]

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.

\[
\text{plot(unit_step, -1, 1, aspect_ratio=1)}
\]

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. 4.1.

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

\[
\text{diff(unit_step(x),x)}
\]

which shows `dirac_delta(x)`.

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

\[
\text{integral(unit_step(x), (x, -3, 3))}
\]

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.

\[
\text{print kronecker_delta(1,2)} \\
\text{print kronecker_delta(1,1)}
\]

and we see 0 and 1 printed.
4.1.3 Piecewise Functions

We can make piecewise functions. For example,

\[
  f(x) = \begin{cases} 
  1 & \text{if } x < 1 \\
  \infty & \text{if } x = 1 \\
  x^2 & \text{if } x > 1 
  \end{cases}
\]

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

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

4.1.4 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 \( x \geq 2 \).

```python
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. 4.2.

4.1.5 Assignments

1. Define the following function:

\[
  f(x) = \begin{cases} 
  3x + 1 & \text{if } x \text{ odd} \\
  x/2 & \text{if } x \text{ even} 
  \end{cases}
\]
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 Sage function.

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

3. Make a piecewise linear function that is zero for negative values of $x$ and takes the value $x/2$ for positive values of $x$.

4. Give the Sage commands to 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 $[-1, +1]$. Make a plot to check that your definition is correct.

### 4.2 Lecture 18: 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.

#### 4.2.1 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. A first version as a Python function is defined below.

```python
def fibonacci(n):
    """
    Returns the n-th Fibonacci number.
    """
    if(n == 0):
        result = 0
    elif(n == 1):
        result = 1
    return result
```

(continues on next page)
Look at the first 10 Fibonacci numbers.

\[
\text{[fibonacci(n) for n in range(11)]}
\]

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. 4.3 as they occur in the computation of $F(2)$.

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. 4.4.

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.
    """
    if D.has_key(n):
        return D[n]
    else:
        if (n == 0):
            result = 0
        elif (n == 1):
            result = 1
        else:
            result = memoized_fibonacci(n-1) + memoized_fibonacci(n-2)
        D[n] = result
    return result
```

Now if we time this new function.
we see 625 loops, best of 3: 590 ns per loop.
[memoized_fibonacci(n) for n in range(30)]

4.2.2 Memoization in Sage

Functions in Sage can compute expressions. Chebyshev polynomials are orthogonal polynomials, available in Sage via the function chebyshev_T.

t3 = chebyshev_T(3, x)
print t3, type(t3)

and we see `'4*x^3 - 3*x <type 'sage.symbolic.expression.Expression'>`.`

The 3-terms recurrence to define Chebyshev polynomial is $T(0,x) = 1$, $T(1,x) = x$, $T(n,x) = 2xT(n-1,x) - T(n-2,x)$. The straightforward definition based on this 3-terms recurrence is below in the function T. Observe the application of expand().

```
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.

```
print 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 Sage function T and call the memoized function mT.

```
def mT(n, x, D = {}):
    #
    # Returns the n-th Chebyshev polynomial in x,
    # using memoization with the dictionary D.
    #
    if D.has_key(n):
```

(continues on next page)
return D[n]
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] = result
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.

4.2.3 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 $c(0, 0) = 1$.

(a) 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).

(b) How many digits does the number $c(100, 33)$ have? Give also the Sage 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 $L(0,x) = 1$, $L(1,x) = x$, and 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 Sage 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: \( T(0, f, a, b) = (f(a) + f(b)) \times (b-a)/2 \) and \( T(n, f, a, b) = T(n-1, f, a, (a+b)/2) + T(n-1, f, (a+b)/2, b) \), for \( n > 0 \).

(a) Write a recursive Sage function for \( T \).

(b) 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 \( \int_0^1 \cos(x)dx \).

4.3 Lecture 19: Working with Functions

List comprehensions in Python take the form \([f(x) \text{ for } x \text{ in } L]\) 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 Sage we can apply list comprehensions to build expressions of arbitrary size and shape.

4.3.1 List Comprehensions

We can use a list comprehension to create a sequence of 20 variables, starting from \( x01, x02, \ldots, x20 \).

\[
v = [\text{var('x' + '02d' % k)} \text{ for } k \text{ in range(1,21)}]
\]

\[\text{print v}\]

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.

\[
p = [x^3 \text{ for } x \text{ in } v]
\]

\[\text{print p}\]

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.

\[
e = \text{sum}(p)
\]

\[\text{print e, type(e)}\]

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 \( \text{operands()} \) method.

\[
\text{ope = e.operands()}
\]

\[\text{print ope, type(ope)}\]

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.

\[
r = [\text{ope[k]}^k \text{ for } k \text{ in range(len(ope))}]
\]

\[\text{print r}\]

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], type(r[2])
print 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.

\[
\text{print } [(r[k]).degree(v[k]) \text{ for } k \text{ in range(len(ope))}]
\]

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 \(x^6 \ : \ 1\) and \(x^{24} \ : \ 0\). Now we can remove all operands with degree higher than 13.

\[
\text{filter = lambda x, k: x*unit_step(13 - x.degree(v[k]))}
\]
\[
\text{s = [filter(r[k],k) for k in range(len(r))}]
\]
\[
\text{print s}
\]

and we see \([1, x^3, x^6, x^9, x^{12}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]\) and then build the expression again with sum().

\[
t = \text{sum(s)}
\]
\[
\text{print t}
\]

and we obtain \(x^{12} + x^9 + x^6 + x^3 + 1\).

### 4.3.2 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.

\[
y(x) = x*(120 - x)
\]
\[
\text{plot(y(x), (x, 0, 120), aspect_ratio=1/100, thickness=3, color='red')}
\]

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. 4.5.

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.

\[
x(t) = 120/45*t
\]
\[
\text{print 'x at 0 :', x(0)}
\]
\[
\text{print 'x at 45 :', x(45)}
\]

and indeed, we see \(x \text{ at 0 : 0}\) and \(x \text{ at 45 : 120}\) printed. Then the altitude is given as the composition of \(y\) after \(x\).
Fig. 4.5: A parabolic trajectory.

```
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:

### 4.3.3 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)
print our_sqrt
```
Fig. 4.6: The parabolic trajectory plotted in three dimensions.
Then we see $x \rightarrow x - 1/2 + (x^2 - 2)/x$. 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
print s
```

We will then execute `our_sqrt(our_sqrt(our_sqrt(our_sqrt(our_sqrt(2))))))` and we evaluate the expression.

```
v = eval(s)
print 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
```

### 4.3.4 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 Sage 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 `q = P.random_element(degree=10)`. Select from `q` all terms with positive coefficients and make a new polynomial with those terms.

4. Let $n$ and $k$ be positive natural numbers with $k < n$ and consider the polynomial $\sum_{i=0}^{n-1} \prod_{j=0}^{k-1} z^{i+j} \mod n$.

   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 $k = 3$.

5. Execute the `newton_step` function on the `our_sqrt` 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 progress of the accuracy of the rational approximations for the square root of 2? Do you observe quadratic convergence?

4.4 Lecture 20: 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.

4.4.1 Symbolic Differentiation

We can compute the derivative of an expression with \texttt{diff}, which is an alias for the \texttt{derivative} method.

\begin{verbatim}
print 'the derivative of cos(x) is', diff(cos(x),x)
\end{verbatim}

and we see that the derivative of \(\cos(x)\) is \(-\sin(x)\). For repeated differentiation, we have an extra argument.

\begin{verbatim}
[diff(cos(x), x, k) for k in range(5)]
\end{verbatim}

and this list comprehension returns \([\cos(x), -\sin(x), -\cos(x), \sin(x), \cos(x)]\). We can compute derivatives of functions.

\begin{verbatim}
f(x,y) = x^2*y + 2*x*y + x
print 'f =', f
print 'the derivative of f is', f.diff()
\end{verbatim}

The function \(f = (x, y) \mapsto x^2*y + 2*x*y + x\) has as its derivative \((x, y) \mapsto (2*x*y + 2*y + 1, x^2 + 2*x)\). 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 \textit{gradient} of \(f\).

\begin{verbatim}
print f.diff(x)
print f.diff(y)
\end{verbatim}

and we see the two partial derivatives \((x, y) \mapsto 2*x*y + 2*y + 1\) with respect to \(x\) and \((x, y) \mapsto x^2 + 2*x\) with respect to \(y\). What if we take the derivative twice?

\begin{verbatim}
print f.diff().diff()
\end{verbatim}

we get

\begin{verbatim}
[ (x, y) \mapsto 2*y (x, y) \mapsto 2*x + 2]
[(x, y) \mapsto 2*x + 2 (x, y) \mapsto 0]
\end{verbatim}

The matrix of second order derivatives of a function in several variables is called the \textit{Hessian}. To compute an element in the matrix we give names of variables as arguments to the \texttt{diff}.

\begin{verbatim}
print f.diff(x).diff(y)
\end{verbatim}

and we obtain \(\frac{\partial^2 f}{\partial x \partial y}\) as \((x, y) \mapsto 2*x + 2\).
4.4.2 Numerical Differentiation

Numerical differentiation is available in `scipy.misc` in the derivative method.

```python
from scipy.misc import derivative as numdif

print 'exact derivative :', -sin(1.0)
print '1st order approx :', numdif(func=cos(x), x0=1.0, dx=1.0e-2)
```

and we then see

```plaintext
exact derivative : -0.841470984807897
1st order approx : -0.841456960361603
```

The numerical differentiation uses central differences. With extrapolation we can improve the accuracy.

```python
ND = [numdif(func=cos(x), 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

```plaintext
-0.841456960361603
-0.841470984527404
-0.841470984807883
-0.841470984807891
-0.841470984807884
-0.841470984807897
```

4.4.3 Implicit Differentiation

We can declare derivatives symbolically. Assume one variable \( y \) is a function of another variable \( x \).

```python
x = var('x')
y = function('y')(x)
dy = y.diff(x)
print dy, 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.

```python
circle = x^2 + y^2 - 1 == 0
print circle, 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.

```python
dc = circle.diff(x)
print dc
```

The equation we now can solve for \( dy \) is

```plaintext
2*y(x)*diff(y(x), x) + 2*x == 0
```

Thus we solve for the derivative.
Introduction to Symbolic Computation, Release 0.5.5

```
yx = solve(dc, dy)
print 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. 4.7.

### 4.4.4 Plotting the Tangent Line

Let us make the plot shown in Fig. 4.7.

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 \(\theta\), we generate a random number in the interval \([0, \pi/2]\).

```python
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.

```python
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.

```python
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.

```python
var('Y')
tangentline = Y - circlepoint[1] - slope*(x-circlepoint[0])
```

Then the plotting instructions which yield Fig. 4.7 are below.

```python
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)
```

### 4.4.5 Assignments

1. Give the Sage command to compute \(\frac{\partial^8 f}{\partial x^3 \partial y^5}\) 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 \(x\).
Fig. 4.7: The tangent line to a point on a circle.
3. Consider the curve defined by the equation $x^3 - y^2 + x = 0$. The point $P$ with coordinates $\left(\frac{1}{2}, \sqrt{10}^4\right)$ 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 Sage 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 $\frac{dy}{dx}$. What is the value for $y'(1) = \left.\frac{dy}{dx}\right|_{x=1}$?

4.5 Lecture 21: 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.

4.5.1 Indefinite and Definite Integrals

The counterpart to derivative is integral.

```sage
integral(cos(x), x)
```

and we see $\sin(x)$. We can verify that integration is anti-differentiation and differentiation is anti-integration.

```sage
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.

```sage
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 expression.

```sage
d = integral(e, (x, 0, 1))
print d
d.show()
```

and we see $\int_0^1 \cos \left( e^{x^2} \right) \, dx$.

Numerical integration gives an approximation for the definite integral.

```sage
print d.n()
```

and we see $0.112823456937$.

4.5.2 Assisting the Integrator

We can compute definite integrals symbolically.
\textbf{Introduction to Symbolic Computation, Release 0.5.5}

\begin{verbatim}
var('a, b')
print integral(x^2, (x, a, b))
\end{verbatim}

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.

\begin{verbatim}
var('a, b')
print integral(1/x^2, (x, a, b))
\end{verbatim}

The exception triggered is \texttt{ValueError} with the comment that Computation failed since Maxima requested additional constraints; \texttt{and moreover: 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.

\begin{verbatim}
ad = integral(1/x^2,x)
print ad
print ad(x=b) - ad(x=a)
\end{verbatim}

The antiderivative 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.

\begin{verbatim}
plot(1/x^2, (x, -1, 1), ymax=100)
\end{verbatim}

The parameter \texttt{ymax=100} puts a bound on the \(y\)-value. The plot is shown in \texttt{Fig. 4.8}.

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 \textit{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

\begin{verbatim}
Is b-a positive, negative or zero?
\end{verbatim}

We see that Sage asks about the sign of \(b-a\). We can add assumptions to variables.

\begin{verbatim}
assume(a > 0, b > a)
print integral(1/x^2, (x, a, b))
\end{verbatim}

and then we have the value \(1/a - 1/b\). We can check the assumptions.

\begin{verbatim}
assumptions()
\end{verbatim}

and we then see \([a > 0, b > a]\). Assumptions are removed with \texttt{forget}.

\begin{verbatim}
forget(b > a)
print assumptions()
forget(a > 0)
print assumptions()
\end{verbatim}

We then see printed:

\begin{verbatim}
[a > 0]
[]
\end{verbatim}

\section*{4.5. Lecture 21: Integration and Summation}
Fig. 4.8: The function $1/x^2$ has an asymptote at $x=0$. 
4.5.3 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

\[
\sum_{k=1}^{n} k.
\]

```python
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 \( n = 100 \).

```python
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 \( \infty \).

```python
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.

```python
L = [float(1/k^2) for k in range(1, 100001)]
print sqrt(6*sum(L))
```

and then we see 3.14159310433.

4.5.4 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 \( F(T) = \int_1^T \frac{\exp(-t^2)}{t} \, dt \).
   
   (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 \( \int_1^2 \frac{\exp(-t^2)}{t} \, dt \).

   (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?
Introduction to Symbolic Computation, Release 0.5.5

Compute \( \int_0^1 \left( \int_0^1 \frac{x - y}{(x + y)^3} \, dy \right) \, dx \) 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?

4.6 Lecture 22: Series, Approximations, and Limits

The manipulation of series is a form of hybrid exact-approximation computation. We start with Taylor series and then 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.

4.6.1 Taylor Series

Let us construct a Taylor series for \( \sin(x) \) of order 6 at \( x = 0 \).

```python
tsin = taylor(sin(x), x, 0, 6)
print tsin, type(tsin)
```

and we see \( \frac{1}{120}x^5 - \frac{1}{6}x^3 + x \) 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.

```python
a = tsin(0.01)
b = sin(0.01)
print a, b, abs(a-b)
```

which shows the numbers \( 0.0099998333416667, 0.00999983333416666 \), 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 \( a \).

```python
x, a = var('x, a')
f = function('f')(x)
.tf = taylor(f(x), x, a , 4)
print tf, type(tf)
```

so we get the general form of a Taylor expansion of any function \( f \) in \( x \) at \( a \) as

\[
\frac{1}{24}(a - x)^4D[0, 0, 0, 0](f)(a) - \frac{1}{6}(a - x)^3D[0, 0, 0](f)(a) + \frac{1}{2}(a - x)^2D[0, 0](f)(a) - (a - x)D[0](f)(a) + f(a)
\]

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

```python
g = x/(1-x-x^2)
tg = taylor(g(x), x, 0, 10)
print tg
print tg.coefficients()
```

The `Taylor` series is \( 55x^{10} + 34x^9 + 21x^8 + 13x^7 + 8x^6 + 5x^5 + 3x^4 + 2x^3 + x^2 + x ` and the coefficients (with corresponding exponents) 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 \( y = 0 \).

```python
x, y = var('x, y')
h = cos(x)*sin(y)
print taylor(h, (x,pi/2), (y,0), 5)
```
and we get $-1/48*(\pi - 2*x)^3*y - 1/12*(\pi - 2*x)*y^3 + 1/2*(\pi - 2*x)*y$.

### 4.6.2 Taylor Series in SymPy

Let us see what systems Sage uses to compute the Taylor series.

```python
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.

```python
from sympy.series import series
print 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:

```python
g = series(sin(x), x0=0, n=None)
print g, type(g)
```

and we see that $g$ is a generator object. We can use a generator with the `next()` method to get the next term in the series.

```python
print g.next()
p = g.next()
```

or in a list comprehension to get the next 5 terms in the series

```python
[g.next() for k in range(5)]
```

### 4.6.3 Power Series

In Sage, Taylor series are not power series, but we can turn any polynomial into a power series.

```python
x = var('x')
q = x^3 + 3*x + 8
pq = q.power_series(QQ)
print pq, 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.

```python
ipq = 1/pq
print ipq, type(ipq)
p = 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.

print 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, 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.

### 4.6.4 Approximations

Padé approximations are rational approximations.

```python
z = PowerSeriesRing(QQ, 'z').gen()
pd = exp(z).pade(4,4)
print pd, 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.

```python
print pd(3.14)
print exp(3.14)
```

and we see that we have two decimal places correct:
Let us compare a power series approximation for \( \sin(x) \) with a Padé approximation.

```python
tsin = taylor(sin(x), x, 0, 10)
psin = tsin.power_series(QQ)
print tsin
```

The Taylor series of order 10 is
\[
\sin(x) \approx x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362880}x^9 + O(x^{10}).
\]
To use as a polynomial we truncate.

```python
polsin = psin.truncate()
```

The polynomial is not such a good approximation for \( \sin(x) \).

```python
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. 4.9. Observe that after \( x = 5 \), the Taylor series diverges fast.

![Fig. 4.9: The plot of the sine function and a 10-th order Taylor approximation.](image)

We can compute a rational approximation for \( \sin(x) \), where degree of numerator and denominator are both of degree 7.

```python
z = PowerSeriesRing(QQ, 'z').gen()
pd = sin(z).pade(7,7)
print pd
```

The output is
\[
\frac{-479249/18361*z^7 + 7540776/2623*z^5 - 234376560/2623*z^3 + 1644477120/2623*z}{z^6 + 453960/2623*z^4 + 39702960/2623*z^2 + 1644477120/2623}.
\]
Fig. 4.10: The plot of the sine function and a 7/7-Padé approximation.
4.6.5 Limits

We can take limits with the limit command. For example, consider the limit definition of the transcendental constant $e = \lim_{k \to \infty} \left(1 + \frac{1}{k}\right)^k$.

```python
var('k')
f = (1+1/k)**k
f.limit(k=oo)
```

and as output we see $e$.

4.6.6 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 $\frac{x}{1-x-x^2}$ 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 $\frac{1}{\sqrt{1-4x}} = \sum_{n=0}^{\infty} \binom{2n}{n} x^n$ in the following way:

   (a) Compute a 7-th order Taylor series of $\frac{1}{\sqrt{1-4x}}$ about $x = 0$.

   (b) Compute $\sum_{n=0}^{\infty} \binom{2n}{n} x^n$ 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. 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 $[0, 2]$. 

CHAPTER 5

Indices and tables

- genindex
- modindex
- search
<table>
<thead>
<tr>
<th>A</th>
<th>absolute value, 12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>accuracy, 7</td>
</tr>
<tr>
<td></td>
<td>algebraic number, 11, 13</td>
</tr>
<tr>
<td></td>
<td>argument, 12</td>
</tr>
<tr>
<td></td>
<td>aspect_ratio, 66</td>
</tr>
<tr>
<td></td>
<td>assign, 15</td>
</tr>
<tr>
<td>C</td>
<td>CDF, 12</td>
</tr>
<tr>
<td></td>
<td>cell, 6</td>
</tr>
<tr>
<td></td>
<td>central differences, 71</td>
</tr>
<tr>
<td></td>
<td>Chebyshev polynomial, 63</td>
</tr>
<tr>
<td></td>
<td>closest exact number, 11</td>
</tr>
<tr>
<td></td>
<td>CoCalc, 4</td>
</tr>
<tr>
<td></td>
<td>color, 66</td>
</tr>
<tr>
<td></td>
<td>common denominator, 48</td>
</tr>
<tr>
<td></td>
<td>Complex Double Field, 12</td>
</tr>
<tr>
<td></td>
<td>complex number, 12</td>
</tr>
<tr>
<td></td>
<td>Computer Algebra, 3</td>
</tr>
<tr>
<td></td>
<td>continued fraction, 9</td>
</tr>
<tr>
<td></td>
<td>convergents, 9</td>
</tr>
<tr>
<td></td>
<td>counted with multiplicities, 12</td>
</tr>
<tr>
<td>D</td>
<td>declare variables, 15</td>
</tr>
<tr>
<td></td>
<td>derivative, 70, 71</td>
</tr>
<tr>
<td></td>
<td>dictionary, 16, 62</td>
</tr>
<tr>
<td></td>
<td>diff, 70</td>
</tr>
<tr>
<td></td>
<td>double float, 10</td>
</tr>
<tr>
<td>E</td>
<td>evaluation, 17</td>
</tr>
<tr>
<td></td>
<td>exact factorization, 51</td>
</tr>
<tr>
<td></td>
<td>expand, 50</td>
</tr>
<tr>
<td></td>
<td>expression swell, 50</td>
</tr>
<tr>
<td>F</td>
<td>factor, 13, 48</td>
</tr>
<tr>
<td></td>
<td>Fibonacci numbers, 60, 78</td>
</tr>
<tr>
<td></td>
<td>files, 29</td>
</tr>
<tr>
<td></td>
<td>filter, 66</td>
</tr>
<tr>
<td></td>
<td>formal root, 51</td>
</tr>
<tr>
<td></td>
<td>function, 71, 78</td>
</tr>
<tr>
<td>G</td>
<td>generating function, 78</td>
</tr>
<tr>
<td></td>
<td>generator, 79</td>
</tr>
<tr>
<td></td>
<td>get_systems, 5, 79</td>
</tr>
<tr>
<td></td>
<td>gradient, 70</td>
</tr>
<tr>
<td>H</td>
<td>Hessian, 70</td>
</tr>
<tr>
<td>I</td>
<td>I, 12</td>
</tr>
<tr>
<td></td>
<td>imaginary part, 12</td>
</tr>
<tr>
<td></td>
<td>imaginary unit, 12</td>
</tr>
<tr>
<td></td>
<td>implicit differentiation, 71</td>
</tr>
<tr>
<td></td>
<td>irrational number, 11</td>
</tr>
<tr>
<td></td>
<td>irreducible, 14</td>
</tr>
<tr>
<td>L</td>
<td>lambda, 66</td>
</tr>
<tr>
<td></td>
<td>list comprehension, 10, 65, 79</td>
</tr>
<tr>
<td>M</td>
<td>machine precision, 10, 19</td>
</tr>
<tr>
<td></td>
<td>memoization, 62</td>
</tr>
<tr>
<td></td>
<td>minimal polynomial, 13</td>
</tr>
<tr>
<td></td>
<td>multiplication table, 14</td>
</tr>
<tr>
<td></td>
<td>multiplicative inverse, 13, 79</td>
</tr>
<tr>
<td>N</td>
<td>numerical approximation, 6</td>
</tr>
<tr>
<td></td>
<td>numerical factorization, 51</td>
</tr>
<tr>
<td>P</td>
<td>Padé approximation, 80</td>
</tr>
</tbody>
</table>
pickling, 31
piecewise, 59
polar representation, 12
polynomial ring, 13
power series, 79
precision, 6, 80
preparse, 30
prevent evaluation, 13
prevent the evaluation, 18
Python files, 30
quotes, 18
random numbers, 19
random point, 72
rational approximation, 9
rational number, 9
real part, 12
RealField, 10
rectangular coordinates, 12
recurrence, 63
reset, 17, 31
restore, 15
Sage, 4
SageMath, 4
SageMath Cell Server, 4
SageMathCloud, 4
save, 30
sequential substitution, 72
serialization, 31
simultaneous substitution, 49, 72
single float, 10
slope, 72
solve, 15
square root, 12
SR, 13
string concatenation, 65
substitute, 16
substitution, 48
Symbolic Computation, 3
symbolic factorization, 51
symbolic ring, 13
SymPy, 79
syntactical substitution, 50
systems, 5
tangent line, 72
Taylor series, 78