Embedding Sage in LaTeX with SageTeX for TeXShop Users

Ron Bannon and others

August 28, 2014

Abstract

This document is designed to assist Essex County College students and faculty in the use of Sage. Questions about this document should be directed to:

Ron Bannon
Room 2210, 973-877-1886, b@nnon.us.

1 Getting Sage

Well, this should probably be a no-brainer at this stage in your development. I am guessing that anyone reading this document has already worked with Sage and then after looking around on the Sage forums you started to hear about SageTeX. And I am sure that SageTeX sounded nearly perfect[1] to anyone familiar with LaTeX and Sage, but knowing where to start was a bit of a mystery. I know when I first tried SageTeX, I failed! Then I went on to do a little bit of research and tinkering, and I was soon off and running SageTeX. The very first thing to make sure of is that you have the *application version*[2] of Sage installed, and if you don't, you need to do so now! You can get the Sage application at

`http://www.sagemath.org/download-mac.html`.

2 Optional Sage Tutorial

If you're already familiar with Sage and just looking to get started with SageTeX you should skip to the next section on configuring [§3] on configuring TeXShop to work with Sage.

[1] It's not, but in time, and with your help it will improve.
[2] It's the file version of Sage that ends in `app.dmg`.

2.1 Introduction

Quite simply, Sage is software that will allow you to explore many aspects of mathematics, including algebra, calculus, linear algebra, combinatorics, numerical mathematics, arbitrary precision arithmetic, number theory, statistics, stochastic models, and much more.

Fortunately, for us, Sage is *free* to use and is available from

`http://www.sagemath.org/`

as a free download. The founder and developers of Sage have a simple mission: "Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab." It does that, and probably more importantly, it may provide a basis for you to become involved in a rich tradition of intellectual *sharing* that is inherent to any scientific discipline, mathematics not withstanding. Unlike proprietary software, you have complete access to the source, and that's important. Not only can you look at the source code of Sage, but you can contribute to its growth by becoming a part of the Sage community of coders and users. Yes, just like any scientific endeavor, it requires a community to grow and I am hoping that some of you will prosper in this very vibrant community.

The originator of the Sage project, William Stein, is a mathematician at the University of Washington and he along with many others are intimately involved in making sure Sage remains open and free. Mathematics is open and free too—that is, the theorems of mathematics are shared and then vetted by many, and similarly Sage is too. You may not be interested in the backend of mathematics, that is the theorems, but they're there if you want to look at them. You may also not be interested in the backend of your software either, few are, but open source software is important because it has no secrets. Mathematics is not magic, and neither should your software!

Sage uses the Python programming language, supporting procedural, functional and object-oriented constructs. A working knowledge of Python is helpful when working with Sage, but is not required to start. Some knowledge of functional programming is desired, but again, Sage, like Python, can easily be used at the introductory level. As your skill-set with Sage develops, you may desire to install Sage on your personal machines. However, the materials that follow can be done immediately by visiting:

`https://sagecell.sagemath.org/`.

Yes, open this webpage now and do each sub-section that follows. It is enough to get you started on this Sage journey, but like most journeys worth taking I suggest you take the time to explore beyond what is presented in this brief introduction. If you're really perplexed you may decide to stop by Ron Bannon's office[3] to get a demo of what Sage can do. But again, you need to *try* and explore Sage by yourself. As in life, it is not enough to watch, so *please* continue to read on!

[3]Room 2210, Main Campus, 973-877-1886

2.2 Let's get started ...

Alright, time to move forward by doing some simple computations using Sage. Here I am using similar examples[4] to the ones that were developed by Mike O'Sullivan, Ryan Rosenbaum, and David Monarres. Again, the materials that follow can be done immediately by visiting:

`https://sagecell.sagemath.org/`.

If, after working through this guide, you like what you see, I strongly suggest you visit the Sage website,

`http://www.sagemath.org/`,

and download and install the software on your personal computer. Hey, it even works on the iPad.

2.3 Sage as a Calculator

This introductory tutorial examines commands that allow you to use Sage much like a graphing calculator. We'll be starting with some very basic arithmetic. Yes, this can be done on a basic calculator too, but Sage is so much more, so please hang in there.

2.3.1 Basic Arithmetic

The basic arithmetic operators are "+", "-", "*", and "/" for addition, subtraction, multiplication and division, while "^" or "**"is used for exponents.

```
sage: 15+16
31
sage: 189-762
-573
sage: 63*92
5796
sage: 1024/256
4
sage: 15/7
15/7
sage: 32^3
32768
sage: 2**9
512
```

The "-" symbol in front of a number indicates that it is negative.

```
sage: -78
```

[4] `http://www-rohan.sdsu.edu/~mosulliv/Courses/sdsu-sage-tutorial/index.html`

```
-78
sage: -99+47
-52
```

As we would expect, Sage adheres to the standard order of operations, PEMDAS (parenthesis, exponents, multiplication, division, addition, subtraction).

```
sage: 2*3^2-1
17
sage: (2*3)^2-1
35
sage: 2*3^(2-1)
6
sage: -5^2
-25
sage: (-5)^2
25
sage: (2+3)^2
25
sage: 2^2+3^2
13
```

When dividing two integers, there is a subtlety; whether Sage will return a fraction or it's decimal approximation. Like most advanced graphing calculators, Sage will attempt to be as *precise* as possible and will return the fraction unless told otherwise. One way to tell Sage that we *want* the decimal approximation is to include a decimal in the expression itself.

```
sage: 15/4
15/4
sage: 15.0/4
3.75000000000000
sage: 15/4.0
3.75000000000000
sage: 15/4*1.0
3.75000000000000
```

2.3.2 Exercises

1. Divide 28 by 2 raised to the 5^{th} power as a rational number, then get its decimal approximation.
2. Compute a decimal approximation of $\sqrt{2}$.
3. Compute $\sqrt{-9}$.

2.3.3 Answers to Exercises

```
sage: 28/2^5
7/8
sage: 28.0/2^5
0.875000000000000
sage: 2^(0.5)
1.41421356237310
sage: (-9)^(1/2)
3*I
```

2.3.4 Integer Division and Factoring

Sometimes when we divide, the division operator doesn't give us all of the information that we want. Often we would like to not just know what the reduced fraction is, or even it's decimal approximation, but rather the unique *quotient* and the *remainder* that are the consequence of the division.

To calculate the quotient we use the "`//`" operator and the "`%`" operator is used for the remainder.

```
sage: 256//97
2
sage: 256%97
62
```

If we want both the quotient and the remainder all at once, we use the `divmod` command.

```
sage: divmod(256,97)
(2, 62)
```

Recall that *b divides a* if 0 is the remainder when we divide the two integers. The integers in Sage have a built-in command (or 'method') which allows us to check whether one integer divides another. The method here is `.divides`.

```
sage: 3.divides(123456)
True
sage: 5.divides(123456)
False
```

A related command is the `.divisors` method. This method returns a list of all positive divisors of the integer specified.

```
sage: 123457.divisors()
[1, 123457]
sage: 125.divisors()
[1, 5, 25, 125]
```

When the divisors of an integer greater than 1 are only 1 and itself then we say that the number is *prime*. To check if a number is prime in Sage, we use the `.is_prime` method.

```
sage: (2^521-1).is_prime()
True
sage: 9876543210987654321.is_prime()
False
```

Notice the parentheses around $2^{521}-1$ in the first example. The parentheses are important to the order of operations in Sage, and if they are not included then Sage will compute something very different than we intended. Try evaluating this code (line 63) without the parentheses and notice the result. When in doubt, the judicious use of *parenthesis* is encouraged.

We use the `.factor` method to compute the *prime factorization* of an integer.

```
sage: 6743.factor()
11 * 613
sage: 9876543210987654321.factor()
3^2 * 17^2 * 101 * 3541 * 27961 * 379721
```

If we are interested in simply knowing which prime numbers divide an integer, we may use it's `.prime_divisors` or `.prime_factors` method.

```
sage: 9876543210987654321.prime_factors()
[3, 17, 101, 3541, 27961, 379721]
sage: (2^19-1).prime_divisors()
[524287]
```

Finally, we have the *greatest common divisor* and *least common multiple* of a pair of integers. A *common divisor* of two integers is any integer which is a divisor of each, whereas a *common multiple* is a number which both integers divide.

The greatest common divisor (gcd), not too surprisingly, is the largest of all of these common divisors. The `gcd()` command is used to calculate this divisor.

```
sage: gcd(65,640)
5
sage: gcd(2^19-1,1234567)
1
```

Notice that if two integers share no common divisors, then their gcd will be 1.

The least common multiple is the smallest integer which both integers divide. The `lcm()` command is used to calculate the least common multiple.

```
sage: lcm(12,8)
24
```

```
sage: lcm(123,321)
13161
```

2.3.5 Exercises

1. Find the quotient and remainder when divinding 98 into 956.
2. Verify that the quotient and remainder computed above are correct.
3. Determine if 3 divides 234878.
4. Compute the list of divisors for each of the integers 134, 491, 422 and 1002.
5. Which of the integers above are *prime*?
6. Calculate $\gcd(a,\ b)$, $\mathrm{lcm}(a,\ b)$ and $a \cdot b$ for the pairs of integers $(2,\ 5)$, $(4,\ 10)$ and $(18,\ 51)$. How do the gcd, lcm and the product of the numbers relate?

2.3.6 Answers to Exercises

```
sage: divmod(956,98)
(9, 74)
sage: 98*9+74
956
sage: 3.divides(234878)
False
sage: 134.divisors()
[1, 2, 67, 134]
sage: 491.divisors()
[1, 491]
sage: 422.divisors()
[1, 2, 211, 422]
sage: 1002.divisors()
[1, 2, 3, 6, 167, 334, 501, 1002]
sage: 491.is_prime()
True
sage: gcd(2,5)
1
sage: lcm(2,5)
10
sage: 2*5
10
```

```
sage: gcd(4,10)
2
sage: lcm(4,10)
20
sage: 4*10
40
sage: gcd(18,51)
3
sage: lcm(18,51)
306
sage: 18*51
918
```

The `gcd(a, b) * lcm(a, b) = a * b`.

2.3.7 Standard Functions and Constants

Sage includes nearly all of the standard functions that one encounters when studying mathematics. In this section, we shall cover some of the most commonly used mathematical functions: the *maximum*, *minimum*, *floor*, *ceiling*, *trigonometric*, *exponential*, and *logarithm* functions. We will also see many of the standard mathematical constants; such as *Euler's constant* (e), π, and *the golden ratio* (ϕ).

The `max()` and `min()` commands return the largest and smallest of a set of numbers. We may input any number of arguments into the `max()` and `min()` functions.

```
sage: max(1,2,0,-19)
2
sage: min(1,2,0,-19)
-19
sage: max(1/3,56/167)
56/167
```

In Sage we use the `abs()` command to compute the *absolute value* of a real number.

```
sage: abs(-10)
10
sage: abs(15-32)
17
```

The `floor()` command rounds a number down to the nearest integer, while `ceil()` rounds up.

```
sage: floor(3.6)
3
```

```
sage: ceil(3.6)
4
sage: floor(-5.1)
-6
sage: ceil(-5.1)
-5
```

We need to be very careful while using `floor()` and `ceil()`.

```
sage: floor(1/(2.1-2))
9
```

This is clearly not correct: $\lfloor 1/(2.1-2) \rfloor = \lfloor 1/0.1 \rfloor = \lfloor 10 \rfloor = 10$. So what happened?

```
sage: 1/(2.1-2)
9.99999999999999
```

Computers store real numbers in *binary*, while we are accustomed to using the decimal representation. The 2.1 in decimal notation is quite simple and short, but when converted to binary it is $10.0\overline{0011}_2 = 10.0001100110011_2\ldots$.

Since computers cannot store an infinite number of digits, this gets rounded off somewhere, resulting in the slight error we saw. In Sage, however, *rational numbers* (fractions) are exact, so we will never see this rounding error.

```
sage: floor(1/(21/10-2))
10
```

Due to this, it is often a good idea to use rational numbers whenever possible instead of decimals, particularly if a high level of precision is required.

The `sqrt` command calculates the *square root* of a real number. As we have seen earlier with fractions, if we want a decimal approximation we can get this by giving a decimal number as the input.

```
sage: sqrt(15)
sqrt(15)
sage: 15.sqrt() # the command is also a method
sqrt(15)
sage: sqrt(15.0)
3.87298334620742
```

To compute other roots, we use a rational exponent. Sage can compute any rational power. If either the exponent or the base is a decimal then the output will be a decimal.

```
sage: 3^(1/2)
sqrt(3)
sage: (3.0)^(1/2)
1.73205080756888
```

```
sage: 8^(1/2)
2*sqrt(2)
sage: 8^(1/3)
2
```

Sage also has available all of the standard trigonometric functions: for sine and cosine we use `sin()` and `cos()`

```
sage: sin(1)
sin(1)
sage: sin(1.0)
0.841470984807897
sage: cos(3/2)
cos(3/2)
sage: cos(3/2.0)
0.0707372016677029
```

Again we see the same behavior that we saw with `sqrt()`, Sage will give us an exact answer. You might think that since there is no way to simplify $\sin(1)$, why bother? Well, some expressions involving sine can indeed be simplified. For example, an important identity from geometry is $\sin(\pi/3) = \sqrt{3}/2$. Sage has a built-in symbolic π, and understands this identity.

```
sage: pi
pi
sage: sin(pi/3)
1/2*sqrt(3)
sage: cos(pi/3)
1/2
sage: tan(pi/3)
sqrt(3)
```

When we type `pi` in Sage we are dealing exactly with π, not some numerical approximation. However, we can call for a numerical approximation using the `.n` method.

```
sage: pi.n()
3.14159265358979
sage: sin(pi)
0
sage: sin(pi.n())
1.22464679914735e-16
```

We see that when using the symbolic `pi`, Sage returns the exact result. However, when we use the approximation we get an approximation back. `e-15` is a shorthand for 10^{-15} and the number `1.22464679914735e-16` should be zero, but there are errors introduced by the approximation. Here are a few examples of using the symbolic, precise π versus the numerical approximation.

```
sage: sin(pi/6)
1/2
sage: sin(pi.n()/6)
0.500000000000000
sage: sin(pi/4)
1/2*sqrt(2)
sage: sin(pi.n()/4)
0.707106781186547
```

Continuing on with the theme, there are some lesser known special angles for which the value of sine or cosine can be cleverly simplified.

```
sage: sin(pi/10)
1/4*sqrt(5) - 1/4
sage: cos(pi/5)
1/4*sqrt(5) + 1/4
sage: sin(5*pi/12)
1/12*sqrt(6)*(sqrt(3) + 3)
```

Other trigonometric functions, the inverse trigonometric functions and hyperbolic functions are also available.

```
sage: arctan(1.0)
0.785398163397448
sage: sinh(9.0)
4051.54190208279
```

Similar to `pi`, Sage has a built-in symbolic constant for the number e, the base of the natural logarithm.

```
sage: e
e
sage: e.n()
2.71828182845905
```

While some might be familiar with using $\ln(x)$ for natural logarithm base e and $\log(x)$ to represent common logarithm base 10, in Sage both represent logarithm base e. We may specify a different base as a second argument to the command: to compute $\log_b(x)$ in Sage we use the command `log(x, b)`.

```
sage: ln(e)
1
sage: log(e)
1
sage: log(e^2)
2
sage: log(10)
log(10)
```

```
sage: log(10.0)
2.30258509299405
sage: log(100,10)
2
```

Exponentiation base e can done using both the `exp()` function and by raising the symbolic constant e to a specified power.

```
sage: exp(2)
e^2
sage: exp(2.0)
7.38905609893065
sage: exp(log(pi))
pi
sage: e^(log(2))
2
```

As an aside, the following additional standard mathematical constants are defined inSage.

```
sage: NaN
NaN
sage: golden_ratio
golden_ratio
sage: log2
log2
sage: euler_gamma
euler_gamma
sage: catalan
catalan
sage: khinchin
khinchin
sage: twinprime
twinprime
sage: mertens
mertens
sage: brun
brun
```

2.3.8 Exercises

1. Compute the floor and ceiling of -3.56.

2. Compute the logarithm base e of 100, compute the logarithm base 10 of 100, then compute the ratio. What should the answer be?

3. Compute the logarithm base 2 of 64.
4. Compare $e^{i\pi}$ with a numerical approximation of it using `pi.n()`.
5. Compute $\sin(\pi/2)$, $\cot(\pi/4)$ and $\csc(\pi/16)$.

2.3.9 Answers to Exercises

```
sage: floor(-3.56)
-4
sage: ceil(-3.56)
-3
sage: log(100.0)
4.60517018598809
sage: log(100.0,10)
2.00000000000000
sage: log(100.0)/log(100.0,10)
2.30258509299405
sage: log(10.0)
2.30258509299405
sage: log(64.0,2)
6.00000000000000
sage: e^(I*pi)
-1
sage: e^(I*pi.n())
-1.00000000000000 + 1.22464679914735e-16*I
sage: sin(pi/2)
1
sage: cot(pi/4)
1
sage: csc(pi/16)
csc(1/16*pi)
sage: csc(pi/16).n()
5.12583089548301
```

2.3.10 Solving Equations and Inequalities

In Sage, equations and inequalities are defined using the *operators* "`==`", "`<=`", and "`>=`" and will return either `True`, `False`, or, if there is a variable, just the equation/inequality.

```
sage: 9 == 9
True
sage: 9 <= 10
```

```
True
sage: 3*x - 10 == 5
3*x - 10 == 5
```

To solve an equation or an inequality we use using the, aptly named, `solve()` command. For the moment, we will only solve for x. The section on variables below explains how to use other variables.

```
sage: solve(3*x - 2 == 5, x)
[
x == (7/3)
]
sage: solve( 2*x - 5 == 1, x)
[
x == 3
]
sage: solve( 2*x - 5 >= 17,x)
[[x >= 11]]
sage: solve( 3*x - 2 > 5, x)
[[x > (7/3)]]
```

Equations can have multiple solutions, Sage returns all solutions found as a list. You should note that I can easily access elements from the list.

```
sage: solve( x^2 + x  == 6, x)
[
x == -3,
x == 2
]
sage: solve(2*x^2 - x + 1 == 0, x)
[
x == -1/4*I*sqrt(7) + 1/4,
x == 1/4*I*sqrt(7) + 1/4
]
sage: soln = solve(2*x^2 - x + 1 == 0, x)
sage: soln
[
x == -1/4*I*sqrt(7) + 1/4,
x == 1/4*I*sqrt(7) + 1/4
]
sage: soln[0]
x == -1/4*I*sqrt(7) + 1/4
sage: soln[1]
x == 1/4*I*sqrt(7) + 1/4
sage: soln[0].rhs()
-1/4*I*sqrt(7) + 1/4
sage: soln[1].rhs()
```

```
1/4*I*sqrt(7) + 1/4
sage: solve( exp(x) == -1, x)
[
x == I*pi
]
```

The solution set of certain inequalities consists of the union and intersection of open intervals.

```
sage: solve( x^2 - 6 >= 3, x )
[[x <= -3], [x >= 3]]
sage: solve( x^2 - 6 <= 3, x )
[[x >= -3, x <= 3]]
```

The `solve()` command will attempt to express the solution of an equation without the use of floating point numbers. If this cannot be done, it will return the solution in a symbolic form.

```
sage: solve( sin(x) == x, x)
[
x == sin(x)
]
sage: solve( exp(x) - x == 0 , x)
[
x == e^x
]
sage: solve( cos(x) - sin(x) == 0 , x)
[
sin(x) == cos(x)
]
sage: solve( cos(x) - exp(x) == 0 , x)
[
cos(x) == e^x
]
```

To find a numeric approximation of the solution we can use the `find_root()` command. Which requires both the equation and a closed interval on which to search for a solution.

```
sage: find_root(sin(x) == x, -pi/2 , pi/2)
0.0
sage: find_root(sin(x) == cos(x), pi, 3*pi/2)
3.92699081699
```

This command will only return one solution on the specified interval, if one exists. It will not find the complete solution set over the entire real numbers. To find a complete set of solutions, the reader must use `find_root()` repeatedly over cleverly selected intervals. Sadly, at this point, Sage cannot do all of the

thinking for us. This feature is not planned until Sage 10.

2.3.11 Declaring Variables

In the previous section we only solved equations in one variable, and we always used x. When a session is started, Sage creates one symbolic variable, x, and it can be used to solve equations. If you want to use an additional symbolic variable, you have to *declare it* using the `var()` command. The name of a symbolic variable can be a letter, or a combination of letters and numbers:

```
sage: y, z, t = var('y', 'z', 't')
sage: phi, theta, rho = var('phi', 'theta', 'rho')
sage: x1, x2 = var('x1', 'x2')
```

Variable names cannot contain spaces, for example `square root` is not a valid variable name, whereas `square_root` is.

Attempting to use a symbolic variable before it has been declared will result in a `NameError` message. In short, you must declare variables before using them! The only exception here is the variable x. We can un-declare a symbolic variable, like the variable `phi` defined above, by using the `restore('phi')` command.

2.3.12 Solving Equations with Several Variables

Small systems of linear equations can be also solved using `solve()`, provided that all the symbolic variables have been declared. The equations must be input as a list, followed by the symbolic variables. The result may be either a unique solution, infinitely many solutions, or no solutions at all.

```
sage: solve( [3*x - y == 2, -2*x -y == 1 ], x, y)
[
[x == (1/5), y == (-7/5)]
]
sage: solve( [2*x + y == -1 , -4*x - 2*y == 2], x, y)
[
[x == -1/2*r1 - 1/2, y == r1]
]
sage: solve( [2*x - y == -1 , 2*x - y == 2], x, y)
[

]
```

In the second equation above, `r1` signifies that there is a free variable which parametrizes the solution set. When there is more than one free variable, Sage enumerates them `r1, r2, . . . , rk`.

```
sage: solve([ 2*x + 3*y + 5*z == 1, 4*x + 6*y + 10*z
   == 2, 6*x + 9*y + 15*z == 3], x,y,z)
```

```
[
[x == -5/2*r2 - 3/2*r3 + 1/2, y == r3, z == r2]
]
```

`solve()` can be very slow for large systems of equations. For these systems, it is best to use the linear algebra functions as they are quite efficient.

Solving inequalities in several variables can lead to complicated expressions, since the regions they define are complicated. In the example below, Sage's solution is a list containing the point of intersection of the lines, then two rays, then the region between the two rays.

```
sage: solve([ x-y >=2, x+y <=3], x,y)
[
[x == (5/2), y == (1/2)],
[x == -y + 3, y < (1/2)],
[x == y + 2, y < (1/2)],
[y + 2 < x, x < -y + 3, y < (1/2)]
]
sage: solve([ 2*x-y< 4, x+y>5, x-y<6], x,y)
[
[-y + 5 < x, x < 1/2*y + 2, 2 < y]
]
```

2.3.13 Exercises

1. Find all of the solutions to the equation $x^3 - x = 7x^2 - 7$.
2. Find the complete solution set for the inequality $|t - 7| \geq 3$.
3. Find all x and y that satisfy both $2x + y = 17$ and $x - 3y = -16$.
4. Use `find_root()` to find a solution of the equation $e^x = \cos(x)$ on the interval $[-\pi/2,\ 0]$.
5. Change the command above so that `find_root` finds the other solution in the same interval.

2.3.14 Answers to Exercises

```
sage: solve(x^3-x==7*x^2-7,x)
[
x == 7,
x == -1,
x == 1
]
```

```
sage: t = var('t')
sage: solve(abs(t-7)>=3,t)
[[t == 10], [t == 4], [t < 4], [10 < t]]
sage: y = var('y')
sage: solve([2*x+y==17,x-3*y==-16],x,y)
[
[x == 5, y == 7]
]
sage: find_root(e^x==cos(x),-pi/2,0)
0.0
sage: find_root(e^x==cos(x),-pi/2,-1)
-1.29269571937
```

2.3.15 Calculus Commands

Sage has many commands that are useful for the study of differential and integral calculus. We will begin our investigation of these command by defining a few functions that we will use throughout the section.

```
sage: f(x) = x*exp(x)
sage: f
x |--> x*e^x
sage: g(x) = (x^2)*cos(2*x)
sage: g
x |--> x^2*cos(2*x)
sage: h(x) = (x^2 + x - 2)/(x-4)
sage: h
x |--> (x^2 + x - 2)/(x - 4)
```

Sage uses `x |-->` to tell you that the expression returned is actually a function and not just a number or string. This means that we can *evaluate* these expressions just like you would expect of any function.

```
sage: f(1)
e
sage: g(2*pi)
4*pi^2
sage: h(-1)
2/5
```

With these functions defined, we will look at how we can use Sage to compute the *limit* of these functions.

2.3.16 Limits

The limit of $f(x) = xe^x$ as $x \to 1$ is computed in Sage by entering the following command:

```
sage: limit(f, x=1)
x |--> e
```

We can do the same with $g(x)$. To evaluate the limit of $g(x) = x^2 \cos(2x)$ as $x \to 2$ we enter the following command:

```
sage: limit(g, x=2)
x |--> 4*cos(4)
```

The functions $f(x)$ and $g(x)$ aren't all that exciting as far as limits are concerned since they are both *continuous* for all real numbers. But $h(x)$ has a discontinuity at $x = 4$, so to investigate what is happening near this discontinuity we will look at the limit of $h(x)$ as $x \to 4$:

```
sage: limit(h, x=4)
x |--> Infinity
```

Now this is an example of why we have to be a little careful when using computer algebra systems. The limit above is not exactly correct. See the graph (Figure 1, page 19) of $h(x)$ near this discontinuity below.

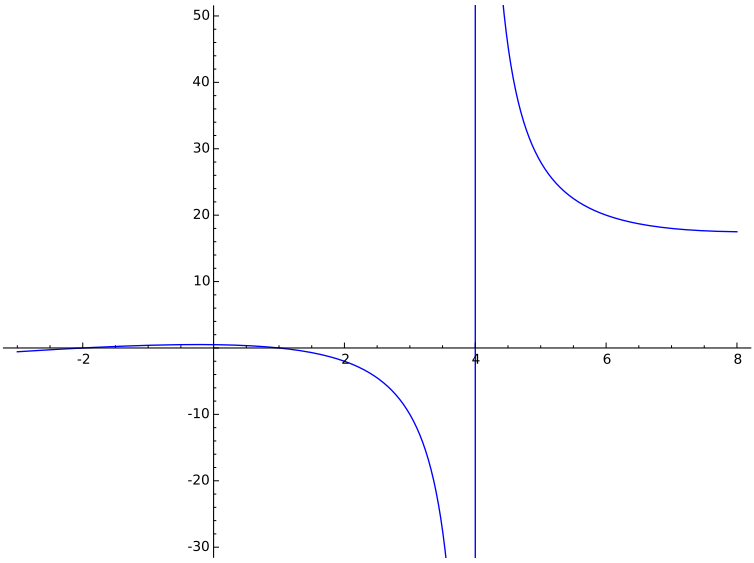


Figure 1: Partial graph of $h(x)$ generated using SageTEX.

What we have when $x = 4$ is a *vertical asymptote* with the function tending

toward *positive* infinity if x is larger than 4 and *negative* infinity from when x less than 4. We can takes these *directional* limits using Sage to confirm this by supplying the extra `dir` argument.

```
sage: limit(h, x=4, dir='right')
x |--> +Infinity
sage: limit(h, x=4, dir='left')
x |--> -Infinity
```

2.3.17 Derivatives

The next thing we are going to do is use Sage to compute the *derivatives* of the functions that we defined earlier. For example, to compute $f'(x)$, $g'(x)$, and $h'(x)$ we will use the `derivative()` command.

```
sage: fp  =  derivative(f,x)
sage: fp
x |--> x*e^x + e^x
sage: gp =  derivative(g, x)
sage: gp
x |--> -2*x^2*sin(2*x) + 2*x*cos(2*x)
sage: hp  = derivative(h,x)
sage: hp
x |--> (2*x + 1)/(x - 4) - (x^2 + x - 2)/(x - 4)^2
```

The first argument is the function which you would like to differentiate and the second argument is the variable with which you would like to differentiate with respect to. For example, if I were to supply a different variable, Sage will hold x constant and take the derivative with respect to that variable.

```
sage: y = var('y')
sage: derivative(f,y)
x |--> 0
sage: derivative(g,y)
x |--> 0
sage: derivative(h,y)
x |--> 0
```

The `derivative()` command returns another function that can be evaluated like any other function.

```
sage: fp(10)
11*e^10
sage: gp(pi/2)
-pi
sage: hp(10)
```

```
1/2
```

With the *derivative function* computed, we can then find the *critical points* using the `solve()` command.

```
sage: solve( fp(x) == 0, x)
[
x == -1
]
sage: solve( hp(x) == 0, x)
[
x == -3*sqrt(2) + 4,
x == 3*sqrt(2) + 4
]
sage: solve( gp(x) == 0, x)
[
x == 0,
x == cos(2*x)/sin(2*x)
]
sage: find_root(gp(x)==0,-1,-0.5)
-0.538436993156
```

Constructing the line *tangent* to our functions at the point $(x, f(x))$ is an important computation which is easily done in Sage. For example, the following command will compute the line tangent to $f(x)$ at the point $(0, f(0))$.

```
sage: Tf = fp(0)*( x - 0 ) + f(0)
sage: Tf
x
```

The same can be done for $g(x)$ and $h(x)$.

```
sage: Tg = gp(0)*( x - 0 ) + g(0)
sage: Tg
0
sage: Th = hp(0)*( x - 0 ) + h(0)
sage: Th
-1/8*x + 1/2
```

2.3.18 Integrals

Sage has the facility to compute both *definite* and *indefinite* integral for many common functions. We will begin by computing the *indefinite* integral, otherwise known as the *anti-derivative*, for each of the functions that we defined earlier. This will be done by using the `integral()` command which has arguments that are similar to `derivative()`.

```
sage: integral(f, x)
x |--> (x - 1)*e^x
sage: integral(g, x)
x |--> 1/2*x*cos(2*x) + 1/4*(2*x^2 - 1)*sin(2*x)
sage: integral(h, x)
x |--> 1/2*x^2 + 5*x + 18*log(x - 4)
```

The function that is returned is only *one* of the many anti-derivatives that exist for each of these functions. The others differ by a constant. We can verify that we have indeed computed the *anti-derivative* by taking the derivative of our indefinite integrals.

```
sage: derivative(integral(f,x), x )
x |--> (x - 1)*e^x + e^x
sage: f
x |--> x*e^x
sage: derivative(integral(g,x), x )
x |--> 1/2*(2*x^2 - 1)*cos(2*x) + 1/2*cos(2*x)
sage: derivative(integral(h,x), x )
x |--> x + 18/(x - 4) + 5
```

Wait, none of these look right. But a little algebra, and the use of a trig-identity or two in the case of `1/2*(2*x^2 - 1)*cos(2*x) + 1/2*cos(2*x)`, you will see that they are indeed the same.

It should also be noted that there are some functions which are continuous and yet there doesn't exist a *closed form* antiderivative. A common example is e^{-x^2} which forms the basis for the *normal distribution* which is ubiquitous throughout statistics. The antiderivative for $2e^{-x^2}/\sqrt{\pi}$ is commonly called $\operatorname{erf}(x)$, otherwise known as the *error function*.

```
sage: y(x) = exp(-x^2)
sage: integral(y,x)
x |--> 1/2*sqrt(pi)*erf(x)
```

We can also compute the *definite* integral for the functions that we defined earlier. This is done by specifying the *limits of integration* as addition arguments.

```
sage: integral(f,x,0,1)
1
sage: integral(g,x,0,1)
1/2*cos(2) + 1/4*sin(2)
sage: integral(h,x,0,1)
-18*log(4) + 18*log(3) + 11/2
```

In each case above, Sage returns a *function* as its result. Each of these functions is a constant function, which is what we would expect. As it was pointed out earlier, Sage will return the expression that retains the most precision and

will not use decimals unless told to. A quick way to tell Sage that an approximation is desired is wrap the `integrate()` command with `n()`, the numerical approximation command.

```
sage: n(integral(f, x, 0, 1))
1.00000000000000
sage: n(integral(g, x, 0, 1))
0.0192509384328492
sage: n(integral(h, x, 0, 1))
0.321722695867944
```

2.3.19 Exercises

1. Compute.

$$\lim_{x \to 2} \frac{x^2 + 2\,x - 8}{x - 2}$$

2. Compute.

$$\lim_{x \to (\pi/2)^+} \sec(x)$$

3. Compute.

$$\lim_{x \to (\pi/2)^-} \sec(x)$$

4. Compute.

$$\frac{\mathrm{d}}{\mathrm{d}x} \left[x^2 e^{3x} \cos(2x)\right]$$

5. Compute.[5]

$$\frac{\mathrm{d}}{\mathrm{d}t} \left[\frac{t^2 + 1}{t - 2}\right]$$

6. Compute.

$$\frac{\mathrm{d}}{\mathrm{d}y} \left[x \cos(x)\right]$$

7. Compute.

$$\int \frac{x + 1}{x^2 + 2x + 1}\, \mathrm{d}x$$

[5] Remember to define t.

8. Compute.

$$\int_{-\pi/4}^{\pi/4} \sec(x)\ \mathrm{d}x$$

9. Compute.

$$\int xe^{-x^2}\ \mathrm{d}x$$

2.3.20 Answers to Exercises

```
sage: limit((x^2+2*x-8)/(x-2),x=2)
6
sage: limit(sec(x),x=pi/2,dir='right')
-Infinity
sage: limit(sec(x),x=pi/2,dir='left')
+Infinity
sage: derivative(x^2*e^(3*x)*cos(2*x),x)
3*x^2*cos(2*x)*e^(3*x) - 2*x^2*e^(3*x)*sin(2*x) + 2*x
   *cos(2*x)*e^(3*x)
sage: t=var('t')
sage: derivative((t^2+1)/(t-2),t)
2*t/(t - 2) - (t^2 + 1)/(t - 2)^2
sage: y=var('y')
sage: derivative(x*cos(x),y)
0
sage: integral((x+1)/(x^2+2*x+1),x)
1/2*log(x^2 + 2*x + 1)
sage: integral(sec(x),x,0,pi/4)*2
log(1/2*sqrt(2) + 1) - log(-1/2*sqrt(2) + 1)
sage: integral(x*e^(-x^2),x)
-1/2*e^(-x^2)
```

2.3.21 Statistics

In this section we will discuss the use of some of the basic descriptive statistic functions availble for use in Sage.

To demonstrate their usage we will first generate a pseudo-random list of integers from 1 to 1000 to be used as a data set. The `randint(a, b)` function generates a random integer from $[a,\ b]$. Note, by the nature of random number generation your list of numbers will be different.

```
sage: data=[Integer(randint(1,1000)) for i in range(20)]
sage: data
[180, 521, 786, 766, 151, 993, 791, 899, 168, 907, 603, 475, 940, 631, 802, 158, 36, 928, 184, 884]
```

We can compute the mean, median, mode, variance, and standard deviation of this data.

```
sage: mean(data)
11803/20
sage: median(data)
1397/2
sage: mode(data)
[928, 993, 802, 899, 36, 475, 168, 521, 791, 907, 184, 786, 180, 158, 151, 940, 631, 884, 603, 766]
sage: variance(data)
41068251/380
sage: std(data)
3/2*sqrt(4563139/95)
```

Note that both the standard deviation and variance are computed in their unbiased forms. If we want to bias these measures then you can use the `bias=True` option.

```
sage: variance(data,bias=True)
41068251/400
sage: std(data,bias=True)
3/20*sqrt(4563139)
```

We can also compute a rolling, or moving, average of the data with the `moving_average` command.

```
sage: moving_average(data,4)
[2253/4, 556, 674, 2701/4, 1417/2, 2851/4, 2765/4, 2577/4, 2153/4, 2925/4, 2649/4, 712, 2531/4, 1627/4, 481, 653/2, 508]
sage: moving_average(data,10)
[3081/5, 1317/2, 6539/10, 6693/10, 3279/5, 7209/10, 3187/5, 5619/10, 2824/5, 2832/5, 5641/10]
sage: moving_average(data,20)
[11803/20]
```

2.3.22 Exercises

1. Generate a list of 1000 random integers on the interval $[-100, 100]$ and compute the average.
2. The heights of eight students, measured in inches, are 71, 73, 59, 62, 65, 61, 73, 61. Find the *average*, *median* and *mode* of the heights of these students.
3. Using the same data, compute the *standard deviation* and *variance* of the sampled heights.
4. Find the *range* of the heights.[6]

2.3.23 Answers to Exercises

```
sage: mean([Integer(randint(-100,100)) for i in range(1000)])
-41/250
sage: data=[71, 73, 59, 62, 65, 61, 73, 61]
sage: mean(data)
525/8
sage: median(data)
127/2
sage: mode(data)
[73, 61]
sage: variance(data)
1903/56
sage: std(data)^2
1903/56
sage: std(data)
1/2*sqrt(1903/14)
sage: max(data)-min(data)
14
```

2.3.24 2D Plotting

Sage has many ways for us to visualize the mathematics with which we are working. In this section we will quickly get you up to speed with some of the basic commands used when plotting functions and working with graphics.

To produce a basic plot of $\sin(x)$ from $x = -\frac{\pi}{2}$ to $x = \frac{\pi}{2}$ we will use the `plot()` command.

```
sage: f(x) = sin(x)
```

[6] Hint: Use the `max()` and `min()` commands.

```
sage: p = plot(f(x), (x, -pi/2, pi/2))
sage: p.show()
None
```

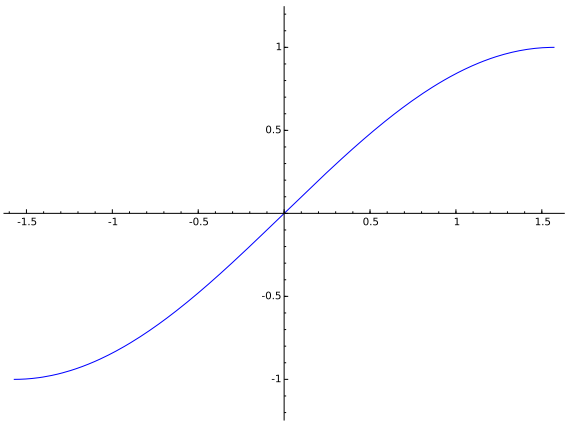


Figure 2: Partial graph of $f(x)$ generated using SageTEX.

By default, the plot (Figure 2, page 27) created will be quite plain. To add axis labels and make our plotted line purple, we can alter (Figure 3, page 28) the plot attribute by adding the `axes_labels` and `color` options.

```
sage: p = plot(f(x), (x,-pi/2, pi/2), axes_labels=['x
    ','sin(x)'], color='purple')
sage: p.show()
None
```

The `color` option accepts string color designations (`purple, green, red, black`, etc. ...), an RGB triple such as `(0.25,0.10,1)`, or an HTML-style hex triple such as `#ff00aa`.

We can change (Figure 4, page 28) the style of line, whether it is solid, dashed, and it's thickness by using the `linestyle` and the `thickness` options.

```
sage: p = plot(f(x), (x,-pi/2, pi/2), linestyle='--',
    thickness=3)
sage: p.show()
None
```

We can display the graphs of two functions on the same axes (Figure 5, page 29) by adding the plots together.

```
sage: f(x) = sin(x)
sage: g(x) = cos(x)
sage: p = plot(f(x),(x,-pi/2,pi/2), color='black')
```

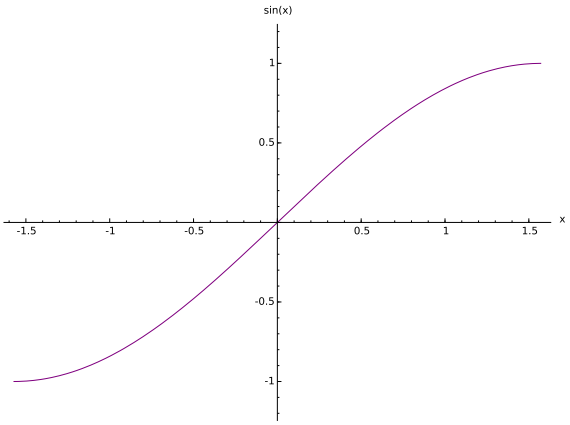


Figure 3: Partial graph of $f(x)$ generated using SageTEX.

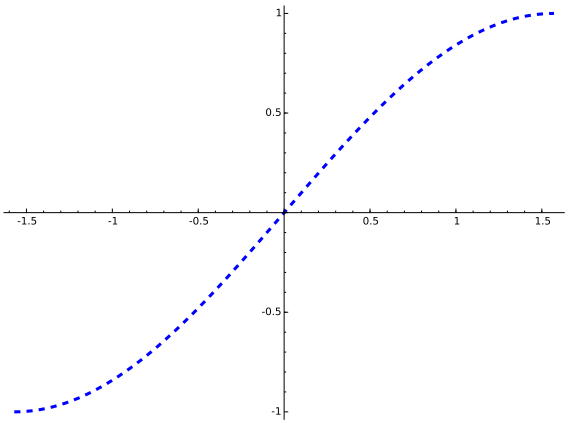


Figure 4: Partial graph of $f(x)$ generated using SageTEX.

```
sage: q = plot(g(x), (x,-pi/2, pi/2), color='red')
sage: r = p + q
sage: r.show()
None
```

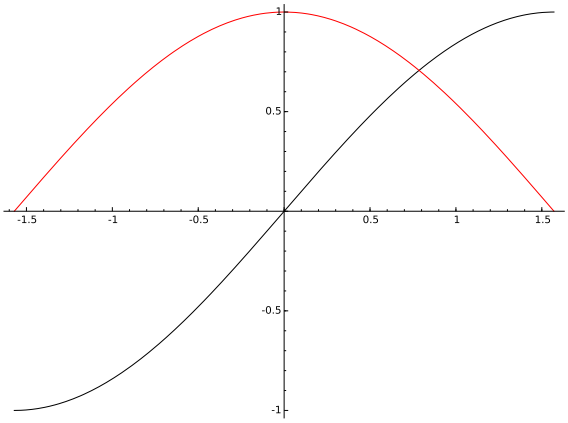


Figure 5: Partial graph of $f(x)$ and $g(x)$ generated using SageTEX.

To tie together our plotting commands with some material we have learned earlier, let's use the `.find_root` command to find the point where $\sin(x)$ and $\cos(x)$ intersect. We will then add this point to the graph (Figure 6, page 30) and label it.

```
sage: find_root( sin(x) == cos(x),-pi/2, pi/2 )
0.785398163397
sage: P = point( [(0.785, sin(0.785))] )
sage: sin(0.785)
0.706825181105366
sage: T = text("(0.79, 0.71)", (0.79, .71 - .20))
sage: s = P + r + T
sage: s.show()
None
```

Sage handles many of the details of producing *nice* looking plots in a way that is transparent to the user. However there are times in which Sage will produce (Figure 7, page 30) a plot which isn't quite what we were expecting.

```
sage: f(x) = (x^3 + x^2 + x)/(x^2 - x - 2 )
sage: p = plot(f(x), (x, -5,5))
sage: p.show()
None
```

The vertical asymptotes of this rational function cause Sage to adjust the

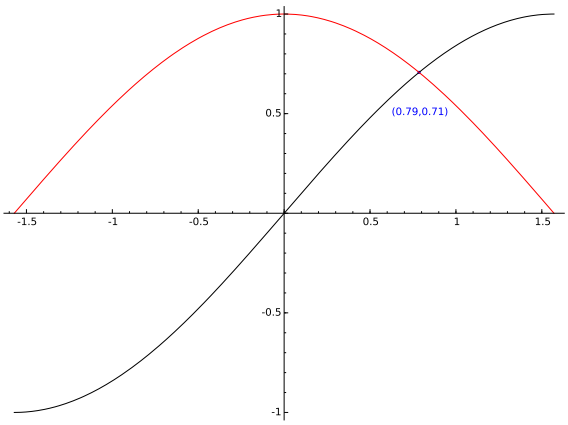


Figure 6: Partial labeled graph of $f(x)$ and $g(x)$ generated using SageTEX.

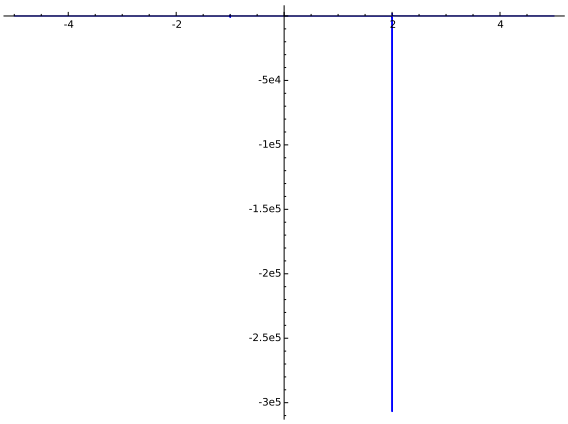


Figure 7: Partial poorly rendered graph of $f(x)$ generated using SageTEX.

aspect ratio of the plot to display the rather large y values near $x = -1$ and $x = 2$. This obfuscates most of the features of this function in a way that we may have not intended. To remedy this we can explicitly adjust (Figure 8, page 31) the vertical and horizontal limits of our plot.

```
sage: f(x) = (x^3 + x^2 + x)/(x^2 - x -2 )
sage: p = plot(f(x), xmin=-2, xmax=4, ymin=-20, ymax
    =20)
sage: p.show()
None
```

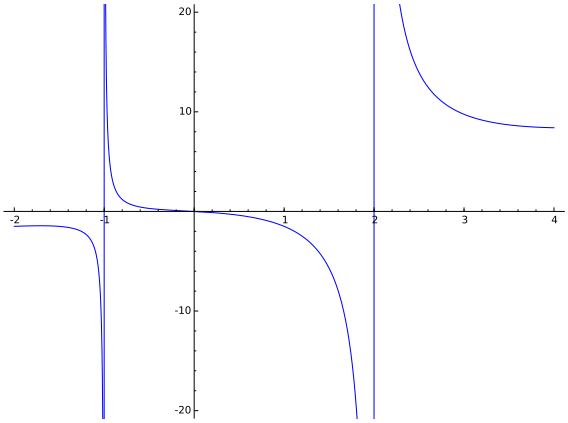


Figure 8: Partial graph of $f(x)$ generated using SageTEX.

This (Figure 8, page 31) displays the features of this particular function in a much more pleasing and informative fashion.

Sage can handle parametric plots (Figure 9, page 32) with the `parametric_plot()` command. The following is a simple circle of radius 3.

```
sage: t = var('t')
sage: p = parametric_plot( [3*cos(t),3*sin(t)],(t
    ,0,2*pi))
sage: p.show()
None
```

The default choice of aspect ratio (depending on your OS)[7] may make the plot above decidedly *un-circle like*. We can adjust this by using the `aspect_ratio` option.

```
sage: p.show(aspect_ratio=1)
None
```

[7] I am using Mac OS X and TEXShop to generate this document and it renders this graph properly.

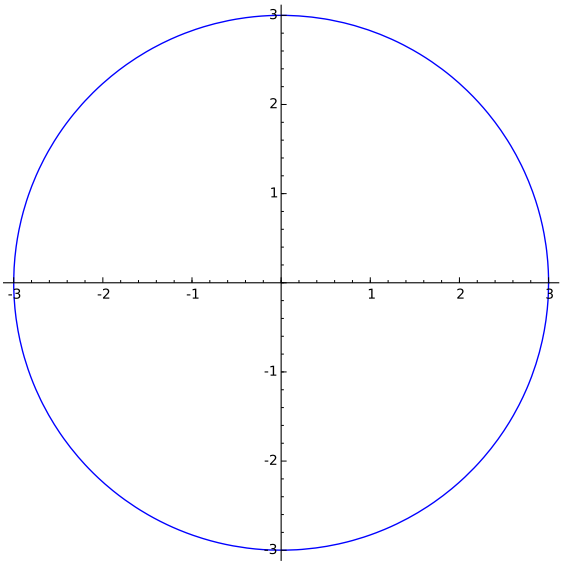


Figure 9: Parametric graph generated using SageTEX.

The different plotting commands accept many of the same options as plot. The following generates the Lissajous Curve $L(3,\ 2)$ (Figure 10, page 33) with a thick red dashed line.

```
sage: p = parametric_plot( [sin(3*t), sin(2*t)], (t,    582
    0, 3*pi), thickness=2,color='red',linestyle='--')
sage: p.show()                                          583
None                                                    584
```

Polar plots (Figure 11, page 33) can be done using the `.polar_plot` command.

```
sage: theta = var('theta')                              585
sage: r(theta) = sin(4*theta)                           586
sage: p = polar_plot((r(theta)), (theta, 0, 2*pi))      587
sage: p.show()                                          588
None                                                    589
```

And finally, Sage can do the plots (Figure 12, page 34) for functions that are implicitly defined. For example, to display the points (x, y) that satisfy the equation $4x^2y - 3y = x^3 - 1$, we enter the following.

```
sage: y = var('y')                                      590
sage: p = implicit_plot(4*x^2*y - 3*y == x^3 - 1, (x    591
    ,-10,10),(y,-10,10))
```

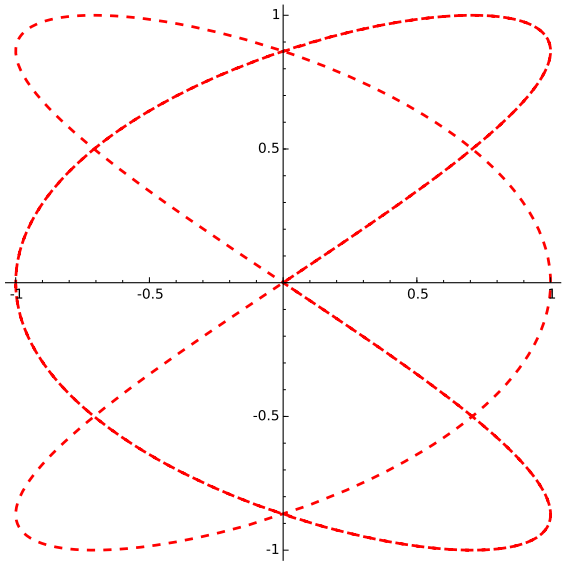


Figure 10: Lissajous Curve generated using SageTEX.

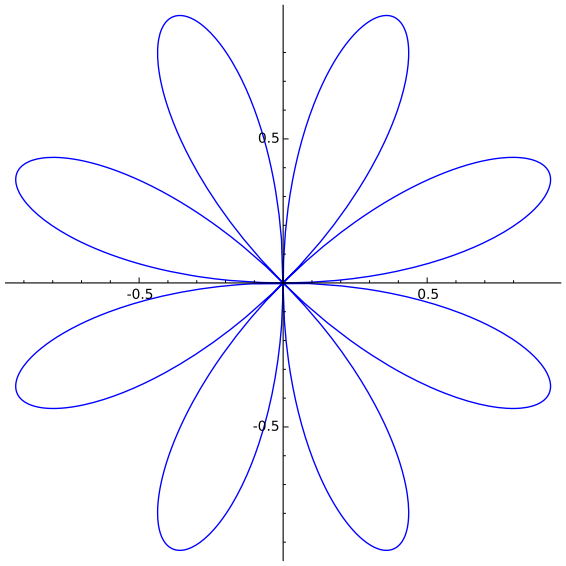


Figure 11: Polar graph generated using SageTEX.

```
sage: p.show()
None
```

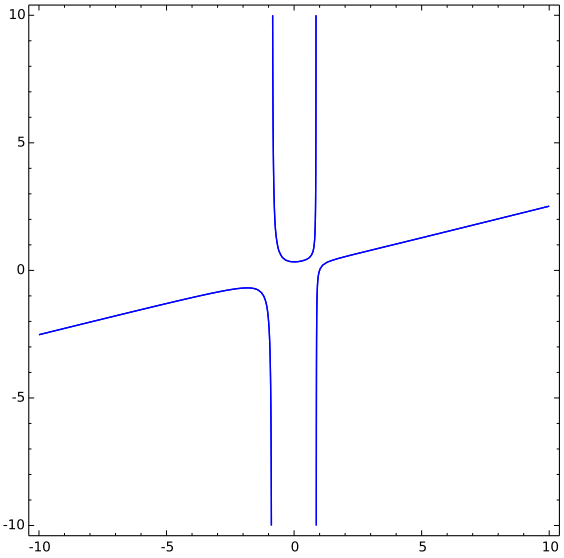


Figure 12: Implicit graph generated using SageTEX.

2.3.25 Exercises

1. Plot the graph of $y = \sin(\pi x - \pi)$ for $-1 \leq x \leq 1$ using a thick red line.
2. Plot the graph of $\cos(\pi x - \pi)$ on the same interval using a thick blue line.
3. Plot the two graphs above on the same set of axes.
4. Plot the graph of $y = 1/x$ for $-1 \leq x \leq 1$ adjusting the range so that only $-10 \leq y \leq 10$.
5. Plot $y \cdot \sin\left(x^2 - y^2\right) = x \cdot \cos(x + y)$, for $-3 \leq x \leq 3$ and $-3 \leq y \leq 3$.
6. Plot $x = \cos(3t)$ and $y = \cos(t + \cos(3t))$.

2.3.26 Answers to Exercises

```
sage: p=plot(sin(pi*x-pi),(x,-1,1),color='red',
   thickness=5)
```

```
sage: p.show()
None
sage: q=plot(cos(pi*x-pi),(x,-1,1),color='blue',
    thickness=5)
sage: q.show()
None
sage: r=p+q
sage: r.show()
None
sage: s=plot(1/x,xmin=-1,xmax=1,ymin=-10,ymax=10)
sage: s.show()
None
sage: ip = implicit_plot(y*sin(x^2-y^2)==x*cos(x+y),(
    x,-3,3),(y,-3,3))
sage: ip.show()
None
sage: t = var('t')
sage: u=parametric_plot([cos(3*t),cos(t+cos(3*t))],(t
    ,0,2*pi))
sage: u.show()
None
```

2.3.27 3D Plots

Producing 3D plots can be done using the `plot3d()` command.

```
sage: x, y = var('x', 'y')
sage: f(x, y) = x^2 - y^2
sage: p3d = plot3d(f(x, y), (x,-10,10), (y,-10,10))
sage: p3d.show()
None
```

Sage handles 3D plotting (Figure 13, page 36) a bit differently than what we have seen thus far. It uses a program named `jmol` to generate interactive plots. So instead of just a static picture we will see either a new window, or if you are using Sage's notebook interface, a java applet in your browser's window.

One nice thing about the way that Sage does this is that you can rotate your plot by just clicking on the surface and dragging it in the direction in which you would like for it to rotate. Zooming in/out can also be done by using your mouse's wheel button (or two-finger vertical swipe on a Mac). Once you have rotated and zoomed the plot to your liking, you can save the plot as a file. Do this by right-clicking/command anywhere in the window/applet and selecting export as a png file.

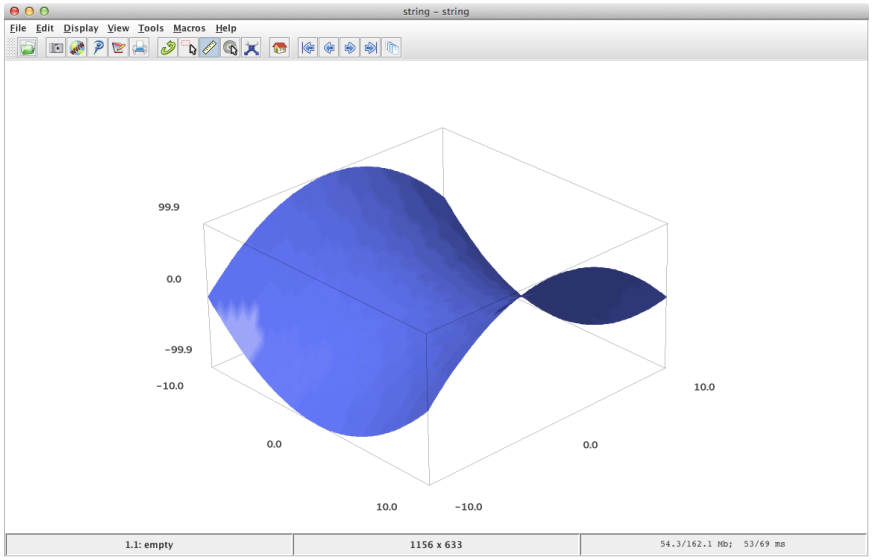


Figure 13: A screen Grab of an interactive `jmol` rendering of the Sage 3D code.

2.3.28 Differential Equations

You can use Sage to investigate ordinary differential equations. To solve the equation $x' + x - 1 = 0$.

```
sage: t = var('t')
sage: x = function('x',t)
sage: DE = diff(x,t) + x - 1
sage: desolve(DE,[x,t])
(c + e^t)*e^(-t)
```

This uses Sages interface to Maxima, and so its output may be a bit different from other Sage output. In this case, this says that the general solution to the differential equation is $x(t) = e^{-t}(e^t + c)$.

You can compute Laplace transforms also; the Laplace transform of

$$t^2 e^t - \sin(t)$$

is computed as follows:

```
sage: s = var('s')
sage: t = var('t')
sage: f = t^2*e^t-sin(t)
sage: f.laplace(t,s)
-1/(s^2 + 1) + 2/(s - 1)^3
```

3 Configuring TEXShop

The default install of TEXShop does not interact with Sage, so you'll need to do some preliminary steps before moving forward. Once you complete this section you will not have to do this again—it's a one shot deal and you should be able to move forward in using SageTEX in your TEXShop documents.

The Sage command line program is inside the Sage application wrapper, and the TEXShop engine folder is going to have to use that binary. To make things a lot simpler, you need to go into your Applications folder and rename the Sage application[8] to just Sage; furthermore, make sure your Sage application is actually in the Applications folder, that is in the path `/Applications/Sage.app`.

Now you'll need to set-up the Sage engine to work with TEXShop. Go to the folder[9]

```
~/Library/TeXShop/Engines/Inactive/Sage
```

copy the file `sage.engine` to

```
~/Library/TeXShop/Engines/
```

Run the Terminal application which can be found in `/Applications/Utilities` folder and type the following

```
cd /usr/local/texlive/texmf-local/tex/latex
```

at the Terminal prompt. Then type

```
sudo ln -s /Applications/Sage.app/Contents/Resources/sage
/data/texmf/tex/generic/sagetex/sagetex.sty sagetex.sty
```

as a single line command, yes it will wrap. Finally, type at the Terminal prompt:

```
sudo mktexlsr
```

4 TEXShop Preamble

Now if you wish to use SageTEX in you TEXShop documents, all source files will need to include the line

```
% !TEX TS-program = sage
```

as the very first line of your LATEX source file. And you will now need to modify your preamble to include this line:

```
\usepackage{sagetex}
```

Here's what my preamble typically includes:

[8] The default name of the Sage application includes versioning information.

[9] You can do this in the Sage by using the "Go" menu and the "Go to Folder ..." menu item.

```
% !TEX TS-program = sage
\documentclass[fleqn]{article}
\title{Aways Something}
\author{Possibly Sombody}
\usepackage{hyperref}
\usepackage{tkz-berge}
\usepackage{sagetex}
\setlength{\sagetexindent}{10ex}
```

5 Getting Started with SageTEX

Did you read through and follow the previous steps? If so, you will now be able to download two files (one `.tex` and one `.pdf`)[10] and create this exact electronic copy of what you are now reading. That's an important first step in learning how to embed Sage into your LaTeX documents. Once done you will continue to look at the LaTeX source to figure your way through what follows. Be aware that as soon as you press the `Typeset` button in TeXShop you'll see the `Console` window appear for *a lot longer* than expected, and a series of graphic images popping up. Be patient, all should typeset as expected. After this initial run you may want to comment out the tutorial section, as it is running an awful lot of commands to show graphics. Now you're going to need to concentrate on both the code and output to learn the basics of SageTEX.

6 Sage, Seen and Unseen

Many times we would like to include a computation in our written work, and this may involve a lot of unnecessary cutting/pasting/re-typing. Sage to the rescue! For example, let's say I want to approximate π to 10 digits? There's no need to exit TeXShop to do so: $\pi \approx 3.141592654$. Okay, that was easy, but what about solving an equation? For example, given $2\,x - 9 = 5$ its solution is $x = 7$. But what if the equation is more difficult? The solutions to $x^2 = 5$ are $x = -\sqrt{5}$ or $x = \sqrt{5}$. Even more difficult, solving $6\,x^3 - 25\,x^2 + 31\,x - 10 = 0$ for x yields these three solutions: $x = 2$ or $x = \frac{1}{2}$ or $x = \frac{5}{3}$.

Suppose now you want to show the Sage code that is being used, and then use the results in the body of the text.

```
x, y = var('x', 'y')
eqn04 = [x - y == 1, x + y  == 3]
s04 = solve(eqn04, x, y)
```

[10] You can download them at at:

`http://devio.us/~bannon/sage/`

.

The solution to the system $x - y = 1$ and $x + y = 3$ is easy to solve, but Sage can do it too.

$$x = 2 \quad y = 1$$

Here's a much more difficult example[11] that would not be so easy to do, let alone typeset.

```
a, b, c = var('a', 'b', 'c')
eqn05 = [a + b*c == 1, b - a*c == 0, a + b == 5]
s05 = solve(eqn05, a, b, c)
```

`eqn05` is a system of equations and is composed of theses three equations: $bc + a = 1$, $-ac + b = 0$, and $a + b = 5$. The declared variables are a, b and c. Here are the results:

$$a = \frac{25i\sqrt{79} + 25}{6i\sqrt{79} - 34}, \quad b = \frac{5i\sqrt{79} + 5}{i\sqrt{79} + 11}, \quad c = \frac{1}{10}i\sqrt{79} + \frac{1}{10},$$

or

$$a = \frac{25i\sqrt{79} - 25}{6i\sqrt{79} + 34}, \quad b = \frac{5i\sqrt{79} - 5}{i\sqrt{79} - 11}, \quad c = -\frac{1}{10}i\sqrt{79} + \frac{1}{10}.$$

I think you'll agree that this is pretty damn neat! Not only is the system being solved, but I can typeset both the input and output.

Now, let's move on to a variety of examples by making some simple statements.

```
f(x) = sin(x)
g(x) = cos(x)
h(x) = tan(x)
```

First, let's just do some simple things with these functions.

$$\begin{aligned}
f(x) &= \sin(x) \\
g(x) &= \cos(x) \\
h(x) &= \tan(x) \\
f(\pi) &= 0 \\
g(\pi) &= -1 \\
h(\pi) &= 0 \\
f(1) &\approx 0.841470984807897 \\
g(2) &\approx -0.416146836547142 \\
h(3) &\approx -0.142546543074278
\end{aligned}$$

[11] This is similar to an example done by Dan Drake, the designer of SageTEX.

Now, some more complicated things with these functions.

$$\begin{aligned}
f'(x) &= \cos(x) \\
g'(x) &= -\sin(x) \\
h'(x) &= \tan(x)^2 + 1 \\
\int f(x)\,\mathrm{d}x &= -\cos(x) + C \\
\int g(x)\,\mathrm{d}x &= \sin(x) + C \\
\int h(x)\,\mathrm{d}x &= \log(\sec(x)) + C \\
\int_0^{\pi/2} f(x)\,\mathrm{d}x &= 1 \\
\int_{-\pi/4}^{\pi/4} g(x)\,\mathrm{d}x &= \sqrt{2} \\
\int_0^{\pi/3} h(x)\,\mathrm{d}x &= \log(2)
\end{aligned}$$

You can even build on these functions.

$$\begin{aligned}
\lim_{x\to 0^+} \frac{f(x)}{x} &= 1 \\
\lim_{x\to 0^-} \frac{f(x)}{x} &= 1 \\
\lim_{x\to 0} \frac{f(x)}{x} &= 1
\end{aligned}$$

Or you may just want to bypass defining your functions.

$$\begin{aligned}
\lim_{x\to 0^+} \frac{|x|}{x} &= 1 \\
\lim_{x\to 0^-} \frac{|x|}{x} &= -1 \\
\lim_{x\to 0} \frac{|x|}{x} &= \mathit{und}
\end{aligned}$$

Thankfully, we can even generate series. For example, the Maclaurin Series of $f(x)$ begins:

$$x \mapsto -\frac{1}{39916800}\,x^{11} + \frac{1}{362880}\,x^9 - \frac{1}{5040}\,x^7 + \frac{1}{120}\,x^5 - \frac{1}{6}\,x^3 + x.$$

And, the Taylor Series centered at $x = 1$ of $f(x)$ begins:

$$x \mapsto -\frac{1}{2}\,(x-1)^2 \sin(1) + (x-1)\cos(1) + \sin(1).$$

Again, the key points here, are the difference between `sagesilent` and `sageblock` is one of appearance. Although you do not see the contents of `sagesilent`, you can still display it in your document: $a = -3$. But at times you want your reader to see the Sage code and the output:

```
t = var('t')
f(t) = log(cos(t)/t^2 + 1)
fp(t) = diff(f(t),t)
```

$$\begin{aligned} f(t) &= \log\left(\frac{\cos(t)}{t^2} + 1\right) \\ f'(t) &= -\frac{\frac{\sin(t)}{t^2} + \frac{2\cos(t)}{t^3}}{\frac{\cos(t)}{t^2} + 1}. \end{aligned}$$

Here's a simple plot (Figure 14, page 41) of $f(x)$. And here is a simple plot

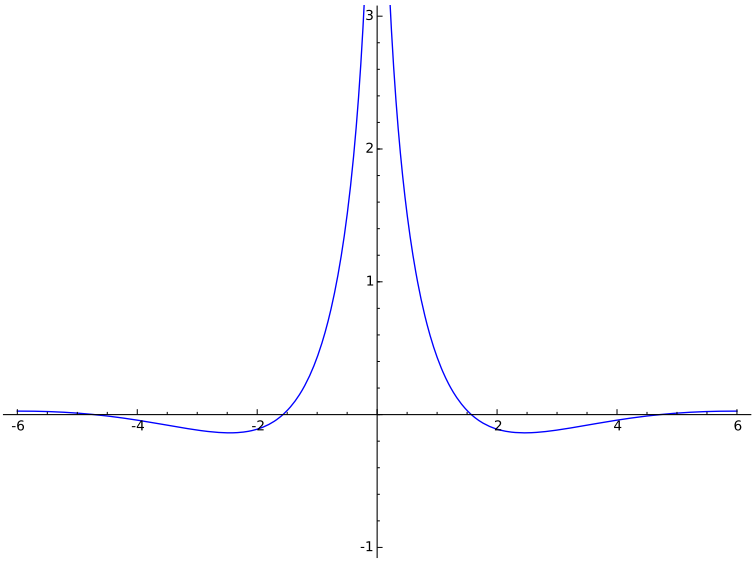


Figure 14: A simple plot of $f(x)$.

(Figure 15, page 42) of $f(x)$ and $f'(x)$ on the same set of axes.

Python indentation works as expected. Here's a example [12] that produces a very nice picture.

```
s = 7
s2 = 2^s
P.<x> = GF(2)[]
```

[12] An example done by Dan Drake, the designer of SageTEX.

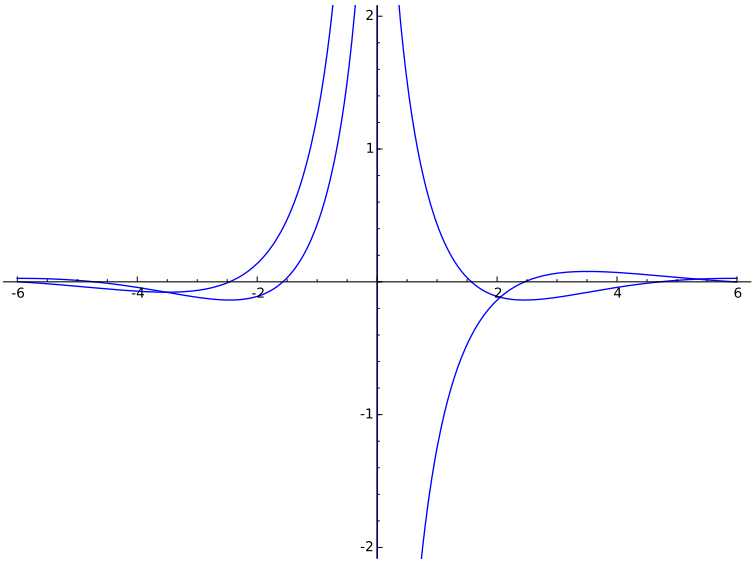


Figure 15: A simple plot of $f(x)$ and $f'(x)$.

```
M = matrix(parent(x),s2)
for i in range(s2):
    p  = (1+x)^i
    pc = p.coeffs()
    a  = pc.count(1)
    for j in range(a):
        idx = pc.index(1)
        M[i,idx+j] = pc.pop(idx)
matrixprogram = matrix_plot(M,cmap='Greys')
```

And here's the picture (Figure 16, page 43) that the code above produces.

---

7 Plotting graphs with TikZ

This material in this subsection is taken nearly verbatim from Drake's excellent guide on SageTEX.

Sage now includes some nice support for plotting graphs using TikZ. Here, we mean things with vertices and edges, not graphs of a function of one or two variables.

The graphics in this section depends on the `tkz-berge` package, which is generally only available in newer TEX distributions (for example, TEXLive 2011 and newer). That package depends in turn on TikZ 2.0, which is also only available in newer TEX distributions.

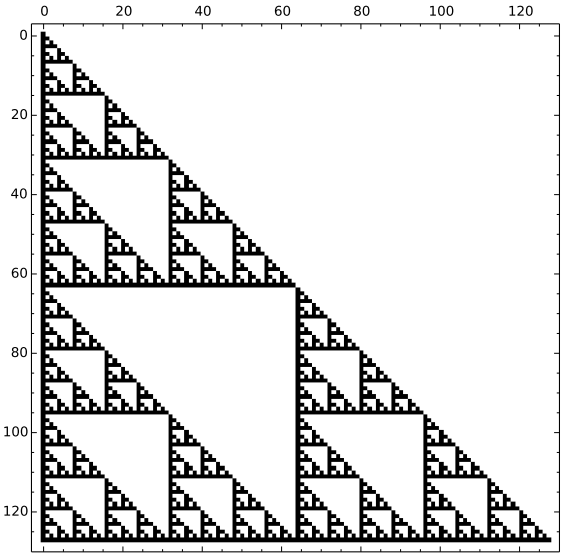
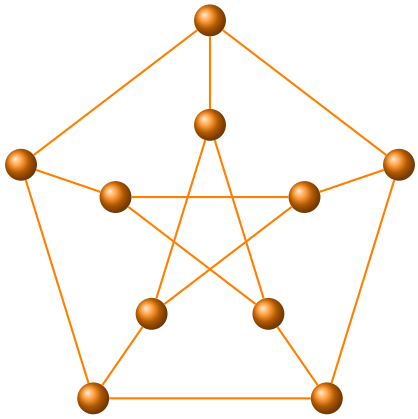


Figure 16: The matrix program plot.

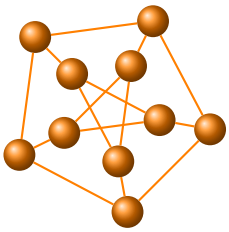
First define our graph:

```
g = graphs.PetersenGraph()
g.set_latex_options(tkz_style='Art')
```

Now just do `\sage{}` on it to plot it. You'll need to use the `tkz-berge` package for this to work; that package in turn depends on `tkz-graph` and TikZ. See `altermundus.fr/pages/tkz.html`; if you're using a recent version of TEXLive, you can use its package manager to install those packages, or get them from CTAN: `www.ctan.org/pkg/tkz-berge`. See "LATEX Options for Graphs" in the Sage reference manual for more details.

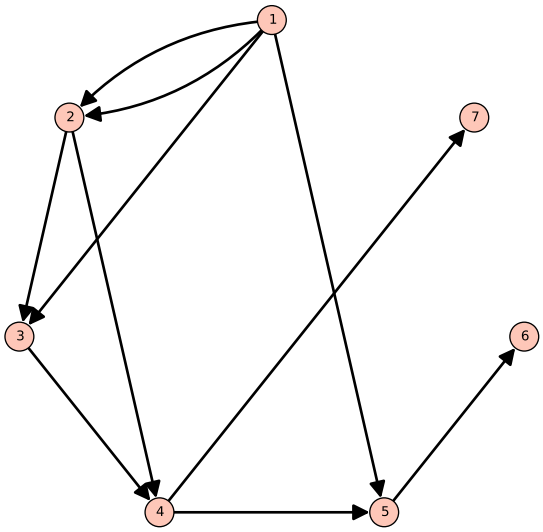


The above command just outputs a `tikzpicture` environment, and you can control that environment using anything supported by TikZ—although the output of `\sage{g}` explicitly hard-codes a lot of things and cannot be flexibly controlled in its current form.



Here's some more graphs, plotted using the usual plot routines.

```
G4 = DiGraph({1:[2,2,3,5], 2:[3,4], 3:[4], 4:[5,7], 5:[6]},\
             multiedges=True)
G4plot = G4.plot(layout='circular')
```



---

8 Pausing SageTEX

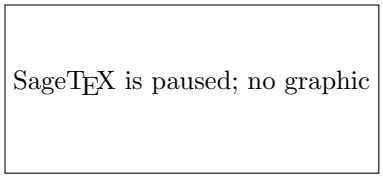
If you're in the edit/compile mode of reviewing this document and you did not comment out a lot of the material, you're probably a bit annoyed at the time it takes to compile. Yes, sometimes you just want to "stop" Sage from

doing its work. You can get Sage to stop by using the `\sagetexpause` and `\sagetexunpause` macros. Go ahead and try to comment out and back in these macros to see what happens. In fact I think you'll be using these macros quite a bit if you realistically use SageTEX. However, I am partial to using `comments` instead.

A calculation: (SageTEX is paused) and a code environment that simulates a time-consuming calculation. While paused, this will get skipped over.

```
import time
time.sleep(10)
```

Graphics are also skipped:



---

9 Make Sage write your LaTeX for you

This material in this subsection is taken nearly verbatim from Drake's excellent guide on SageTEX.

With SageTEX, you can not only have Sage do your math for you, it can write parts of your LaTeX document for you! For example, I hate writing `tabular` environments; there's too many fiddly little bits of punctuation and whatnot... and what if you want to add a column? It's a pain—or rather, it *was* a pain. Just write a Sage/Python function that outputs a string of LaTeX code, and use `\sagestr`. Here's how to make Pascal's triangle.

```
def pascals_triangle(n):
    # start of the table
    s  = [r"\begin{tabular}{cc|" + "r" * (n+1) + "}"]
    s.append(r"  & & $k$: & \\")
    # second row, with k values:
    s.append(r"  & ")
    for k in [0..n]:
        s.append("& {0} ".format(k))
    s.append(r"\\")
    # the n = 0 row:
    s.append(r"\hline" + "\n" + r"$n$: & 0 & 1 & \\")
    # now the rest of the rows
    for r in [1..n]:
        s.append(" & {0} ".format(r))
        for k in [0..r]:
            s.append("& {0} ".format(binomial(r, k)))
        s.append(r"\\")
```

```
        # add the last line and return
        s.append(r"\end{tabular}")
        return ''.join(s)

    # how big should the table be?
    n = 8
```

Okay, now here's the table. To change the size, edit `n` above. If you have several tables, you can use this to get them all the same size, while changing only one thing.

		k:								
		0	1	2	3	4	5	6	7	8
n:	0	1								
	1	1	1							
	2	1	2	1						
	3	1	3	3	1					
	4	1	4	6	4	1				
	5	1	5	10	10	5	1			
	6	1	6	15	20	15	6	1		
	7	1	7	21	35	35	21	7	1	
	8	1	8	28	56	70	56	28	8	1

---

10 The `sagecommandline` environment

As you have probably noticed, at least if you read through the tutorial section of this document, that I have extensively embedded Sage commands into my LaTeX document. The Sage commands were run and then the output is magically displayed in my document. Here's an example.

```
sage: 3^5 - 5^3                                                         628
118                                                                     629
sage: diff(sin(x^2),x)                                                  630
2*x*cos(x^2)                                                            631
sage: factor(8+27*x^3)                                                  632
(9*x^2 - 6*x + 4)*(3*x + 2)                                             633
sage: latex(factor(8+27*x^3))                                           634
{\left(9 \, x^{2} - 6 \, x + 4\right)} {\left(3 \, x                    635
    + 2\right)}
```

You should note that the Sage commands in the LaTeX source do not include the Sage output, but the typeset output shows the Sage output. There are some subtle differences between Sage in the Terminal and in a LaTeX document, and that has to do with the way Python treats statements and expressions. I suggest you run your commands in the Terminal to see if you're getting what you expect

to get in your LaTeX documents. It's not a perfect world, so please do check. This is a part of the infinite *learning curve*, and it is essential that you take the time to verify that your Sage code in your LaTeX documents is what you expect.

If you need to set a label in your Sage you can use the @ sign to do so.

```
sage: A = matrix([[1,2,3],[4,5,6],[7,8,9]])
sage: A
[1 2 3]
[4 5 6]
[7 8 9]
sage: B = diagonal_matrix([1, 1, 1])
sage: B
[1 0 0]
[0 1 0]
[0 0 1]
sage: C = matrix([[9,8,7],[6,5,4],[3,2,1]])
sage: C
[9 8 7]
[6 5 4]
[3 2 1]
sage: A*B*C
[ 30  24  18]
[ 84  69  54]
[138 114  90]
```

You can now refer to the line 641, which is on page 47. Oh, don't forget you can even get the typeset output.

$$\begin{aligned}\mathbf{ABC} &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{pmatrix}\end{aligned}$$

---

11 Randomized Test Generation

If you're ever planning on creating a *randomized* version of a mathematics test you may want to look over the code in this section. Although random here does not mean that each version will be equal in its level of difficulty, but it is a good way to randomize a test bank of questions. For example, we can now generate a random mathematical expression.

$$A = 1$$

$$
\begin{aligned}
B &= 4 \\
C &= 4 \\
D &= 4 \\
E &= 6 \\
F &= 6 \\
G &= 7 \\
H &= 8 \\
f(x) &= 4x^2 - 28x + 24 = 4(x-1)(x-6) \\
g(x) &= 32x^2 + 20x - 42 = 2(8x-7)(2x+3) \\
h(x) &= 4x^2 + 31x + 42 = (4x+7)(x+6) \\
j(x) &= \frac{4x^2 + 31x + 42}{2(16x^2 + 10x - 21)} \\
k(x) &= \frac{4\left(x^2 - 7x + 6\right)}{4x^2 + 31x + 42} \\
m(x) &= 6\sin(4x+4) + 4 \\
n(x) &= 6\cos(4x+4) + 4 \\
p(x) &= 7\tan(4x+1) + 8
\end{aligned}
$$

And then do some mathematics on them. Like expanding.

$$f(x) \cdot g(x) = 128x^4 - 816x^3 + 40x^2 + 1656x - 1008$$

And integrating.

$$\int_1^2 g(x)\,\mathrm{d}x = \frac{188}{3}$$

And differentiating.

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}\left(4x^2 + 31x + 42\right) &= 8x + 31 \\
\frac{\mathrm{d}}{\mathrm{d}x}\left[7\tan(4x+1) + 8\right] &= 28\tan(4x+1)^2 + 28
\end{aligned}
$$

Or factoring a big integer.

$$628420859775346172805680578 56 = 2^{13} \cdot 3^2 \cdot 7 \cdot 113 \cdot 1194679 \cdot 901966538535443$$

Or set-up some random equation to solve.

$$4x^2 + 31x + 42 = \frac{4x^2 + 31x + 42}{2(16x^2 + 10x - 21)}$$

The solution set is:

$$x_1 = -\frac{3}{16}\sqrt{41} - \frac{5}{16}$$

$$\begin{aligned} x_2 &= \frac{3}{16}\sqrt{41} - \frac{5}{16} \\ x_3 &= -6 \end{aligned}$$

Or even create graphs! But be careful, because you need to be aware that the functions are a bit random and the domain/range will change. Making this look good is a challenge.

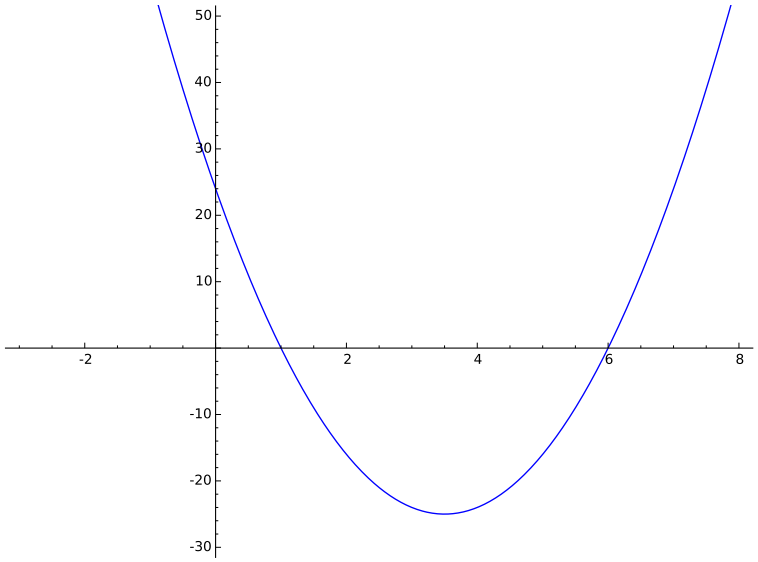


Figure 17: Partial graph of $f(x)$ generated using SageTEX.

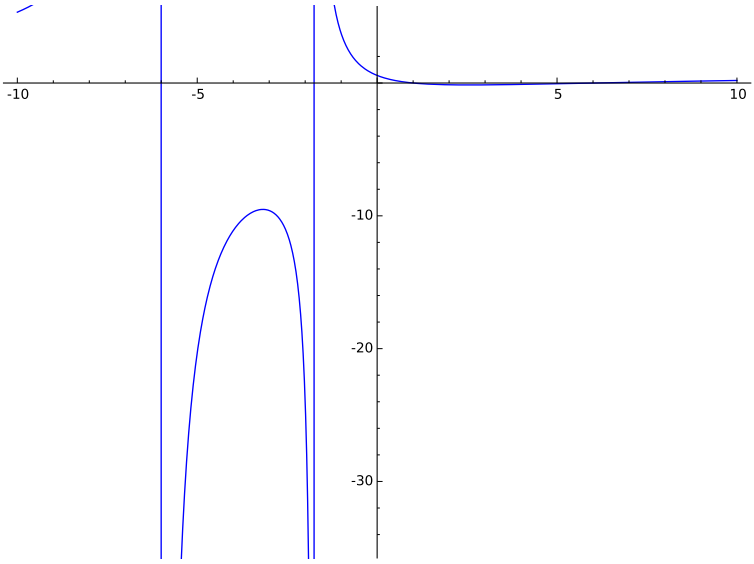


Figure 18: Partial graph of $k(x)$ generated using SageTEX.

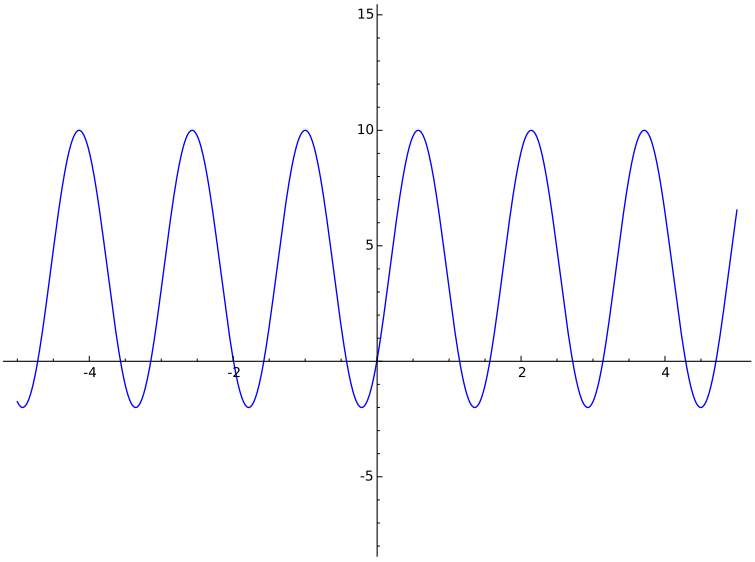


Figure 19: Partial graph of $n(x)$ generated using SageTEX.