
Sage Constructions

Release 4.8

The Sage Development Team

January 20, 2012

CONTENTS

1	Calculus	3
1.1	Differentiation	3
1.2	Integration	4
1.3	Ordinary differential equations	6
1.4	Fourier series of periodic functions	7
2	Plotting	9
2.1	Plotting functions in 2D	9
2.2	Plotting curves	10
2.3	openmath	11
2.4	Tachyon 3D plotting	11
2.5	gnuplot	12
2.6	Plotting surfaces	12
3	Groups	15
3.1	Permutation groups	15
3.2	Conjugacy classes	16
3.3	Normal subgroups	17
3.4	Centers	17
3.5	The group id database	18
4	Linear algebra	19
4.1	Vector spaces	19
4.2	Matrix powers	19
4.3	Kernels	19
4.4	Eigenvectors and eigenvalues	21
4.5	Row reduction	23
4.6	Characteristic polynomial	23
4.7	Solving systems of linear equations	24
5	Linear codes and ciphers	27
5.1	Codes	27
5.2	Ciphers	29
6	Graph theory	31
6.1	Networkx	31
6.2	Cayley graphs	31
6.3	Graphs from adjacency matrices	32
7	Representation theory	33

7.1	Ordinary characters	33
7.2	Brauer characters	35
8	Rings	37
8.1	Matrix rings	37
8.2	Polynomial rings	37
8.3	p -adic numbers	38
8.4	Quotient rings of polynomials	38
9	Polynomials	41
9.1	Polynomial powers	41
9.2	Factorization	42
9.3	Polynomial GCD's	42
9.4	Roots of polynomials	43
9.5	Evaluation of multivariate functions	43
9.6	Roots of multivariate polynomials	44
9.7	Gröbner bases	44
10	Elementary number theory	47
10.1	Taking modular powers	47
10.2	Discrete logs	47
10.3	Prime numbers	47
10.4	Divisors	48
10.5	Quadratic residues	48
11	Modular forms	49
11.1	Cusp forms	49
11.2	Coset representatives	49
11.3	Modular symbols and Hecke operators	50
11.4	Genus formulas	51
12	Elliptic curves	53
12.1	Conductor	53
12.2	j -invariant	53
12.3	The $GF(q)$ -rational points on E	54
12.4	Modular form associated to an elliptic curve over \mathbb{Q}	54
13	Number fields	55
13.1	Ramification	55
13.2	Class numbers	56
13.3	Integral basis	57
14	Algebraic Geometry	59
14.1	Point counting on curves	59
14.2	Riemann-Roch spaces using Singular	63
15	Interface Issues	67
15.1	Background jobs	67
15.2	Referencing Sage	67
15.3	Logging your Sage session	69
15.4	LaTeX conversion	69
15.5	Sage and other computer algebra systems	69
15.6	Command-line Sage help	69
15.7	Reading and importing files into Sage	70
15.8	Installation for the impatient	71

15.9 Python language program code for Sage commands	72
15.10 “Special functions” in Sage	72
15.11 What is Sage?	73
16 Contributions to this document	75
17 Indices and tables	77
Index	79

This document collects the answers to some questions along the line “How do I construct ... in Sage?” Though much of this material can be found in the manual or tutorial or string documentation of the Python code, it is hoped that this will provide the casual user with some basic examples on how to start using Sage in a useful way.

This work is licensed under a [Creative Commons Attribution-Share Alike 3.0 License](#).

Please send suggestions, additions, corrections to the `sage-devel` Google group!

The Sage wiki <http://wiki.sagemath.org/> contains a wealth of information. Moreover, all these can be tested in the Sage notebook interface <http://www.sagenb.org/> on the web.

CALCULUS

Here are some examples of calculus symbolic computations using Sage. They use the Maxima interface.

Work is being done to make the commands for the symbolic calculations given below more intuitive and natural. At the moment, we use the maxima class interface.

1.1 Differentiation

Differentiation:

```
sage: var('x k w')
(x, k, w)
sage: f = x^3 * e^(k*x) * sin(w*x); f
x^3*e^(k*x)*sin(w*x)
sage: f.diff(x)
k*x^3*e^(k*x)*sin(w*x) + w*x^3*e^(k*x)*cos(w*x) + 3*x^2*e^(k*x)*sin(w*x)
sage: latex(f.diff(x))
k x^{3} e^{\left(k x\right)} \sin\left(w x\right) + w x^{3} e^{\left(k x\right)} \cos\left(w x\right)
```

If you type `view(f.diff(x))` another window will open up displaying the compiled output. In the notebook, you can enter

```
var('x k w')
f = x^3 * e^(k*x) * sin(w*x)
show(f)
show(f.diff(x))
```

into a cell and press `shift-enter` for a similar result. You can also differentiate and integrate using the commands

```
R = PolynomialRing(QQ, "x")
x = R.gen()
p = x^2 + 1
show(p.derivative())
show(p.integral())
```

in a notebook cell, or

```
sage: R = PolynomialRing(QQ, "x")
sage: x = R.gen()
sage: p = x^2 + 1
sage: p.derivative()
2*x
sage: p.integral()
1/3*x^3 + x
```

on the command line. At this point you can also type `view(p.derivative())` or `view(p.integral())` to open a new window with output typeset by LaTeX.

1.1.1 Critical points

You can find critical points of a piecewise defined function:

```
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f1 = x^0
sage: f2 = 1-x
sage: f3 = 2*x
sage: f4 = 10*x-x^2
sage: f = Piecewise([[ (0,1), f1], [(1,2), f2], [(2,3), f3], [(3,10), f4]])
sage: f.critical_points()
[5.0]
```

1.1.2 Power series

Taylor series:

```
sage: var('f0 k x')
(f0, k, x)
sage: g = f0/sinh(k*x)^4
sage: g.taylor(x, 0, 3)
-62/945*f0*k^2*x^2 + 11/45*f0 - 2/3*f0/(k^2*x^2) + f0/(k^4*x^4)
sage: maxima(g).powerseries('x', 0)
16*f0*(sum((2^(2*i1-1)-1)*bern(2*i1)*k^(2*i1-1)*x^(2*i1-1)/(2*i1)!, i1, 0, inf))^4
```

Of course, you can view the LaTeX-ed version of this using `view(g.powerseries('x', 0))`.

The Maclaurin and power series of $\log\left(\frac{\sin(x)}{x}\right)$:

```
sage: f = log(sin(x)/x)
sage: f.taylor(x, 0, 10)
-1/467775*x^10 - 1/37800*x^8 - 1/2835*x^6 - 1/180*x^4 - 1/6*x^2
sage: [bernoulli(2*i) for i in range(1,7)]
[1/6, -1/30, 1/42, -1/30, 5/66, -691/2730]
sage: maxima(f).powerseries(x, 0)
'sum((-1)^i2*2^(2*i2-1)*bern(2*i2)*x^(2*i2)/(i2*(2*i2)!), i2, 1, inf)
```

1.2 Integration

Numerical integration is discussed in *Riemann and trapezoid sums for integrals* below.

Sage can integrate some simple functions on its own:

```
sage: f = x^3
sage: f.integral(x)
1/4*x^4
sage: integral(x^3, x)
1/4*x^4
sage: f = x*sin(x^2)
sage: integral(f, x)
-1/2*cos(x^2)
```

Sage can also compute symbolic definite integrals involving limits.

```
sage: var('x, k, w')
(x, k, w)
sage: f = x^3 * e^(k*x) * sin(w*x)
sage: f.integrate(x)
-((k^6*w + 3*k^4*w^3 + 3*k^2*w^5 + w^7)*x^3 - 24*k^3*w + 24*k*w^3 - 6*(k^5*w + 2*k^3*w^3 + k*w^5)*x
sage: integrate(1/x^2, x, 1, infinity)
1
```

1.2.1 Convolution

You can find the convolution of any piecewise defined function with another (off the domain of definition, they are assumed to be zero). Here is f , $f * f$, and $f * f * f$, where $f(x) = 1$, $0 < x < 1$:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: f = Piecewise([[ (0,1), 1*x^0]])
sage: g = f.convolution(f)
sage: h = f.convolution(g)
sage: P = f.plot(); Q = g.plot(rgbcolor=(1,1,0)); R = h.plot(rgbcolor=(0,1,1))
```

To view this, type `show(P+Q+R)`.

1.2.2 Riemann and trapezoid sums for integrals

Regarding numerical approximation of $\int_a^b f(x) dx$, where f is a piecewise defined function, can

- compute (for plotting purposes) the piecewise linear function defined by the trapezoid rule for numerical integration based on a subdivision into N subintervals
- the approximation given by the trapezoid rule,
- compute (for plotting purposes) the piecewise constant function defined by the Riemann sums (left-hand, right-hand, or midpoint) in numerical integration based on a subdivision into N subintervals,
- the approximation given by the Riemann sum approximation.

```
sage: f1(x) = x^2
sage: f2(x) = 5-x^2
sage: f = Piecewise([[ (0,1), f1], [(1,2), f2]])
sage: f.trapezoid(4)
Piecewise defined function with 4 parts, [[(0, 1/2), 1/2*x],
[(1/2, 1), 9/2*x - 2], [(1, 3/2), 1/2*x + 2],
[(3/2, 2), -7/2*x + 8]]
sage: f.riemann_sum_integral_approximation(6,mode="right")
19/6
sage: f.integral()
Piecewise defined function with 2 parts,
[[ (0, 1), x |--> 1/3*x^3], [(1, 2), x |--> -1/3*x^3 + 5*x - 13/3]]
sage: f.integral(definite=True)
3
```

1.2.3 Laplace transforms

If you have a piecewise-defined polynomial function then there is a “native” command for computing Laplace transforms. This calls Maxima but it’s worth noting that Maxima cannot handle (using the direct interface illustrated in the

last few examples) this type of computation.

```
sage: var('x s')
(x, s)
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f = Piecewise([[ (0,1), f1], [(1,2), f2]])
sage: f.laplace(x, s)
(s + 1)*e^(-2*s)/s^2 - e^(-s)/s + 1/s - e^(-s)/s^2
```

For other “reasonable” functions, Laplace transforms can be computed using the Maxima interface:

```
sage: var('k, s, t')
(k, s, t)
sage: f = 1/exp(k*t)
sage: f.laplace(t, s)
1/(k + s)
```

is one way to compute LT’s and

```
sage: var('s, t')
(s, t)
sage: f = t^5*exp(t)*sin(t)
sage: L = laplace(f, t, s); L
3840*(s - 1)^5/(s^2 - 2*s + 2)^6 - 3840*(s - 1)^3/(s^2 - 2*s + 2)^5 +
720*(s - 1)/(s^2 - 2*s + 2)^4
```

is another way.

1.3 Ordinary differential equations

Symbolically solving ODEs can be done using Sage interface with Maxima. See

```
sage:desolvers?
```

for available commands. Numerical solution of ODEs can be done using Sage interface with Octave (an experimental package), or routines in the GSL (Gnu Scientific Library).

An example, how to solve ODE’s symbolically in Sage using the Maxima interface (do not type the . . .):

```
sage: y=function('y',x); desolve(diff(y,x,2) + 3*x == y, dvar = y, ics = [1,1,1])
3*x - 2*e^(x - 1)
sage: desolve(diff(y,x,2) + 3*x == y, dvar = y)
k1*e^x + k2*e^(-x) + 3*x
sage: desolve(diff(y,x) + 3*x == y, dvar = y)
(3*(x + 1)*e^(-x) + c)*e^x
sage: desolve(diff(y,x) + 3*x == y, dvar = y, ics = [1,1]).expand()
3*x - 5*e^(x - 1) + 3

sage: f=function('f',x); desolve_laplace(diff(f,x,2) == 2*diff(f,x)-f, dvar = f, ics = [0,1,2])
x*e^x + e^x

sage: desolve_laplace(diff(f,x,2) == 2*diff(f,x)-f, dvar = f)
-x*e^x*f(0) + x*e^x*D[0](f)(0) + e^x*f(0)
```

If you have Octave and gnuplot installed,

```
sage: octave.de_system_plot(['x+y', 'x-y'], [1,-1], [0,2]) # optional octave required
```

yields the two plots $(t, x(t)), (t, y(t))$ on the same graph (the t -axis is the horizontal axis) of the system of ODEs

$$x' = x + y, x(0) = 1; y' = x - y, y(0) = -1,$$

for $0 \leq t \leq 2$. The same result can be obtained by using `desolve_system_rk4`:

```
sage: x, y, t = var('x y t')
sage: P=desolve_system_rk4([x+y, x-y], [x,y], ics=[0,1,-1], ivar=t, end_points=2)
sage: p1 = list_plot([[i,j] for i,j,k in P], plotjoined=True)
sage: p2 = list_plot([[i,k] for i,j,k in P], plotjoined=True, color='red')
sage: p1+p2
```

Another way this system can be solved is to use the command `desolve_system`.

```
sage: t=var('t'); x=function('x',t); y=function('y',t)
sage: des = [diff(x,t) == x+y, diff(y,t) == x-y]
sage: desolve_system(des, [x,y], ics = [0, 1, -1])
[x(t) == cosh(sqrt(2)*t), y(t) == sqrt(2)*sinh(sqrt(2)*t) - cosh(sqrt(2)*t)]
```

The output of this command is *not* a pair of functions.

Finally, can solve linear DEs using power series:

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=10)
sage: a = 2 - 3*t + 4*t^2 + O(t^10)
sage: b = 3 - 4*t^2 + O(t^7)
sage: f = a.solve_linear_de(prec=5, b=b, f0=3/5)
sage: f
3/5 + 21/5*t + 33/10*t^2 - 38/15*t^3 + 11/24*t^4 + O(t^5)
sage: f.derivative() - a*f - b
O(t^4)
```

1.4 Fourier series of periodic functions

If $f(x)$ is a piecewise-defined polynomial function on $-L < x < L$ then the Fourier series

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{n\pi x}{L}\right) + b_n \sin\left(\frac{n\pi x}{L}\right) \right]$$

converges. In addition to computing the coefficients a_n, b_n , it will also compute the partial sums (as a string), plot the partial sums (as a function of x over $(-L, L)$, for comparison with the plot of $f(x)$ itself), compute the value of the FS at a point, and similar computations for the cosine series (if $f(x)$ is even) and the sine series (if $f(x)$ is odd). Also, it will plot the partial F.S. Cesaro mean sums (a “smoother” partial sum illustrating how the Gibbs phenomenon is mollified).

```
sage: f1 = lambda x: -1
sage: f2 = lambda x: 2
sage: f = Piecewise([[0,pi/2), f1], [(pi/2,pi), f2]])
sage: f.fourier_series_cosine_coefficient(5,pi)
-3/5/pi
sage: f.fourier_series_sine_coefficient(2,pi)
-3/pi
```

```
sage: f.fourier_series_partial_sum(3,pi)
-3*sin(2*x)/pi + sin(x)/pi - 3*cos(x)/pi + 1/4
```

Type `show(f.plot_fourier_series_partial_sum(15,pi,-5,5))` and `show(f.plot_fourier_series_partial_sum_cesaro(15,pi,-5,5))` (and be patient) to view the partial sums.

PLOTTING

Sage can plot using `matplotlib`, `openmath`, `gnuplot`, or `surf` but only `matplotlib` and `openmath` are included with Sage in the standard distribution. For `surf` examples, see *Plotting surfaces*.

Plotting in Sage can be done in many different ways. You can plot a function (in 2 or 3 dimensions) or a set of points (in 2-D only) via `gnuplot`, you can plot a solution to a differential equation via Maxima (which in turn calls `gnuplot` or `openmath`), or you can use Singular's interface with the plotting package `surf` (which does not come with Sage). `gnuplot` does not have an implicit plotting command, so if you want to plot a curve or surface using an implicit plot, it is best to use the Singular's interface to `surf`, as described in chapter `ch:AG`, Algebraic geometry.

2.1 Plotting functions in 2D

The default plotting method in uses the excellent `matplotlib` package.

To view any of these, type `P.save("<path>/myplot.png")` and then open it in a graphics viewer such as `gimp`.

You can plot piecewise-defined functions:

```
sage: f1 = lambda x:1
sage: f2 = lambda x:1-x
sage: f3 = lambda x:exp(x)
sage: f4 = lambda x:sin(2*x)
sage: f = Piecewise([[ (0,1), f1], [(1,2), f2], [(2,3), f3], [(3,10), f4]])
sage: f.plot()
```

Other function plots can be produced as well:

A red plot of the Jacobi elliptic function $\operatorname{sn}(x, 2)$, $-3 < x < 3$ (do not type the . . . :

```
sage: L = [(i/100.0, maxima.eval('jacobi_sn (%s/100.0,2.0)')%i)]
... for i in range(-300,300)]
sage: show(line(L, rgbcolor=(3/4,1/4,1/8)))
```

A red plot of J -Bessel function $J_2(x)$, $0 < x < 10$:

```
sage: L = [(i/10.0, maxima.eval('bessel_j (2,%s/10.0)')%i) for i in range(100)]
sage: show(line(L, rgbcolor=(3/4,1/4,5/8)))
```

A purple plot of the Riemann zeta function $\zeta(1/2 + it)$, $0 < t < 30$:

```
sage: I = CDF.0
sage: show(line([zeta(1/2 + k*I/6) for k in range(180)], rgbcolor=(3/4,1/2,5/8)))
```

2.2 Plotting curves

To plot a curve in Sage, you can use `Singular` and `surf` (<http://surf.sourceforge.net/>, also available as an experimental package) or use `matplotlib` (included with Sage).

2.2.1 matplotlib

Here are several examples. To view them, type `p.save("<path>/my_plot.png")` (where `<path>` is a directory path which you have write permissions to where you want to save the plot) and view it in a viewer (such as GIMP).

A blue conchoid of Nicomedes:

```
sage: L = [[1+5*cos(pi/2+pi*i/100), tan(pi/2+pi*i/100)*\
... (1+5*cos(pi/2+pi*i/100))] for i in range(1,100)]
sage: line(L, rgbcolor=(1/4,1/8,3/4))
```

A blue hypotrochoid (3 leaves):

```
sage: n = 4; h = 3; b = 2
sage: L = [[n*cos(pi*i/100)+h*cos((n/b)*pi*i/100), \
... n*sin(pi*i/100)-h*sin((n/b)*pi*i/100)] for i in range(200)]
sage: line(L, rgbcolor=(1/4,1/4,3/4))
```

A blue hypotrochoid (4 leaves):

```
sage: n = 6; h = 5; b = 2
sage: L = [[n*cos(pi*i/100)+h*cos((n/b)*pi*i/100), \
... n*sin(pi*i/100)-h*sin((n/b)*pi*i/100)] for i in range(200)]
sage: line(L, rgbcolor=(1/4,1/4,3/4))
```

A red limaçon of Pascal:

```
sage: L = [[sin(pi*i/100)+sin(pi*i/50), -(1+cos(pi*i/100)+cos(pi*i/50))]\
... for i in range(-100,101)]
sage: line(L, rgbcolor=(1,1/4,1/2))
```

A light green trisectrix of Maclaurin:

```
sage: L = [[2*(1-4*cos(-pi/2+pi*i/100)^2), 10*tan(-pi/2+pi*i/100)*\
... (1-4*cos(-pi/2+pi*i/100)^2)] for i in range(1,100)]
sage: line(L, rgbcolor=(1/4,1,1/8))
```

A green lemniscate of Bernoulli (we omit `i==100` since that would give a 0 division error):

```
sage: v = [(1/cos(-pi/2+pi*i/100), tan(-pi/2+pi*i/100)) for i in range(1,200) if i!=100 ]
sage: L = [(a/(a^2+b^2), b/(a^2+b^2)) for a,b in v]
sage: line(L, rgbcolor=(1/4,3/4,1/8))
```

2.2.2 surf

In particular, since `surf` is only available on a UNIX type OS (and is not included with Sage), plotting using the commands below in Sage is only available on such an OS. Incidentally, `surf` is included with several popular Linux distributions.

```

sage: s = singular.eval
sage: s('LIB "surf.lib";')
...
sage: s("ring rr0 = 0, (x1,x2), dp;")
''
sage: s("ideal I = x1^3 - x2^2;")
''
sage: s("plot(I);")
...

```

Press `q` with the surf window active to exit from surf and return to Sage.

You can save this plot as a surf script. In the surf window which pops up, just choose `file, save as`, etc.. (Type `q` or select `file, quit`, to close the window.)

The plot produced is omitted but the gentle reader is encouraged to try it out.

2.3 openmath

Openmath is a TCL/Tk GUI plotting program written by W. Schelter.

The following command plots the function $\cos(2x) + 2e^{-x}$

```

sage: maxima.plot2d('cos(2*x) + 2*exp(-x)', '[x,0,1]', \
...   '[plot_format,openmath]') # optional -- pops up a window.

```

(Mac OS X users: Note that these openmath commands were run in a session of started in an xterm shell, not using the standard Mac Terminal application.)

```

sage: maxima.eval('load("plotdf");')
'"/local/share/maxima/.../share/dynamics/plotdf.lisp"'
sage: maxima.eval('plotdf(x+y, [trajectory_at,2,-0.1]); ') #optional

```

This plots a direction field (the plotdf Maxima package was also written by W. Schelter.)

A 2D plot of several functions:

```

sage: maxima.plot2d(' [x,x^2,x^3]', '[x,-1,1]', '[plot_format,openmath]') #optional

```

Openmath also does 3D plots of surfaces of the form $z = f(x, y)$, as x and y range over a rectangle. For example, here is a “live” 3D plot which you can move with your mouse:

```

sage: maxima.plot3d ("sin(x^2 + y^2)", "[x, -3, 3]", "[y, -3, 3]", \
...   '[plot_format, openmath]') #optional

```

By rotating this suitably, you can view the contour plot.

2.4 Tachyon 3D plotting

The ray-tracing package Tachyon is distributed with Sage. The 3D plots look very nice but tend to take a bit more setting up. Here is an example of a parametric space curve:

```

sage: f = lambda t: (t,t^2,t^3)
sage: t = Tachyon(camera_center=(5,0,4))
sage: t.texture('t')
sage: t.light((-20,-20,40), 0.2, (1,1,1))
sage: t.parametric_plot(f,-5,5,'t',min_depth=6)

```

Type `t.show()` to view this.

Other examples are in the Reference Manual.

2.5 gnuplot

You must have `gnuplot` installed to run these commands. This is an “experimental package” which, if it isn’t installed already on your machine, can be (hopefully!) installed by typing `./sage -i gnuplot-4.0.0` on the command line in the Sage home directory.

First, here’s way to plot a function: {plot!a function}

```
sage: maxima.plot2d('sin(x)', '[x, -5, 5]')
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot2d('sin(x)', '[x, -5, 5]', opts)
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "/tmp/sin-plot.eps"]'
sage: maxima.plot2d('sin(x)', '[x, -5, 5]', opts)
```

The eps file is saved by default to the current directory but you may specify a path if you prefer.

Here is an example of a plot of a parametric curve in the plane:

```
sage: maxima.plot2d_parametric(["sin(t)", "cos(t)"], "t", [-3.1, 3.1])
sage: opts = '[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], \
... [gnuplot_out_file, "circle-plot.eps"]'
sage: maxima.plot2d_parametric(["sin(t)", "cos(t)"], "t", [-3.1, 3.1], options=opts)
```

Here is an example of a plot of a parametric surface in 3-space: {plot!a parametric surface}

```
sage: maxima.plot3d_parametric(["v*sin(u)", "v*cos(u)", "v"], ["u", "v"], \
... [-3.2, 3.2], [0, 3]) # optional -- pops up a window.
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-cos-plot.eps"]'
sage: maxima.plot3d_parametric(["v*sin(u)", "v*cos(u)", "v"], ["u", "v"], \
... [-3.2, 3.2], [0, 3], opts) # optional -- pops up a window.
```

To illustrate how to pass `gnuplot` options in , here is an example of a plot of a set of points involving the Riemann zeta function $\zeta(s)$ (computed using Pari but plotted using Maxima and Gnuplot): {plot!points} {Riemann zeta function}

```
sage: zeta_ptsx = [ (pari(1/2 + i*I/10).zeta().real()).precision(1) \
... for i in range (70, 150)]
sage: zeta_ptsy = [ (pari(1/2 + i*I/10).zeta().imag()).precision(1) \
... for i in range (70, 150)]
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy) # optional -- pops up a window.
sage: opts='[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], \
... [gnuplot_out_file, "zeta.eps"]'
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy, opts) # optional -- pops up a window.
```

2.6 Plotting surfaces

To plot a surface in is no different that to plot a curve, though the syntax is slightly different. In particular, you need to have `surf` loaded. {plot!surface using surf}

```
sage: singular.eval('ring rr1 = 0, (x,y,z), dp;')
''
sage: singular.eval('ideal I(1) = 2x2-1/2x3 +1-y+1;')
''
```

```
sage: singular.eval('plot(I(1));')  
...
```


GROUPS

3.1 Permutation groups

A permutation group is a subgroup of some symmetric group S_n . Sage has a Python class `PermutationGroup`, so you can work with such groups directly:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G
Permutation Group with generators [(1,2,3)(4,5)]
sage: g = G.gens()[0]; g
(1,2,3)(4,5)
sage: g*g
(1,3,2)
sage: G = PermutationGroup(['(1,2,3)'])
sage: g = G.gens()[0]; g
(1,2,3)
sage: g.order()
3
```

For the example of the Rubik's cube group (a permutation subgroup of S_{48} , where the non-center facets of the Rubik's cube are labeled 1, 2, ..., 48 in some fixed way), you can use the GAP-Sage interface as follows.

```
sage: cube = "cubegp := Group(
( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19),
( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35),
(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11),
(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24),
(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27),
(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40) )"
sage: gap(cube)
'permutation group with 6 generators'
sage: gap("Size(cubegp)")
43252003274489856000'
```

Another way you can choose to do this:

- Create a file `cubegroup.py` containing the lines

```
cube = "cubegp := Group(
( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19),
( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35),
(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11),
(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24),
(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27),
(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40) )"
```

- Place the file in the subdirectory `$SAGE_ROOT/local/lib/python2.4/site-packages/sage` of your Sage home directory.
- Read (i.e., “{import}”) it into Sage:

```
sage: import sage.cubegroup
sage: sage.cubegroup.cube
'cubegp := Group(( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)
(11,35,27,19), ( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)
( 6,22,46,35), (17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)
( 8,30,41,11), (25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)
( 8,33,48,24), (33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)
( 1,14,48,27), (41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)
(16,24,32,40) )'
```

```
sage: gap(sage.cubegroup.cube)
'permutation group with 6 generators'
sage: gap("Size(cubegp)")
'43252003274489856000'
```

(You will have line wrap instead of the above carriage returns in your Sage output.)

- Use the CubeGroup class

```
sage: rubik = CubeGroup()
sage: rubik
The PermutationGroup of all legal moves of the Rubik's cube.
sage: print rubik
The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).
sage: G = rubik.group()
sage: G.order()
43252003274489856000
```

(1) has implemented classical groups (such as `:math:\text{'GU}(3,GF(5))\text{'}`) and matrix groups over a finite field with user-defined generators.

(2) also has implemented finite and infinite (but finitely generated) abelian groups.

3.2 Conjugacy classes

You can compute conjugacy classes of a finite group using “natively”:

```
sage: G = PermutationGroup(['(1,2,3)', '(1,2)(3,4)', '(1,7)'])
sage: CG = G.conjugacy_classes_representatives()
sage: gamma = CG[2]
sage: CG; gamma
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,7), (1,2,3,4), (1,2,3,4,7)]
(1,2)(3,4)
```

You can use the Sage-GAP interface.

```
sage: gap.eval("G := Group((1,2)(3,4), (1,2,3))")
'Group([ (1,2)(3,4), (1,2,3) ])'
sage: gap.eval("CG := ConjugacyClasses(G)")
'[ ()^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,4)^G ]'
```

```

sage: gap.eval("gamma := CG[3]")
' (1,2,3)^G'
sage: gap.eval("g := Representative(gamma)")
' (1,2,3)'

```

Or, here's another (more "pythonic") way to do this type of computation:

```

sage: G = gap.Group(' [(1,2,3), (1,2)(3,4), (1,7)]')
sage: CG = G.ConjugacyClasses()
sage: gamma = CG[2]
sage: g = gamma.Representative()
sage: CG; gamma; g
[ ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), ( ) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2)(3,4) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2,3) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2,3)(4,7) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2,3,4) ),
  ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2,3,4,7) ) ]
ConjugacyClass( SymmetricGroup( [ 1, 2, 3, 4, 7 ] ), (1,2) )
(1,2)

```

3.3 Normal subgroups

If you want to find all the normal subgroups of a permutation group G (up to conjugacy), you can use 's interface to GAP:

```

sage: G = AlternatingGroup( 5 )
sage: gap(G).NormalSubgroups()
[ Group( ( ) ), AlternatingGroup( [ 1 .. 5 ] ) ]

```

or

```

sage: G = gap("AlternatingGroup( 5 )")
sage: G.NormalSubgroups()
[ Group( ( ) ), AlternatingGroup( [ 1 .. 5 ] ) ]

```

Here's another way, working more directly with GAP:

```

sage: print gap.eval("G := AlternatingGroup( 5 )")
Alt( [ 1 .. 5 ] )
sage: print gap.eval("normal := NormalSubgroups( G )")
[ Group( ( ) ), Alt( [ 1 .. 5 ] ) ]
sage: G = gap.new("DihedralGroup( 10 )")
sage: G.NormalSubgroups()
[ Group( <identity> of ... ), Group( [ f2 ] ), Group( [ f1, f2 ] ) ]
sage: print gap.eval("G := SymmetricGroup( 4 )")
Sym( [ 1 .. 4 ] )
sage: print gap.eval("normal := NormalSubgroups( G );")
[ Group( ( ) ), Group( [ (1,4)(2,3), (1,3)(2,4) ] ),
  Group( [ (2,4,3), (1,4)(2,3), (1,3)(2,4) ] ), Sym( [ 1 .. 4 ] ) ]

```

3.4 Centers

How do you compute the center of a group in Sage?

Although Sage calls GAP to do the computation of the group center, `center` is “wrapped” (i.e., Sage has a class `PermutationGroup` with associated class method “center”), so the user does not need to use the `gap` command. Here’s an example:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by [()]
```

A similar syntax for matrix groups also works:

```
sage: G = SL(2, GF(5) )
sage: G.center()
Matrix group over Finite Field of size 5 with 1 generators:
[[[4, 0], [0, 4]]]
sage: G = PSL(2, 5 )
sage: G.center()
Subgroup of (The projective special linear group of degree 2 over Finite Field of size 5) generated by [()]
```

Note: `center` can be spelled either way in GAP, not so in Sage.

3.5 The group id database

The function `group_id` requires that the Small Groups Library of E. A. O’Brien, B. Eick, and H. U. Besche be installed (you can do this by typing `./sage -i database_gap-4.4.9` in the Sage home directory).

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
sage: G.order()
120
sage: G.group_id()      # requires optional GAP database package
[120, 34]
```

Another example of using the small groups database: `group_id`

```
sage: gap_console()
GAP4, Version: 4.4.6 of 02-Sep-2005, x86_64-unknown-linux-gnu-gcc
gap> G:=Group((4,6,5)(7,8,9), (1,7,2,4,6,9,5,3));
Group([ (4,6,5)(7,8,9), (1,7,2,4,6,9,5,3) ])
gap> StructureDescription(G);
"((C3 x C3) : Q8) : C3) : C2"
```

LINEAR ALGEBRA

4.1 Vector spaces

The `VectorSpace` command creates a vector space class, from which one can create a subspace. Note the basis computed by Sage is “row reduced”.

```
sage: V = VectorSpace(GF(2), 8)
sage: S = V.subspace([V([1, 1, 0, 0, 0, 0, 0, 0]), V([1, 0, 0, 0, 0, 1, 1, 0])])
sage: S.basis()
[
(1, 0, 0, 0, 0, 1, 1, 0),
(0, 1, 0, 0, 0, 1, 1, 0)
]
sage: S.dimension()
2
```

4.2 Matrix powers

How do I compute matrix powers in Sage? The syntax is illustrated by the example below.

```
sage: R = IntegerModRing(51)
sage: M = MatrixSpace(R, 3, 3)
sage: A = M([1, 2, 3, 4, 5, 6, 7, 8, 9])
sage: A^1000*A^1007
```

```
[ 3  3  3]
[18  0 33]
[33 48 12]
sage: A^2007
```

```
[ 3  3  3]
[18  0 33]
[33 48 12]
```

4.3 Kernels

The kernel is computed by applying the `kernel` method to the matrix object. The following examples illustrate the syntax.

```
sage: M = MatrixSpace(IntegerRing(), 4, 2) (range(8))
sage: M.kernel()
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3  2]
[ 0  1 -2  1]
```

A kernel of dimension one over \mathbb{Q} :

```
sage: A = MatrixSpace(RationalField(), 3) (range(9))
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

A trivial kernel:

```
sage: A = MatrixSpace(RationalField(), 2) ([1, 2, 3, 4])
sage: A.kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: M = MatrixSpace(RationalField(), 0, 2) (0)
sage: M
[]
sage: M.kernel()
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
sage: M = MatrixSpace(RationalField(), 2, 0) (0)
sage: M.kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1  0]
[0  1]
```

Kernel of a zero matrix:

```
sage: A = MatrixSpace(RationalField(), 2) (0)
sage: A.kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1  0]
[0  1]
```

Kernel of a non-square matrix:

```
sage: A = MatrixSpace(RationalField(), 3, 2) (range(6))
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

The 2-dimensional kernel of a matrix over a cyclotomic field:

```
sage: K = CyclotomicField(12); a = K.gen()
sage: M = MatrixSpace(K, 4, 2) ([1, -1, 0, -2, 0, -a^2-1, 0, a^2-1])
sage: M
[          1          -1]
[          0          -2]
```

```

[          0 -zeta12^2 - 1]
[          0  zeta12^2 - 1]
sage: M.kernel()
Vector space of degree 4 and dimension 2 over Cyclotomic Field of order 12
and degree 4
Basis matrix:
[          0          1          0 -2*zeta12^2]
[          0          0          1 -2*zeta12^2 + 1]

```

A nontrivial kernel over a complicated base field.

```

sage: K = FractionField(PolynomialRing(RationalField(), 2, 'x'))
sage: M = MatrixSpace(K, 2) ([[K.gen(1), K.gen(0)], [K.gen(1), K.gen(0)]])
sage: M
[x1 x0]
[x1 x0]
sage: M.kernel()
Vector space of degree 2 and dimension 1 over Fraction Field of Multivariate
Polynomial Ring in x0, x1 over Rational Field
Basis matrix:
[ 1 -1]

```

Other methods for integer matrices are `elementary_divisors`, `smith_form` (for the Smith normal form), `echelon_form` for the Hermite normal form, `frobenius` for the Frobenius normal form (rational canonical form).

There are many methods for matrices over a field such as \mathbb{Q} or a finite field: `row_span`, `nullity`, `transpose`, `swap_rows`, `matrix_from_columns`, `matrix_from_rows`, among many others.

See the file `matrix.py` for further details.

4.4 Eigenvectors and eigenvalues

How do you compute eigenvalues and eigenvectors using Sage?

Sage has a full range of functions for computing eigenvalues and both left and right eigenvectors and eigenspaces. If our matrix is A , then the `eigenmatrix_right` (resp. `eigenmatrix_left`) command also gives matrices D and P such that $AP = DP$ (resp. $PA = DP$.)

```

sage: A = matrix(QQ, [[1, 1, 0], [0, 2, 0], [0, 0, 3]])
sage: A
[1 1 0]
[0 2 0]
[0 0 3]
sage: A.eigenvalues()
[3, 2, 1]
sage: A.eigenvectors_right()
[(3, [(0, 0, 1)
], 1), (2, [(1, 1, 0)
], 1), (1, [(1, 0, 0)
], 1)]
sage: A.eigenspaces_right()
[(3, Vector space of degree 3 and dimension 1 over Rational Field

```

```
User basis matrix:
[0 0 1]),
(2, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[1 1 0]),
(1, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[1 0 0])
]
```

```
sage: D, P = A.eigenmatrix_right()
sage: D
[3 0 0]
[0 2 0]
[0 0 1]
sage: P
[0 1 1]
[0 1 0]
[1 0 0]
sage: A*P == P*D
True
```

For eigenvalues outside the fraction field of the base ring of the matrix, you can choose to have all the eigenspaces output when the algebraic closure of the field is implemented, such as the algebraic numbers, $\overline{\mathbb{Q}\mathbb{Q}}$. Or you may request just a single eigenspace for each irreducible factor of the characteristic polynomial, since the others may be formed through Galois conjugation. The eigenvalues of the matrix below are $\sqrt[3]{3}$ and we exhibit each possible output.

Also, currently Sage does not implement multiprecision numerical eigenvalues and eigenvectors, so calling the eigen functions on a matrix from $\mathbb{C}\mathbb{C}$ or $\mathbb{R}\mathbb{R}$ will probably give inaccurate and nonsensical results (a warning is also printed). Eigenvalues and eigenvectors of matrices with floating point entries (over $\mathbb{C}\mathbb{D}\mathbb{F}$ and $\mathbb{R}\mathbb{D}\mathbb{F}$) can be obtained with the “eigenmatrix” commands.

```
sage: MS = MatrixSpace(QQ, 2, 2)
sage: A = MS([1,-4,1, -1])
sage: A.eigenspaces_left(format='all')
[
(-1.732050807568878?*I, Vector space of degree 2 and dimension 1 over Algebraic Field
User basis matrix:
[
1 -1 - 1.732050807568878?*I]),
(1.732050807568878?*I, Vector space of degree 2 and dimension 1 over Algebraic Field
User basis matrix:
[
1 -1 + 1.732050807568878?*I])
]
sage: A.eigenspaces_left(format='galois')
[
(a0, Vector space of degree 2 and dimension 1 over Number Field in a0 with defining polynomial x^2 +
User basis matrix:
[
1 a0 - 1])
]
```

Another approach is to use the interface with Maxima:

```
sage: A = maxima("matrix ([1, -4], [1, -1])")
sage: eig = A.eigenvectors()
sage: eig
[[[-sqrt(3)*%i, sqrt(3)*%i], [1, 1]], [[1, (sqrt(3)*%i+1)/4]], [[1, -(sqrt(3)*%i-1)/4]]]
```

This tells us that $\vec{v}_1 = [1, (\sqrt{3}i + 1)/4]$ is an eigenvector of $\lambda_1 = -\sqrt{3}i$ (which occurs with multiplicity one) and

$\vec{v}_2 = [1, (-\sqrt{3}i + 1)/4]$ is an eigenvector of $\lambda_2 = \sqrt{3}i$ (which also occurs with multiplicity one).

Here are two more examples:

```
sage: A = maxima("matrix ([11, 0, 0], [1, 11, 0], [1, 3, 2])")
sage: A.eigenvectors()
[[[2, 11], [1, 2]], [[0, 0, 1], [0, 1, 1/3]]]
sage: A = maxima("matrix ([-1, 0, 0], [1, -1, 0], [1, 3, 2])")
sage: A.eigenvectors()
[[[-1, 2], [2, 1]], [[0, 1, -1], [0, 0, 1]]]
```

Warning: Notice how the ordering of the output is reversed, though the matrices are almost the same.

Finally, you can use Sage's GAP interface as well to compute "rational" eigenvalues and eigenvectors:

```
sage: print gap.eval("A := [[1,2,3],[4,5,6],[7,8,9]]")
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
sage: print gap.eval("v := Eigenvectors( Rationals,A)")
[ [ 1, -2, 1 ] ]
sage: print gap.eval("lambda := Eigenvalues( Rationals,A)")
[ 0 ]
```

4.5 Row reduction

The row reduced echelon form of a matrix is computed as in the following example.

```
sage: M = MatrixSpace(RationalField(), 2, 3)
sage: A = M([1, 2, 3, 4, 5, 6])
sage: A
[1 2 3]
[4 5 6]
sage: A.parent()
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: A[0,2] = 389
sage: A
[ 1  2 389]
[ 4  5  6]
sage: A.echelon_form()
[  1  0 -1933/3]
[  0  1  1550/3]
```

4.6 Characteristic polynomial

The characteristic polynomial is a Sage method for square matrices.

First a matrix over \mathbf{Z} :

```
sage: A = MatrixSpace(IntegerRing(), 2) ( [[1, 2], [3, 4]] )
sage: f = A.charpoly()
sage: f
x^2 - 5*x - 2
sage: f.parent()
Univariate Polynomial Ring in x over Integer Ring
```

We compute the characteristic polynomial of a matrix over the polynomial ring $\mathbf{Z}[a]$:

```

sage: R = PolynomialRing(IntegerRing(), 'a'); a = R.gen()
sage: M = MatrixSpace(R, 2) ([[a, 1], [a, a+1]])
sage: M
[  a      1]
[  a a + 1]
sage: f = M.charpoly()
sage: f
x^2 + (-2*a - 1)*x + a^2
sage: f.parent()
Univariate Polynomial Ring in x over Univariate Polynomial Ring in a over
Integer Ring

sage: M.trace()
2*a + 1
sage: M.determinant()
a^2

```

We compute the characteristic polynomial of a matrix over the multi-variate polynomial ring $\mathbf{Z}[u, v]$:

```

sage: R.<u,v> = PolynomialRing(ZZ, 2)
sage: A = MatrixSpace(R, 2) ([u, v, u^2, v^2])
sage: f = A.charpoly(); f
x^2 + (-v^2 - u)*x - u^2*v + u*v^2

```

It's a little difficult to distinguish the variables. To fix this, we might want to rename the indeterminate “Z”, which we can easily do as follows:

```

sage: f = A.charpoly('Z'); f
Z^2 + (-v^2 - u)*Z - u^2*v + u*v^2

```

4.7 Solving systems of linear equations

Using maxima, you can easily solve linear equations:

```

sage: var('a,b,c')
(a, b, c)
sage: eqn = [a+b*c==1, b-a*c==0, a+b==5]
sage: s = solve(eqn, a,b,c); s
[[a == (25*I*sqrt(79) + 25)/(6*I*sqrt(79) - 34),
  b == (5*I*sqrt(79) + 5)/(I*sqrt(79) + 11),
  c == 1/10*I*sqrt(79) + 1/10],
 [a == (25*I*sqrt(79) - 25)/(6*I*sqrt(79) + 34),
  b == (5*I*sqrt(79) - 5)/(I*sqrt(79) - 11),
  c == -1/10*I*sqrt(79) + 1/10]]

```

You can even nicely typeset the solution in LaTeX:

```

sage.: print latex(s)
...

```

To have the above appear onscreen via xdvi, type `view(s)`.

You can also solve linear equations symbolically using the `solve` command:

```

sage: var('x,y,z,a')
(x, y, z, a)
sage: eqns = [x + z == y, 2*a*x - y == 2*a^2, y - 2*z == 2]

```

```
sage: solve(eqns, x, y, z)
[[x == a + 1, y == 2*a, z == a - 1]]
```

Here is a numerical Numpy example:

```
sage: from numpy import arange, eye, linalg
sage: A = eye(10)      ## the 10x10 identity matrix
sage: b = arange(1,11)
sage: x = linalg.solve(A,b)
```

Another way to solve a system numerically is to use Sage's octave interface:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A   = M33([1,2,3,4,5,6,7,8,0])
sage: V3  = VectorSpace(QQ,3)
sage: b   = V3([1,2,3])
sage: octave.solve_linear_system(A,b) # requires optional octave
[-0.33333299999999999, 0.66666700000000001, 0]
```


LINEAR CODES AND CIPHERS

5.1 Codes

A linear code of length n is a finite dimensional subspace of $GF(q)^n$. Sage can compute with linear error-correcting codes to a limited extent. It basically has some wrappers to GAP and GUAVA commands. GUAVA 2.8 is not included with Sage 4.0's install of GAP but can be installed as an optional package.

Sage can compute Hamming codes

```
sage: C = HammingCode(3,GF(3))
sage: C
Linear code of length 13, dimension 10 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C.gen_mat()
[1 0 0 0 0 0 0 0 0 0 0 1 2 0]
[0 1 0 0 0 0 0 0 0 0 0 0 1 2]
[0 0 1 0 0 0 0 0 0 0 0 1 0 2]
[0 0 0 1 0 0 0 0 0 0 0 1 1 1]
[0 0 0 0 1 0 0 0 0 0 0 1 1 2]
[0 0 0 0 0 1 0 0 0 0 0 2 0 2]
[0 0 0 0 0 0 1 0 0 0 0 1 2 1]
[0 0 0 0 0 0 0 1 0 0 0 2 1 1]
[0 0 0 0 0 0 0 0 1 0 2 2 0]
[0 0 0 0 0 0 0 0 0 1 0 1 1]
```

the four Golay codes

```
sage: C = ExtendedTernaryGolayCode()
sage: C
Linear code of length 12, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
6
sage: C.gen_mat()
[1 0 0 0 0 0 2 0 1 2 1 2]
[0 1 0 0 0 0 1 2 2 2 1 0]
[0 0 1 0 0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 1 1 0 2 2 2]
[0 0 0 0 1 0 2 1 2 2 0 1]
[0 0 0 0 0 1 0 2 1 2 2 1]
```

as well as binary Reed-Muller codes, quadratic residue codes, quasi-quadratic residue codes, “random” linear codes, and a code generated by a matrix of full rank (using, as usual, the rows as the basis).

For a given code, C , Sage can return a generator matrix, a check matrix, and the dual code:

```

sage: C = HammingCode(3,GF(2))
sage: Cperp = C.dual_code()
sage: C; Cperp
Linear code of length 7, dimension 4 over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C.gen_mat()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
sage: C.check_mat()
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
sage: C.dual_code()
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C = HammingCode(3,GF(4,'a'))
sage: C.dual_code()
Linear code of length 21, dimension 3 over Finite Field in a of size 2^2

```

For C and a vector $v \in GF(q)^n$, Sage can try to decode v (i.e., find the codeword $c \in C$ closest to v in the Hamming metric) using syndrome decoding. As of yet, no special decoding methods have been implemented.

```

sage: C = HammingCode(3,GF(2))
sage: MS = MatrixSpace(GF(2),1,7)
sage: F = GF(2); a = F.gen()
sage: v1 = [a,a,F(0),a,a,F(0),a]
sage: C.decode(v1)
(1, 1, 0, 1, 0, 0, 1)
sage: v2 = matrix([[a,a,F(0),a,a,F(0),a]])
sage: C.decode(v2)
(1, 1, 0, 1, 0, 0, 1)
sage: v3 = vector([a,a,F(0),a,a,F(0),a])
sage: c = C.decode(v3); c
(1, 1, 0, 1, 0, 0, 1)

```

To plot the (histogram of) the weight distribution of a code, one can use the matplotlib package included with Sage:

```

sage: C = HammingCode(4,GF(2))
sage: C
Linear code of length 15, dimension 11 over Finite Field of size 2
sage: w = C.weight_distribution(); w
[1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1]
sage: J = range(len(w))
sage: W = IndexedSequence([ZZ(w[i]) for i in J],J)
sage: P = W.plot_histogram()

```

Now type `show(P)` to view this.

There are several coding theory functions we are skipping entirely. Please see the reference manual or the file `coding/linear_codes.py` for examples.

Sage can also compute algebraic-geometric codes, called AG codes, via the Singular interface `$ sec:agcodes`. One may also use the AG codes implemented in GUAVA via the Sage interface to GAP `gap_console()`. See the GUAVA manual for more details. {GUAVA}

5.2 Ciphers

5.2.1 LFSRs

A special type of stream cipher is implemented in Sage, namely, a linear feedback shift register (LFSR) sequence defined over a finite field. Stream ciphers have been used for a long time as a source of pseudo-random number generators. {linear feedback shift register}

S. Golomb {G} gives a list of three statistical properties a sequence of numbers $\mathbf{a} = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$, should display to be considered “random”. Define the autocorrelation of \mathbf{a} to be

$$C(k) = C(k, \mathbf{a}) = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=1}^N (-1)^{a_n + a_{n+k}}.$$

In the case where a is periodic with period P then this reduces to

Assume a is periodic with period P .

- balance: $|\sum_{n=1}^P (-1)^{a_n}| \leq 1$.
- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

(For sequences satisfying these first two properties, it is known that $\epsilon = -1/P$ must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1’s as there are of 0’s.

A sequence satisfying these properties will be called pseudo-random. {pseudo-random}

A general feedback shift register is a map $f : \mathbf{F}_q^d \rightarrow \mathbf{F}_q^d$ of the form

$$\begin{aligned} f(x_0, \dots, x_{n-1}) &= (x_1, x_2, \dots, x_n), \\ x_n &= C(x_0, \dots, x_{n-1}), \end{aligned}$$

where $C : \mathbf{F}_q^d \rightarrow \mathbf{F}_q$ is a given function. When C is of the form

$$C(x_0, \dots, x_{n-1}) = c_0 x_0 + \dots + c_{n-1} x_{n-1},$$

for some given constants $c_i \in \mathbf{F}_q$, the map is called a linear feedback shift register (LFSR). The sequence of coefficients c_i is called the key and the polynomial

is sometimes called the connection polynomial.

Example: Over $GF(2)$, if $[c_0, c_1, c_2, c_3] = [1, 0, 0, 1]$ then $C(x) = 1 + x + x^4$,

The LFSR sequence is then

$$\begin{aligned} &1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, \\ &1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, \dots \end{aligned}$$

The sequence of 0, 1’s is periodic with period $P = 2^4 - 1 = 15$ and satisfies Golomb’s three randomness conditions. However, this sequence of period 15 can be “cracked” (i.e., a procedure to reproduce $g(x)$) by knowing only 8 terms! This is the function of the Berlekamp-Massey algorithm {M}, implemented as `lfsr_connection_polynomial` (which produces the reverse of `berlekamp_massey`).

```
sage: F = GF(2)
sage: o = F(0)
sage: l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n); s
[1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: lfsr_autocorrelation(s,15,7)
4/15
sage: lfsr_autocorrelation(s,15,0)
8/15
sage: lfsr_connection_polynomial(s)
x^4 + x + 1
sage: berlekamp_massey(s)
x^4 + x^3 + 1
```

5.2.2 Classical ciphers

has a type for cryptosystems (created by David Kohel, who also wrote the examples below), implementing classical cryptosystems. The general interface is as follows:

```
sage: S = AlphabeticStrings()
sage: S
Free alphabetic string monoid on A-Z
sage: E = SubstitutionCryptosystem(S)
sage: E
Substitution cryptosystem on Free alphabetic string monoid on A-Z
sage: K = S([ 25-i for i in range(26) ])
sage: e = E(K)
sage: m = S("THECATINTHEHAT")
sage: e(m)
GSVXZGRMGSVSZG
```

Here's another example:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S,15);
sage: m = S("THECATANDTHEHAT")
sage: G = E.key_space()
sage: G
Symmetric group of order 15! as a permutation group
sage: g = G([ 3, 2, 1, 6, 5, 4, 9, 8, 7, 12, 11, 10, 15, 14, 13 ])
sage: e = E(g)
sage: e(m)
EHTTACDNAEHTTAH
```

The idea is that a cryptosystem is a map $E : KS \rightarrow \text{Hom}_{\text{Set}}(MS, CS)$ where KS , MS , and CS are the key space, plaintext (or message) space, and ciphertext space, respectively. E is presumed to be injective, so `e.key()` returns the pre-image key.

GRAPH THEORY

See the Sage wiki page http://wiki.sagemath.org/graph_survey for an excellent survey of existing graph theory software.

6.1 Networkx

Networkx (<http://networkx.lanl.gov>) “is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks”. More details can also be found on http://wiki.sagemath.org/graph_survey or in Robert Miller’s SageDays 3 talk.

```
sage: C = graphs.CubeGraph(4)
```

Now `type C.show(vertex_labels=False, vertex_size=60, graph_border=True, figsize=[9, 8])` to view this with some of the options.

The digraph below is a 3-cycle with vertices $\{0, 1, 2\}$ and edges $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$:

```
sage: D = DiGraph( { 0: [1], 1: [2], 2: [0] } )
```

Type `D.show()` to view this.

6.2 Cayley graphs

includes wrappers to many NetworkX commands, written mainly by Emily Kirkman and Robert Miller. The implementation of Cayley graphs was written by Bobby Moretti and Robert Miller.

```
sage: G = DihedralGroup(5)
sage: C = G.cayley_graph(); C
Digraph on 10 vertices
sage: C.diameter()
3
sage: C.girth()
4
sage: C.automorphism_group().order()
10
sage: len(C.edges())
20
```

6.3 Graphs from adjacency matrices

To construct the graph G with $n \times n$ adjacency matrix A , you want a graph X so that the vertex-set of G is $\{1, \dots, n\}$, and $[i, j]$ is an edge of G if and only if $A[i][j] = 1$.

Here is an example of the syntax in (copied from Robert Miller's SageDays 3 talk): Define the distance $d(x, y)$ from x to y to be the minimum length of a (directed) path in G joining a vertex x to a vertex y if such a path exists, and -1 otherwise. A diameter of -1 is returned if G is not (strongly) connected. Otherwise, the diameter of G is equal to the maximum (directed) distance $d(x, y)$ in G (as x and y range over all the vertices of G).

```
sage: M = Matrix ([ [0, 1, 1], [1, 0, 1], [1, 1, 0] ])
sage: # (the order is the number of edges)
sage: G = Graph(M); G.order()
3
sage: G.distance(0,2)
1
sage: G.diameter()
1
```

REPRESENTATION THEORY

7.1 Ordinary characters

How can you compute character tables of a finite group in Sage? The Sage-GAP interface can be used to compute character tables.

You can construct the table of character values of a permutation group G as a Sage matrix, using the method `character_table` of the `PermutationGroup` class, or via the pexpect interface to the GAP command `CharacterTable`.

```
sage: G = PermutationGroup([[ (1,2), (3,4) ], [ (1,2,3,4) ]])
sage: G.order()
8
sage: G.character_table()
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
[ 2  0  0  0 -2]
sage: CT = gap(G).CharacterTable()
sage: print gap.eval("Display(%s)"%CT.name())
CT1
```

```

 2  3  2  2  2  3

 1a 2a 2b 4a 2c
2P 1a 1a 1a 2c 1a
3P 1a 2a 2b 4a 2c

X.1  1  1  1  1  1
X.2  1 -1 -1  1  1
X.3  1 -1  1 -1  1
X.4  1  1 -1 -1  1
X.5  2  .  .  . -2
```

Here is another example:

```
sage: G = PermutationGroup([[ (1,2), (3,4) ], [ (1,2,3) ]])
sage: G.character_table()
[      1      1      1      1]
[      1      1 -zeta3 - 1      zeta3]
[      1      1      zeta3 -zeta3 - 1]
[      3     -1      0      0]
sage: gap.eval("G := Group((1,2) (3,4), (1,2,3))")
```

```
'Group([ (1,2)(3,4), (1,2,3) ])'
sage: gap.eval("T := CharacterTable(G)")
'CharacterTable( Alt( [ 1 .. 4 ] ) )'
sage: print gap.eval("Display(T)")
CT2
```

```
  2  2  2  .  .
  3  1  .  1  1
```

```
  1a 2a 3a 3b
2P 1a 1a 3b 3a
3P 1a 2a 1a 1a
```

```
X.1  1  1  1  1
X.2  1  1  A /A
X.3  1  1 /A  A
X.4  3 -1  .  .
```

```
A = E(3)^2
    = (-1-ER(-3))/2 = -1-b3
```

where $E(3)$ denotes a cube root of unity, $ER(-3)$ denotes a square root of -3 , say $i\sqrt{3}$, and $b3 = \frac{1}{2}(-1 + i\sqrt{3})$. Note the added `print` Python command. This makes the output look much nicer.

```
sage: print gap.eval("irr := Irr(G)")
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 3, -1, 0, 0 ] ) ]
sage: print gap.eval("Display(irr)")
[ [ 1, 1, 1, 1 ],
  [ 1, 1, E(3)^2, E(3) ],
  [ 1, 1, E(3), E(3)^2 ],
  [ 3, -1, 0, 0 ] ]
sage: gap.eval("CG := ConjugacyClasses(G)")
'[ ()^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,4)^G ]'
sage: gap.eval("gamma := CG[3]")
'(1,2,3)^G'
sage: gap.eval("g := Representative(gamma)")
'(1,2,3)''
sage: gap.eval("chi := irr[2]")
'Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] )'
sage: gap.eval("g^chi")
'E(3)^2'
```

This last quantity is the value of the character `chi` at the group element `g`.

Alternatively, if you turn IPython “pretty printing” off, then the table prints nicely.

```
sage: %Pprint
Pretty printing has been turned OFF
sage: gap.eval("G := Group((1,2)(3,4), (1,2,3))")
'Group([ (1,2)(3,4), (1,2,3) ])'
sage: gap.eval("T := CharacterTable(G)")
'CharacterTable( Alt( [ 1 .. 4 ] ) )'
sage: gap.eval("Display(T)")
CT1

  2  2  2  .  .
```

```

      3 1 . 1 1
      1a 2a 3a 3b
2P 1a 1a 3b 3a
3P 1a 2a 1a 1a

X.1 1 1 1 1
X.2 1 1 A /A
X.3 1 1 /A A
X.4 3 -1 . .

A = E(3)^2
  = (-1-ER(-3))/2 = -1-b3
sage: gap.eval("irr := Irr(G)")
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 3, -1, 0, 0 ] ) ]
sage: gap.eval("Display(irr)")
[ [ 1, 1, 1, 1 ],
  [ 1, 1, E(3)^2, E(3) ],
  [ 1, 1, E(3), E(3)^2 ],
  [ 3, -1, 0, 0 ] ]
sage: %Pprint
Pretty printing has been turned ON

```

7.2 Brauer characters

The Brauer character tables in GAP do not yet have a “native” interface. To access them you can directly interface with GAP using `pexpect` and the `gap.eval` command.

The example below using the GAP interface illustrates the syntax.

```

sage: print gap.eval("G := Group((1,2)(3,4), (1,2,3))")
Group([ (1,2)(3,4), (1,2,3) ])
sage: print gap.eval("irr := IrreducibleRepresentations(G,GF(7))") # random arch. dependent output
[ [ (1,2)(3,4), (1,2,3) ] -> [ [ Z(7)^0 ], [ Z(7)^4 ] ],
  [ (1,2)(3,4), (1,2,3) ] -> [ [ Z(7)^0 ], [ Z(7)^2 ] ],
  [ (1,2)(3,4), (1,2,3) ] -> [ [ Z(7)^0 ], [ Z(7)^0 ] ],
  [ (1,2)(3,4), (1,2,3) ] ->
    [ [ Z(7)^2, Z(7)^5, Z(7) ], [ Z(7)^3, Z(7)^2, Z(7)^3 ],
      [ Z(7), Z(7)^5, Z(7)^2 ] ],
  [ [ 0*Z(7), Z(7)^0, 0*Z(7) ], [ 0*Z(7), 0*Z(7), Z(7)^0 ],
    [ Z(7)^0, 0*Z(7), 0*Z(7) ] ] ]
sage: gap.eval("brvals := List(irr, chi->List(ConjugacyClasses(G), c->BrauerCharacterValue(Image(chi, R), R)))")
sage: print gap.eval("Display(brvals)") # random architecture dependent output
[ [ 1, 1, E(3)^2, E(3) ],
  [ 1, 1, E(3), E(3)^2 ],
  [ 1, 1, 1, 1 ],
  [ 3, -1, 0, 0 ] ]
sage: print gap.eval("T := CharacterTable(G)")
CharacterTable( Alt( [ 1 .. 4 ] ) )
sage: print gap.eval("Display(T)")
CT3

```

```
2 2 2 . .
3 1 . 1 1
```

```
1a 2a 3a 3b
2P 1a 1a 3b 3a
3P 1a 2a 1a 1a
```

```
X.1 1 1 1 1
X.2 1 1 A /A
X.3 1 1 /A A
X.4 3 -1 . .
```

$$\begin{aligned} A &= E(3)^2 \\ &= (-1-ER(-3))/2 = -1-b3 \end{aligned}$$

RINGS

8.1 Matrix rings

How do you construct a matrix ring over a finite ring in Sage? The `MatrixSpace` constructor accepts any ring as a base ring. Here's an example of the syntax:

```
sage: R = IntegerModRing(51)
sage: M = MatrixSpace(R, 3, 3)
sage: M(0)
[0 0 0]
[0 0 0]
[0 0 0]
sage: M(1)
[1 0 0]
[0 1 0]
[0 0 1]
sage: 5*M(1)
[5 0 0]
[0 5 0]
[0 0 5]
```

8.2 Polynomial rings

How do you construct a polynomial ring over a finite field in Sage? Here's an example:

```
sage: R = PolynomialRing(GF(97), 'x')
sage: x = R.gen()
sage: f = x^2+7
sage: f in R
True
```

Here's an example using the Singular interface:

```
sage: R = singular.ring(97, '(a,b,c,d)', 'lp')
sage: I = singular.ideal(['a+b+c+d', 'ab+ad+bc+cd', 'abc+abd+acd+bcd', 'abcd-1'])
sage: R
// characteristic : 97
// number of vars : 4
//      block 1 : ordering lp
//      : names a b c d
//      block 2 : ordering C
sage: I
```

```
a+b+c+d,
a*b+a*d+b*c+c*d,
a*b*c+a*b*d+a*c*d+b*c*d,
a*b*c*d-1
```

Here is another approach using GAP:

```
sage: R = gap.new("PolynomialRing(GF(97), 4)"); R
PolynomialRing( GF(97), ["x_1", "x_2", "x_3", "x_4"] )
sage: I = R.IndeterminatesOfPolynomialRing(); I
[ x_1, x_2, x_3, x_4 ]
sage: vars = (I.name(), I.name(), I.name(), I.name())
sage: _ = gap.eval("x_0 := %s[1];; x_1 := %s[2];; x_2 := %s[3];;\
... x_3 := %s[4];;%vars)
sage: f = gap.new("x_1*x_2+x_3"); f
x_2*x_3+x_4
sage: f.Value(I, [1,1,1,1])
Z(97)^34
```

8.3 p -adic numbers

How do you construct p -adics in Sage? A great deal of progress has been made on this (see SageDays talks by David Harvey and David Roe). Here only a few of the simplest examples are given.

To compute the characteristic and residue class field of the ring \mathbb{Z}_p of integers of \mathbb{Q}_p , use the syntax illustrated by the following examples.

```
sage: K = Qp(3)
sage: K.residue_class_field()
Finite Field of size 3
sage: K.residue_characteristic()
3
sage: a = K(1); a
1 + O(3^20)
sage: 82*a
1 + 3^4 + O(3^20)
sage: 12*a
3 + 3^2 + O(3^21)
sage: a in K
True
sage: b = 82*a
sage: b^4
1 + 3^4 + 3^5 + 2*3^9 + 3^12 + 3^13 + 3^16 + O(3^20)
```

8.4 Quotient rings of polynomials

How do you construct a quotient ring in Sage?

We create the quotient ring $GF(97)[x]/(x^3 + 7)$, and demonstrate many basic functions with it.

```
sage: R = PolynomialRing(GF(97), 'x')
sage: x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a')
sage: a = S.gen()
sage: S
```

```

Univariate Quotient Polynomial Ring in a over Finite Field of size 97 with
modulus x^3 + 7
sage: S.is_field()
True
sage: a in S
True
sage: x in S
True
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Finite Field of size 97
sage: S.modulus()
x^3 + 7
sage: S.degree()
3

```

In Sage, `in` means that there is a “canonical coercion” into the ring. So the integer x and a are both in S , although x really needs to be coerced.

You can also compute in quotient rings without actually computing then using the command `quo_rem` as follows.

```

sage: R = PolynomialRing(GF(97), 'x')
sage: x = R.gen()
sage: f = x^7+1
sage: (f^3).quo_rem(x^7-1)
(x^14 + 4*x^7 + 7, 8)

```


POLYNOMIALS

9.1 Polynomial powers

How do I compute modular polynomial powers in Sage?

To compute $x^{2006} \pmod{x^3 + 7}$ in $GF(97)[x]$, we create the quotient ring $GF(97)[x]/(x^3 + 7)$, and compute x^{2006} in it. As a matter of Sage notation, we must distinguish between the x in $GF(97)[x]$ and the corresponding element (which we denote by a) in the quotient ring $GF(97)[x]/(x^3 + 7)$.

```
sage: R = PolynomialRing(GF(97), 'x')
sage: x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a')
sage: a = S.gen()
sage: S
Univariate Quotient Polynomial Ring in a over
Finite Field of size 97 with modulus x^3 + 7
sage: a^2006
4*a^2
```

Another approach to this:

```
sage: R = PolynomialRing(GF(97), 'x')
sage: x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a')
sage: a = S.gen()
sage: a^20062006
80*a
sage: print gap.eval("R:= PolynomialRing( GF(97))")
GF(97) [x_1]
sage: print gap.eval("i:= IndeterminatesOfPolynomialRing(R)")
[ x_1 ]
sage: gap.eval("x:= i[1];; f:= x;")
,,
sage: print gap.eval("PowerMod( R, x, 20062006, x^3+7 );")
Z(97)^41*x_1
sage: print gap.eval("PowerMod( R, x, 20062006, x^3+7 );")
Z(97)^41*x_1
sage: print gap.eval("PowerMod( R, x, 2006200620062006, x^3+7 );")
Z(97)^4*x_1^2
sage: a^2006200620062006
43*a^2
sage: print gap.eval("PowerMod( R, x, 2006200620062006, x^3+7 );")
Z(97)^4*x_1^2
```

```
sage: print gap.eval("Int(Z(97)^4)")
43
```

9.2 Factorization

You can factor a polynomial using Sage.

Using Sage to factor a univariate polynomial is a matter of applying the method `factor` to the `PolynomialRingElement` object `f`. In fact, this method actually calls Pari, so the computation is fairly fast.

```
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = (x^3 - 1)^2 - (x^2 - 1)^2
sage: f.factor()
(x - 1)^2 * x^2 * (x^2 + 2*x + 2)
```

Using the Singular interface, Sage also factors multivariate polynomials.

```
sage: x, y = PolynomialRing(RationalField(), 2, ['x', 'y']).gens()
sage: f = 9*y^6 - 9*x^2*y^5 - 18*x^3*y^4 - 9*x^5*y^4 + 9*x^6*y^2 + 9*x^7*y^3\
... + 18*x^8*y^2 - 9*x^11
sage: f.factor()
(-9) * (x^5 - y^2) * (x^6 - 2*x^3*y^2 - x^2*y^3 + y^4)
```

9.3 Polynomial GCD's

This example illustrates single variable polynomial GCD's:

```
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = 3*x^3 + x
sage: g = 9*x*(x+1)
sage: f.gcd(g)
x
```

This example illustrates multivariate polynomial GCD's:

```
sage: R = PolynomialRing(RationalField(), 3, ['x', 'y', 'z'], 'lex')
sage: x, y, z = PolynomialRing(RationalField(), 3, ['x', 'y', 'z'], 'lex').gens()
sage: f = 3*x^2*(x+y)
sage: g = 9*x*(y^2 - x^2)
sage: f.gcd(g)
x^2 + x*y
```

Here's another way to do this:

```
sage: R2 = singular.ring(0, '(x,y,z)', 'lp')
sage: a = singular.new('3x2*(x+y)')
sage: b = singular.new('9x*(y2-x2)')
sage: g = a.gcd(b)
sage: g
x^2+x*y
```

This example illustrates univariate polynomial GCD's via the GAP interface.

```
sage: R = gap.PolynomialRing(gap.GF(2)); R
PolynomialRing( GF(2), ["x_1"] )
```

```

sage: i = R.IndeterminatesOfPolynomialRing(); i
[ x_1 ]
sage: x_1 = i[1]
sage: f = (x_1^3 - x_1 + 1)*(x_1 + x_1^2); f
x_1^5+x_1^4+x_1^3+x_1
sage: g = (x_1^3 - x_1 + 1)*(x_1 + 1); g
x_1^4+x_1^3+x_1^2+Z(2)^0
sage: f.Gcd(g)
x_1^4+x_1^3+x_1^2+Z(2)^0

```

We can, of course, do the same computation in `Sage`, which uses the NTL library (which does huge polynomial gcd's over finite fields very quickly).

```

sage: x = PolynomialRing(GF(2), 'x').gen()
sage: f = (x^3 - x + 1)*(x + x^2); f
x^5 + x^4 + x^3 + x
sage: g = (x^3 - x + 1)*(x + 1)
sage: f.gcd(g)
x^4 + x^3 + x^2 + 1

```

9.4 Roots of polynomials

Sage can compute roots of a univariate polynomial.

```

sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = x^3 - 1
sage: f.roots()
[(1, 1)]
sage: f = (x^3 - 1)^2
sage: f.roots()
[(1, 2)]
sage: x = PolynomialRing(CyclotomicField(3), 'x').gen()
sage: f = x^3 - 1
sage: f.roots()
[(1, 1), (zeta3, 1), (-zeta3 - 1, 1)]

```

The first of the pair is the root, the second of the pair is its multiplicity.

There are some situations where GAP does find the roots of a univariate polynomial but GAP does not do this generally. (The roots must generate either a finite field or a subfield of a cyclotomic field.) However, there is a GAP package called `RadiRoot`, which must be installed into 's' installation of GAP, which does help to do this for polynomials with rational coefficients (`radiroot` itself requires other packages to be installed; please see its webpage for more details). The `Factors` command actually has an option which allows you to increase the groundfield so that a factorization actually returns the roots. Please see the examples given in section 64.10 "Polynomial Factorization" of the GAP Reference Manual for more details.

9.5 Evaluation of multivariate functions

You can evaluate polynomials in Sage as usual by substituting in points:

```

sage: x = PolynomialRing(RationalField(), 3, 'x').gens()
sage: f = x[0] + x[1] - 2*x[1]*x[2]
sage: f
-2*x1*x2 + x0 + x1

```

```
sage: f(1,2,0)
3
sage: f(1,2,5)
-17
```

This also will work with rational functions:

```
sage: h = f / (x[1] + x[2])
sage: h
(-2*x1*x2 + x0 + x1)/(x1 + x2)
sage: h(1,2,3)
-9/5
```

Sage also performs symbolic manipulation:

```
sage: var('x,y,z')
(x, y, z)
sage: f = (x + 3*y + x^2*y)^3; f
(x^2*y + x + 3*y)^3
sage: f(x=1,y=2,z=3)
729
sage: f.expand()
x^6*y^3 + 3*x^5*y^2 + 9*x^4*y^3 + 3*x^4*y + 18*x^3*y^2 +
27*x^2*y^3 +
x^3 + 9*x^2*y + 27*x*y^2 + 27*y^3
sage: f(x = 5/z)
(3*y + 25*y/z^2 + 5/z)^3
sage: g = f.subs(x = 5/z); g
(3*y + 25*y/z^2 + 5/z)^3
sage: h = g.rational_simplify(); h
(27*y^3*z^6 + 135*y^2*z^5 + 225*(3*y^3 + y)*z^4 + 125*(18*y^2 + 1)*z^3 +
1875*(3*y^3 + y)*z^2 + 15625*y^3 + 9375*y^2*z)/z^6
```

9.6 Roots of multivariate polynomials

Sage (using the interface to Singular) can solve multivariate polynomial equations in some situations (they assume that the solutions form a zero-dimensional variety) using Gröbner bases. Here is a simple example:

```
sage: R = PolynomialRing(QQ, 2, 'ab', order='lp')
sage: a,b = R.gens()
sage: I = (a^2-b^2-3, a-2*b)*R
sage: B = I.groebner_basis(); B
[a - 2*b, b^2 - 1]
```

So $b = \pm 1$ and $a = 2b$.

9.7 Gröbner bases

This computation uses Singular behind the scenes to compute the Gröbner basis.

```
sage: R = PolynomialRing(QQ, 4, 'abcd', order='lp')
sage: a,b,c,d = R.gens()
sage: I = (a+b+c+d, a*b+a*d+b*c+c*d, a*b*c+a*b*d+a*c*d+b*c*d, a*b*c*d-1)*R; I
Ideal (a + b + c + d, a*b + a*d + b*c + c*d, a*b*c + a*b*d + a*c*d + b*c*d,
a*b*c*d - 1) of Multivariate Polynomial Ring in a, b, c, d over Rational Field
```

```

sage: B = I.groebner_basis(); B
[a + b + c + d,
 b^2 + 2*b*d + d^2,
 b*c - b*d + c^2*d^4 + c*d - 2*d^2,
 b*d^4 - b + d^5 - d,
 c^3*d^2 + c^2*d^3 - c - d,
 c^2*d^6 - c^2*d^2 - d^4 + 1]

```

You can work with multiple rings without having to switch back and forth like in Singular. For example,

```

sage: a,b,c = QQ['a,b,c'].gens()
sage: X,Y = GF(7)['X,Y'].gens()
sage: I = ideal(a, b^2, b^3+c^3)
sage: J = ideal(X^10 + Y^10)

sage: I.minimal_associated_primes ()
[Ideal (c, b, a) of Multivariate Polynomial Ring in a, b, c over Rational Field]

sage: J.minimal_associated_primes ()      # slightly random output
[Ideal (Y^4 + 3*X*Y^3 + 4*X^2*Y^2 + 4*X^3*Y + X^4) of Multivariate Polynomial
Ring in X, Y over Finite Field of size 7,
 Ideal (Y^4 + 4*X*Y^3 + 4*X^2*Y^2 + 3*X^3*Y + X^4) of Multivariate Polynomial
Ring in X, Y over Finite Field of size 7,
 Ideal (Y^2 + X^2) of Multivariate Polynomial Ring in X, Y over Finite Field
of size 7]

```

All the real work is done by Singular.

Sage also includes `gfan` which provides other fast algorithms for computing Gröbner bases. See the section on “Gröbner fans” in the Reference Manual for more details.

ELEMENTARY NUMBER THEORY

10.1 Taking modular powers

How do I compute modular powers in Sage?

To compute $51^{2006} \pmod{97}$ in Sage, type

```
sage: R = Integers(97)
sage: a = R(51)
sage: a^2006
12
```

Instead of `R = Integers(97)` you can also type `R = IntegerModRing(97)`. Another option is to use the interface with GMP:

```
sage: 51.powermod(99203843984, 97)
96
```

10.2 Discrete logs

To find a number x such that $b^x \equiv a \pmod{m}$ (the discrete log of $a \pmod{m}$), you can call 's log command:

```
sage: r = Integers(125)
sage: b = r.multiplicative_generator()^3
sage: a = b^17
sage: a.log(b)
17
```

This also works over finite fields:

```
sage: FF = FiniteField(16, "a")
sage: a = FF.gen()
sage: c = a^7
sage: c.log(a)
7
```

10.3 Prime numbers

How do you construct prime numbers in Sage?

The class `Primes` allows for primality testing:

```
sage: 2^(2^12)+1 in Primes()
False
sage: 11 in Primes()
True
```

The usage of `next_prime` is self-explanatory:

```
sage: next_prime(2005)
2011
```

The Pari command `primepi` is used via the command `pari(x).primepi()`. This returns the number of primes $\leq x$, for example:

```
sage: pari(10).primepi()
4
```

Using `primes_first_n` or `primes` one can check that, indeed, there are 4 primes up to 10:

```
sage: primes_first_n(5)
[2, 3, 5, 7, 11]
sage: list(primes(1, 10))
[2, 3, 5, 7]
```

10.4 Divisors

How do you compute the sum of the divisors of an integer in Sage?

Sage uses `divisors(n)` for the number (usually denoted $d(n)$) of divisors of n and `sigma(n, k)` for the sum of the k^{th} powers of the divisors of n (so `divisors(n)` and `sigma(n, 0)` are the same). For example:

```
sage: divisors(28); sum(divisors(28)); 2*28
[1, 2, 4, 7, 14, 28]
56
56
sage: sigma(28,0); sigma(28,1); sigma(28,2)
6
56
1050
```

10.5 Quadratic residues

Try this:

```
sage: Q = quadratic_residues(23); Q
[0, 1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18]
sage: N = [x for x in range(22) if kronecker(x,23)==-1]; N
[5, 7, 10, 11, 14, 15, 17, 19, 20, 21]
```

Q is the set of quadratic residues mod 23 and N is the set of non-residues.

Here is another way to construct these using the `kronecker` command (which is also called the “Legendre symbol”):

```
sage: [x for x in range(22) if kronecker(x,23)==1]
[1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18]
sage: [x for x in range(22) if kronecker(x,23)==-1]
[5, 7, 10, 11, 14, 15, 17, 19, 20, 21]
```

MODULAR FORMS

One of Sage's computational specialities is (the very technical field of) modular forms and can do a lot more than is even suggested in this very brief introduction.

11.1 Cusp forms

How do you compute the dimension of a space of cusp forms using Sage?

To compute the dimension of the space of cusp forms for Gamma use the command `dimension_cusp_forms`. Here is an example from section “Modular forms” in the Tutorial:

```
sage: dimension_cusp_forms (Gamma0 (11) , 2)
1
sage: dimension_cusp_forms (Gamma0 (1) , 12)
1
sage: dimension_cusp_forms (Gamma1 (389) , 2)
6112
```

Related commands: `dimension_new__cusp_forms_gamma0` (for dimensions of newforms), `dimension_modular_forms` (for modular forms), and `dimension_eis` (for Eisenstein series). The syntax is similar - see the Reference Manual for examples.

In future versions of Sage, more related commands will be added.

11.2 Coset representatives

The explicit representation of fundamental domains of arithmetic quotients H/Γ can be determined from the cosets of Γ in $SL_2(\mathbf{Z})$. How are these cosets computed in Sage?

Here is an example of computing the coset representatives of $SL_2(\mathbf{Z})/\Gamma_0(11)$:

```
sage: G = Gamma0 (11) ; G
Congruence Subgroup Gamma0 (11)
sage: list (G.coset_reps ())
[[1 0]
 [0 1],
 [ 0 -1]
 [ 1  0],
 [1 0]
 [1 1],
 [ 0 -1]]
```

```

[ 1  2],
[ 0 -1]
[ 1  3],
[ 0 -1]
[ 1  4],
[ 0 -1]
[ 1  5],
[ 0 -1]
[ 1  6],
[ 0 -1]
[ 1  7],
[ 0 -1]
[ 1  8],
[ 0 -1]
[ 1  9],
[ 0 -1]
[ 1 10]]

```

11.3 Modular symbols and Hecke operators

Next we illustrate computation of Hecke operators on a space of modular symbols of level 1 and weight 12.

```

sage: M = ModularSymbols(1,12)
sage: M.basis()
([X^8*Y^2, (0,0)], [X^9*Y, (0,0)], [X^10, (0,0)])
sage: t2 = M.T(2)
sage: f = t2.charpoly('x'); f
x^3 - 2001*x^2 - 97776*x - 1180224
sage: factor(f)
(x - 2049) * (x + 24)^2
sage: M.T(11).charpoly('x').factor()
(x - 285311670612) * (x - 534612)^2

```

Here t_2 represents the Hecke operator T_2 on the space of Full Modular Symbols for $\Gamma_0(1)$ of weight 12 with sign 0 and dimension 3 over \mathbb{Q} .

```

sage: M = ModularSymbols(Gamma1(6),3,sign=0)
sage: M
Modular Symbols space of dimension 4 for Gamma_1(6) of weight 3 with sign 0
and over Rational Field
sage: M.basis()
([X, (0,5)], [X, (3,2)], [X, (4,5)], [X, (5,4)])
sage: M._compute_hecke_matrix_prime(2).charpoly()
x^4 - 17*x^2 + 16
sage: M.integral_structure()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

```

See the section on modular forms in the Tutorial or the Reference Manual for more examples.

11.4 Genus formulas

Sage can compute the genus of $X_0(N)$, $X_1(N)$, and related curves. Here are some examples of the syntax:

```
sage: dimension_cusp_forms (Gamma0 (22))  
2  
sage: dimension_cusp_forms (Gamma0 (30))  
3  
sage: dimension_cusp_forms (Gamma1 (30))  
9
```

See the code for computing dimensions of spaces of modular forms (in `sage/modular/dims.py`) or the paper by Oesterlé and Cohen {CO} for some details.

ELLIPTIC CURVES

12.1 Conductor

How do you compute the conductor of an elliptic curve (over \mathbb{Q}) in Sage?

Once you define an elliptic curve E in Sage, using the `EllipticCurve` command, the conductor is one of several “methods” associated to E . Here is an example of the syntax (borrowed from section 2.4 “Modular forms” in the tutorial):

```
sage: E = EllipticCurve([1, 2, 3, 4, 5])
sage: E
Elliptic Curve defined by  $y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5$  over
Rational Field
sage: E.conductor()
10351
```

12.2 j -invariant

How do you compute the j -invariant of an elliptic curve in Sage?

Other methods associated to the `EllipticCurve` class are `j_invariant`, `discriminant`, and `weierstrass_model`. Here is an example of their syntax.

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by  $y^2 + y = x^3 - x^2 - 10*x - 20$  over Rational Field
sage: E.j_invariant()
-122023936/161051
sage: E.short_weierstrass_model()
Elliptic Curve defined by  $y^2 = x^3 - 13392*x - 1080432$  over Rational Field
sage: E.discriminant()
-161051
sage: E = EllipticCurve(GF(5), [0, -1, 1, -10, -20])
sage: E.short_weierstrass_model()
Elliptic Curve defined by  $y^2 = x^3 + 3*x + 3$  over Finite Field of size 5
sage: E.j_invariant()
4
```

12.3 The $GF(q)$ -rational points on E

How do you compute the number of points of an elliptic curve over a finite field?

Given an elliptic curve defined over $\mathbb{F} = GF(q)$, Sage can compute its set of \mathbb{F} -rational points

```
sage: E = EllipticCurve(GF(5), [0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by y^2 + y = x^3 + 4*x^2 over Finite Field of size 5
sage: E.points()
[(0 : 0 : 1), (0 : 1 : 0), (0 : 4 : 1), (1 : 0 : 1), (1 : 4 : 1)]
sage: E.cardinality()
5
sage: G = E.abelian_group()
sage: G
Additive abelian group isomorphic to Z/5 embedded in Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 + 4*x^2 over Finite Field of size 5
sage: G.permutation_group()
Permutation Group with generators [(1,2,3,4,5)]
```

12.4 Modular form associated to an elliptic curve over \mathbb{Q}

Let E be a “nice” elliptic curve whose equation has integer coefficients, let N be the conductor of E and, for each n , let a_n be the number appearing in the Hasse-Weil L -function of E . The Taniyama-Shimura conjecture (proven by Wiles) states that there exists a modular form of weight two and level N which is an eigenform under the Hecke operators and has a Fourier series $\sum_{n=0}^{\infty} a_n q^n$. Sage can compute the sequence a_n associated to E . Here is an example.

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.conductor()
11
sage: E.anlist(20)
[0, 1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]
sage: E.analytic_rank()
0
```

NUMBER FIELDS

13.1 Ramification

How do you compute the number fields with given discriminant and ramification in Sage?

Sage can access the Jones database of number fields with bounded ramification and degree less than or equal to 6. It must be installed separately (`database_jones_numfield`).

First load the database:

```
sage: J = JonesDatabase()           # requires optional database
sage: J                             # requires optional database
John Jones's table of number fields with bounded ramification and degree <= 6
```

List the degree and discriminant of all fields in the database that have ramification at most at 2:

```
sage: [(k.degree(), k.disc()) for k in J.unramified_outside([2])] # requires optional database
[(4, -2048), (2, 8), (4, -1024), (1, 1), (4, 256), (2, -4), (4, 2048), (4, 512), (4, 2048), (2, -8),
```

List the discriminants of the fields of degree exactly 2 unramified outside 2:

```
sage: [k.disc() for k in J.unramified_outside([2],2)] # requires optional database
[8, -4, -8]
```

List the discriminants of cubic field in the database ramified exactly at 3 and 5:

```
sage: [k.disc() for k in J.ramified_at([3,5],3)] # requires optional database
[-6075, -6075, -675, -135]
sage: factor(6075)
3^5 * 5^2
sage: factor(675)
3^3 * 5^2
sage: factor(135)
3^3 * 5
```

List all fields in the database ramified at 101:

```
sage: J.ramified_at(101)           # requires optional database
[Number Field in a with defining polynomial x^2 - 101,
Number Field in a with defining polynomial x^4 - x^3 + 13*x^2 - 19*x + 361,
Number Field in a with defining polynomial x^5 - x^4 - 40*x^3 - 93*x^2 - 21*x + 17,
Number Field in a with defining polynomial x^5 + x^4 - 6*x^3 - x^2 + 18*x + 4,
Number Field in a with defining polynomial x^5 + 2*x^4 + 7*x^3 + 4*x^2 + 11*x - 6]
```

13.2 Class numbers

How do you compute the class number of a number field in Sage?

The `class_number` is a method associated to a `QuadraticField` object:

```
sage: K = QuadraticField(29, 'x')
sage: K.class_number()
1
sage: K = QuadraticField(65, 'x')
sage: K.class_number()
2
sage: K = QuadraticField(-11, 'x')
sage: K.class_number()
1
sage: K = QuadraticField(-15, 'x')
sage: K.class_number()
2
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in x with defining
polynomial x^2 + 15
sage: K = QuadraticField(401, 'x')
sage: K.class_group()
Class group of order 5 with structure C5 of Number Field in x with defining
polynomial x^2 - 401
sage: K.class_number()
5
sage: K.discriminant()
401
sage: K = QuadraticField(-479, 'x')
sage: K.class_group()
Class group of order 25 with structure C25 of Number Field in x with defining
polynomial x^2 + 479
sage: K.class_number()
25
sage: K.pari_polynomial()
x^2 + 479
sage: K.degree()
2
```

Here's an example involving a more general type of number field:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: K = NumberField(x^5+10*x+1, 'a')
sage: K
Number Field in a with defining polynomial x^5 + 10*x + 1
sage: K.degree()
5
sage: K.pari_polynomial()
x^5 + 10*x + 1
sage: K.discriminant()
25603125
sage: K.class_group()
Class group of order 1 of Number Field in a with defining
polynomial x^5 + 10*x + 1
sage: K.class_number()
1
```

- See also the link for class numbers at <http://mathworld.wolfram.com/ClassNumber.html> at the Math World site

for tables, formulas, and background information.

- For cyclotomic fields, try:

```
sage: K = CyclotomicField(19)
sage: K.class_number()      # long time
1
```

For further details, see the documentation strings in the `ring/number_field.py` file.

13.3 Integral basis

How do you compute an integral basis of a number field in Sage?

Sage can compute a list of elements of this number field that are a basis for the full ring of integers of a number field.

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: K = NumberField(x^5+10*x+1, 'a')
sage: K.integral_basis()
[1, a, a^2, a^3, a^4]
```

Next we compute the ring of integers of a cubic field in which 2 is an “essential discriminant divisor”, so the ring of integers is not generated by a single element.

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: K = NumberField(x^3 + x^2 - 2*x + 8, 'a')
sage: K.integral_basis()
[1, 1/2*a^2 + 1/2*a, a^2]
```


ALGEBRAIC GEOMETRY

14.1 Point counting on curves

How do you count points on an elliptic curve over a finite field in Sage?

Over prime finite fields, includes both the the baby step giant step method and the SEA (Schoof-Elkies-Atkin) algorithm (implemented in PARI by Christophe Doche and Sylvain Duquesne). An example taken form the Reference manual:

```
sage: E = EllipticCurve(GF(10007), [1, 2, 3, 4, 5])
sage: E.cardinality()
10076
```

The command `E.points()` will return the actual list of rational points.

How do you count points on a plane curve over a finite field? The `rational_points` command produces points by a simple enumeration algorithm. Here is an example of the syntax:

```
sage: x, y, z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: C = Curve(y^2*z^7 - x^9 - x*z^8); C
Projective Curve over Finite Field of size 5 defined by -x^9 + y^2*z^7 - x*z^8
sage: C.rational_points()
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1), (3 : 1 : 1), (3 : 4 : 1)]
sage: C.rational_points(algorithm="bn")
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1), (3 : 1 : 1), (3 : 4 : 1)]
```

The option `algorithm="bn` uses Sage's Singular interface and calls the `brnoeth` package.

Here is another example using Sage's `rational_points` applied to Klein's quartic over $GF(8)$.

```
sage: x, y, z = PolynomialRing(GF(8, 'a'), 3, 'xyz').gens()
sage: f = x^3*y+y^3*z+x*z^3
sage: C = Curve(f); C
Projective Curve over Finite Field in a of size 2^3 defined by x^3*y + y^3*z + x*z^3
sage: C.rational_points()
[(0 : 0 : 1),
 (0 : 1 : 0),
 (1 : 0 : 0),
 (1 : a : 1),
 (1 : a^2 : 1),
 (1 : a^2 + a : 1),
 (a : 1 : 1),
 (a : a^2 : 1),
 (a : a^2 + 1 : 1),
 (a + 1 : a + 1 : 1),
```

```
(a + 1 : a^2 : 1),
(a + 1 : a^2 + a + 1 : 1),
(a^2 : 1 : 1),
(a^2 : a^2 + a : 1),
(a^2 : a^2 + a + 1 : 1),
(a^2 + 1 : a + 1 : 1),
(a^2 + 1 : a^2 + 1 : 1),
(a^2 + 1 : a^2 + a : 1),
(a^2 + a : 1 : 1),
(a^2 + a : a : 1),
(a^2 + a : a + 1 : 1),
(a^2 + a + 1 : a : 1),
(a^2 + a + 1 : a^2 + 1 : 1),
(a^2 + a + 1 : a^2 + a + 1 : 1)]
```

14.1.1 Other methods

- For a plane curve, you can use Singular's `closed_points` command. The input is the vanishing ideal I of the curve X in a ring of 2 variables $F[x, y]$. The `closed_points` command returns a list of prime ideals (each a Gröbner basis), corresponding to the (distinct affine closed) points of $V(I)$. Here's an example:

```
sage: singular_console()
SINGULAR / Development
A Computer Algebra System for Polynomial Computations / version 3-0-1
0<
by: G.-M. Greuel, G. Pfister, H. Schoenemann \ October 2005
FB Mathematik der Universitaet, D-67653 Kaiserslautern \
// ** executing /home/wdj/sagefiles/sage-0.9.4/local/LIB/.singularrc
> LIB "brnoeth.lib";
> ring s = 2, (x,y), lp;
> ideal I = x4+x,y4+y;
> list L = closed_points(I);
> L;
[1]:
  _[1] = y
  _[2] = x
[2]:
  _[1] = y
  _[2] = x+1
[3]:
  _[1] = y
  _[2] = x2+x+1
[4]:
  _[1] = y+1
  _[2] = x
[5]:
  _[1] = y+1
  _[2] = x+1
[6]:
  _[1] = y+1
  _[2] = x2+x+1
[7]:
  _[1] = y2+y+1
  _[2] = x+1
[8]:
  _[1] = y2+y+1
  _[2] = x
```

```
[9]:
  _[1] = y^2+y+1
  _[2] = x+y
[10]:
  _[1] = y^2+y+1
  _[2] = x+y+1
> Auf Wiedersehen.
```

```
sage: singular.lib("brnoeth.lib")
sage: s = singular.ring(2, '(x,y)', 'lp')
sage: I = singular.ideal(['x^4+x, y^4+y'])
sage: L = singular.closed_points(I)
sage: # Here you have all the points :
sage: print L
[1]:
  _[1]=y^2+y+1
  _[2]=x+1
...
```

- Another way to compute rational points is to use Singular's NSplaces command. Here's the Klein quartic over $GF(8)$ done this way:

```
sage: singular.LIB("brnoeth.lib")
sage: s = singular.ring(2, '(x,y)', 'lp')
...
sage: f = singular.poly('x^3y+y^3+x')
...
sage: klein1 = f.Adj_div(); print klein1
[1]:
  [1]:
    // characteristic : 2
  // number of vars : 2
  // block 1 : ordering lp
  //          : names x y
  // block 2 : ordering C
  ...
sage: # define a curve X = {f = 0} over GF(2)
sage: klein2 = singular.NSplaces(3, klein1)
sage: print singular.eval('extcurve(3,%s)'%klein2.name())
Total number of rational places : NrRatPl = 23
...
sage: klein3 = singular.extcurve(3, klein2)
```

Above we defined a curve $X = \{f = 0\}$ over $GF(8)$ in Singular.

```
sage: print klein1
[1]:
  [1]:
    // characteristic : 2
  // number of vars : 2
  // block 1 : ordering lp
  //          : names x y
  // block 2 : ordering C
  [2]:
    // characteristic : 2
  // number of vars : 3
  // block 1 : ordering lp
  //          : names x y z
  // block 2 : ordering C
```

```
[2]:
  4,3
[3]:
  [1]:
    1,1
  [2]:
    1,2
[4]:
  0
[5]:
  [1]:
    [1]:
      // characteristic : 2
// number of vars : 3
// block 1 : ordering ls
//           : names x y t
// block 2 : ordering C
  [2]:
    1,1
sage: print klein1[3]
[1]:
  1,1
[2]:
  1,2
```

For the places of degree 3:

```
sage: print klein2[3]
[1]:
  1,1
[2]:
  1,2
[3]:
  3,1
[4]:
  3,2
[5]:
  3,3
[6]:
  3,4
[7]:
  3,5
[8]:
  3,6
[9]:
  3,7
```

Each point below is a pair: (degree, point index number).

```
sage: print klein3[3]
[1]:
  1,1
[2]:
  1,2
[3]:
  3,1
[4]:
  3,2
[5]:
```

```

    3,3
[6]:
    3,4
[7]:
    3,5
[8]:
    3,6
[9]:
    3,7

```

To actually get the points of $X(GF(8))$:

```

sage: R = klein3[1][5]
sage: R.set_ring()
sage: singular("POINTS;")
[1]:
  [1]:
    0
  [2]:
    1
  [3]:
    0
[2]:
  [1]:
    1
  [2]:
    0
  [3]:
    0
...

```

plus 21 others (omitted). There are a total of 23 rational points.

14.2 Riemann-Roch spaces using Singular

To compute a basis of the Riemann-Roch space of a divisor D on a curve over a field F , one can use Sage's wrapper `riemann_roch_basis` of Singular's implementation of the Brill Noether algorithm. Note that this wrapper currently only works when F is prime and the divisor D is supported on rational points. Below are examples of how to use `riemann_roch_basis` and how to use Singular itself to help an understanding of how the wrapper works.

- Using `riemann_roch_basis`:

```

sage: x, y, z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: f = x^7 + y^7 + z^7
sage: X = Curve(f); pts = X.rational_points()
sage: D = X.divisor([ (3, pts[0]), (-1,pts[1]), (10, pts[5]) ])
sage: X.riemann_roch_basis(D)
[(-2*x + y)/(x + y), (-x + z)/(x + y)]

```

- Using Singular's BrillNoether command (for details see the section Brill-Noether in the Singular online documentation (http://www.singular.uni-kl.de/Manual/html/sing_960.htm and the paper {CF}):

```

sage: singular.LIB('brnoeth.lib')
sage: _ = singular.ring(5, '(x,y)', 'lp')
sage: print singular.eval("list X = Adj_div(-x5+y2+x);")
Computing affine singular points ...
Computing all points at infinity ...

```

```

Computing affine singular places ...
Computing singular places at infinity ...
Computing non-singular places at infinity ...
Adjunction divisor computed successfully

The genus of the curve is 2
sage: print singular.eval("X = NSplaces(1,X);")
Computing non-singular affine places of degree 1 ...
sage: print singular("X[3];")
[1]:
  1,1
[2]:
  1,2
[3]:
  1,3
[4]:
  1,4
[5]:
  1,5
[6]:
  1,6

```

The first integer of each pair in the above list is the degree d of a point. The second integer is the index of this point in the list POINTS of the ring $X[5][d][1]$. Note that the order of this latter list is different every time the algorithm is run, e.g. 1, 1 in the above list refers to a different rational point each time. A divisor is given by defining a list G of integers of the same length as $X[3]$ such that if the k -th entry of $X[3]$ is d , i , then the k -th entry of G is the multiplicity of the divisor at the i -th point in the list POINTS of the ring $X[5][d][1]$. Let us proceed by defining a “random” divisor of degree 12 and computing a basis of its Riemann-Roch space:

```

sage: singular.eval("intvec G = 4,4,4,0,0,0;")
'intvec G = 4,4,4,0,0,0;'
sage: singular.eval("def R = X[1][2];")
'def R = X[1][2];'
sage: singular.eval("setring R;")
'setring R;'
sage: print singular.eval("list LG = BrillNoether(G,X);")
Forms of degree 6 :
28

Vector basis successfully computed

```

14.2.1 AG codes

Sage can compute an AG code $C = C_X(D, E)$ by calling Singular’s BrillNoether to compute a basis of the Riemann Roch space $L(D) = L_X(D)$. In addition to the curve X and the divisor D , you must also specify the evaluation divisor E .

Note that this section has not been updated since the wrapper `riemann_roch_basis` has been fixed. See above for how to properly define a divisor for Singular’s BrillNoether command.

Here’s an example, one which computes a generator matrix of an associated AG code. This time we use Singular’s `AGCode_L` command.

```

sage: singular.LIB('brnoeth.lib')
sage: singular.eval("ring s = 2, (x,y), lp;")
'ring s = 2, (x,y), lp;'
sage: print singular.eval("list HC = Adj_div(x3+y2+y);")

```

```

Computing affine singular points ...
Computing all points at infinity ...
Computing affine singular places ...
Computing singular places at infinity ...
Computing non-singular places at infinity ...
Adjunction divisor computed successfully

The genus of the curve is 1
sage: print singular.eval("list HC1 = NSplaces(1..2,HC);")
Computing non-singular affine places of degree 1 ...
Computing non-singular affine places of degree 2 ...
sage: print singular.eval("HC = extcurve(2,HC1);")
Total number of rational places : NrRatPl = 9

```

We set the following to junk to discard the output:

```

sage: junk = singular.eval("intvec G = 5;")           # the rational divisor G = 5*HC[3][1]
sage: junk = singular.eval("def R = HC[1][2];")
sage: singular.eval("setring R;")
'setring R;'

```

The vector G represents the divisor “5 times the point at infinity”.

Next, we compute the Riemann-Roch space.

```

sage: print singular.eval("BrillNoether(G,HC);")
Forms of degree 3 :
10

```

Vector basis successfully computed

```

[1]:
  _[1]=x
  _[2]=z
[2]:
  _[1]=y
  _[2]=z
[3]:
  _[1]=1
  _[2]=1
[4]:
  _[1]=y2+yz
  _[2]=xz
[5]:
  _[1]=y3+y2z
  _[2]=x2z

```

That was the basis of the Riemann-Roch space, where each pair of functions represents the quotient (first function divided by second function). Each of these basis elements get evaluated at certain points to construct the generator matrix of the code. We next construct the points.

```

sage: singular.eval("def R = HC[1][5];")
'// ** redefining R **'
sage: singular.eval("setring R;")
''
sage: print singular.eval("POINTS;")
[1]:
  [1]:
    0

```

```

[2]:
  1
[3]:
  0
[2]:
[1]:
  0
[2]:
  1
[3]:
  1
[3]:
[1]:
  0
[2]:
  0
[3]:
  1
[4]:
[1]:
  (a+1)
[2]:
  (a)
[3]:
  1
...

```

plus 5 more, for a total of 9 rational points on the curve. We define our “evaluation divisor” D using a subset of these points (all but the first):

```

sage: singular.eval("def ER = HC[1][4];")
''
sage: singular.eval("setring ER;")
''
sage: # D = sum of the rational places no. 2..9 over F_4
sage: singular.eval("intvec D = 2..9;")
''
sage: # let us construct the corresponding evaluation AG code :
sage: print singular.eval("matrix C = AGcode_L(G,D,HC);")
Forms of degree 3 :
10

```

Vector basis successfully computed

```

sage: # here is a linear code of type [8,5,> = 3] over F_4
sage: print singular.eval("print(C);")
0,0,(a+1),(a), 1, 1, (a), (a+1),
1,0,(a), (a+1),(a),(a+1),(a), (a+1),
1,1,1, 1, 1, 1, 1, 1,
0,0,(a), (a+1),1, 1, (a+1),(a),
0,0,1, 1, (a),(a+1),(a+1),(a)

```

This is, finally, our desired generator matrix, where a represents a generator of the field extension of degree 2 over the base field $GF(2)$.

Can this be “wrapped”?

INTERFACE ISSUES

15.1 Background jobs

Yes, a Sage job can be run in the background on a UNIX system. The canonical thing to do is type

```
nohup sage < command_file > output_file &
```

The advantage of `nohup` is that Sage will continue running after you log out.

Currently Sage will appear as “sage-ipython” or “python” in the output of the (unix) `top` command, but in future versions of Sage it will appear as `sage`.

15.2 Referencing Sage

To reference Sage, please add the following to your bibliography:

```
\bibitem[Sage]{sage}
Stein, William, \emph{Sage: {O}pen {S}ource {M}athematical {S}oftware
({V}ersion 2.10.2)}, The Sage~Group, 2008, {\tt http://www.sagemath.org}.
```

Here is the bibtex entry:

```
@manual{sage,
  Key = {Sage},
  Author = {William Stein},
  Organization = {The Sage~Group},
  Title = {{Sage}: {O}pen {S}ource {M}athematical {S}oftware ({V}ersion 2.10.2)},
  Note= {{\tt http://www.sagemath.org}},
  Year = 2008
}
```

If you happen to use the Sage interface to PARI, GAP or Singular, you should definitely reference them as well. Likewise, if you use code that is implemented using PARI, GAP, or Singular, reference the corresponding system (you can often tell from the documentation if PARI, GAP, or Singular is used in the implementation of a function).

For PARI, you may use

```
@manual{PARI2,
  organization = "{The PARI~Group}",
  title        = "{PARI/GP, version {\tt 2.1.5}}",
  year         = 2004,
  address      = "Bordeaux",
```

```
    note          = "available from \url{http://pari.math.u-bordeaux.fr/}"
  }
```

or

```
\bibitem{PARI2} PARI/GP, version {\tt 2.1.5}, Bordeaux, 2004,
\url{http://pari.math.u-bordeaux.fr/}.
```

(replace the version number by the one you used).

For GAP, you may use

```
[GAP04] The GAP Group, GAP -- Groups, Algorithms, and Programming,
Version 4.4; 2005. (http://www.gap-system.org)
```

or

```
@manual{GAP4,
  key          = "GAP",
  organization = "The GAP~Group",
  title        = "{GAP -- Groups, Algorithms, and Programming,
                  Version 4.4}",
  year         = 2005,
  note         = "{\tt http://www.gap-system.org}",
  keywords     = "groups; *; gap; manual"}

\bibitem[GAP]{GAP4}
  The GAP~Group, \emph{GAP -- Groups, Algorithms, and Programming, Version 4.4}; 2005,
  {\tt http://www.gap-system.org}.
```

For Singular, you may use

```
[GPS05] G.-M. Greuel, G. Pfister, and H. Sch\"onemann.
{\sc Singular} 3.0. A Computer Algebra System for Polynomial
Computations. Centre for Computer Algebra, University of
Kaiserslautern (2005). {\tt http://www.singular.uni-kl.de}.
```

or

```
@TechReport{GPS05,
  author = {G.-M. Greuel and G. Pfister and H. Sch\"onemann},
  title =  {{{\sc Singular} 3.0},
  type =   {{A Computer Algebra System for Polynomial Computations}},
  institution = {Centre for Computer Algebra},
  address = {University of Kaiserslautern},
  year =     {2005},
  note =     {{{\tt http://www.singular.uni-kl.de}},
}
}
```

or

```
\bibitem[GPS05]{GPS05}
G.-M.~Greuel, G.~Pfister, and H.~Sch\"onemann.
\newblock {{{\sc Singular} 3.0}. A Computer Algebra System for Polynomial Computations.
\newblock Centre for Computer Algebra, University of Kaiserslautern (2005).
\newblock {\tt http://www.singular.uni-kl.de}.
```

15.3 Logging your Sage session

Yes you can log your sessions.

- (a) Modify line 186 of the `.ipythonrc` file (or open `.ipythonrc` into an editor and search for “logfile”). This will only log your input lines, not the output.
- (b) You can also write the output to a file, by running Sage in the background (*Background jobs*).
- (c) Start in a KDE konsole (this only work in linux). Go to Settings → History ... and select unlimited. Start your session. When ready, go to edit → save history as

Some interfaces (such as the interface to Singular or that to GAP) allow you to create a log file. For Singular, there is a logfile option (in `singular.py`). In GAP, use the command `LogTo`.

15.4 LaTeX conversion

Yes, you can output some of your results into LaTeX.

```
sage: M = MatrixSpace(RealField(), 3, 3)
sage: A = M([1, 2, 3, 4, 5, 6, 7, 8, 9])
sage: print latex(A)
\left(\begin{array}{rrr}
1.000000000000000 & 2.000000000000000 & 3.000000000000000 \\
4.000000000000000 & 5.000000000000000 & 6.000000000000000 \\
7.000000000000000 & 8.000000000000000 & 9.000000000000000
\end{array}\right)
sage: view(A)
```

At this point a dvi preview should automatically be called to display in a separate window the LaTeX output produced.

LaTeX previewing for multivariate polynomials and rational functions is also available:

```
sage: x = PolynomialRing(QQ, 3, 'x').gens()
sage: f = x[0] + x[1] - 2*x[1]*x[2]
sage: h = f / (x[1] + x[2])
sage: print latex(h)
\frac{-2 x_{1} x_{2} + x_{0} + x_{1}}{x_{1} + x_{2}}
```

15.5 Sage and other computer algebra systems

If `foo` is a Pari, GAP (without ending semicolon), Singular, Maxima command, resp., enter `gp("foo")` for Pari, `gap.eval("foo")` `singular.eval("foo")`, `maxima("foo")`, resp.. These programs merely send the command string to the external program, execute it, and read the result back into Sage. Therefore, these will not work if the external program is not installed and in your PATH.

15.6 Command-line Sage help

If you know only part of the name of a Sage command and want to know where it occurs in Sage, a new option for 0.10.11 has been added to make it easier to hunt it down. Just type `sage -grep <string>` to find all occurrences of `<string>` in the Sage source code. For example,

```
was@form:~/s/local/bin$ sage -grep berlekamp_massey
matrix/all.py:from berlekamp_massey import berlekamp_massey
matrix/berlekamp_massey.py:def berlekamp_massey(a):
matrix/matrix.py:import berlekamp_massey
matrix/matrix.py:        g =
berlekamp_massey.berlekamp_massey(cols[i].list())
```

Type `help(foo)` or `foo??` for help and `foo.[tab]` for searching of Sage commands. Type `help()` for Python commands.

For example

```
help(Matrix)
```

returns

Help on function Matrix in module sage.matrix.constructor:

```
Matrix(R, nrows, ncols, entries = 0, sparse = False)
    Create a matrix.
```

INPUT:

```
R -- ring
nrows -- int; number of rows
ncols -- int; number of columns
entries -- list; entries of the matrix
sparse -- bool (default: False); whether or not to store matrices as sparse
```

OUTPUT:

```
a matrix
```

EXAMPLES:

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

```
sage: Matrix(FiniteField(5), 2, 3, range(6))
[0 1 2]
[3 4 0]
```

```
sage: Matrix(IntegerRing(), 10, 10, range(100)).parent()
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
```

```
sage: Matrix(IntegerRing(), 10, 10, range(100), sparse = True).parent()
Full MatrixSpace of 10 by 10 sparse matrices over Integer Ring
```

in a new screen. Type `q` to return to the Sage screen.

15.7 Reading and importing files into Sage

A file imported into Sage must end in `.py`, e.g., `foo.py` and contain legal Python syntax. For a simple example see *Permutation groups* with the Rubik's cube group example above.

Another way to read a file in is to use the `load` or `attach` command. Create a file called `example.sage` (located in the home directory of Sage) with the following content:

```
print "Hello World"
print 2^3
```

Read in and execute `example.sage` file using the `load` command.

```
sage: load "example.sage"
Hello World
8
```

You can also attach a Sage file to a running session:

```
sage: attach "example.sage"
Hello World
8
```

Now if you change `example.sage` and enter one blank line into Sage, then the contents of `example.sage` will be automatically reloaded into Sage:

```
sage: !emacs example.sage&      #change 2^3 to 2^4
sage:                          #hit return
*****
                Reloading 'example.sage'
*****
Hello World
16
```

15.8 Installation for the impatient

We shall explain the basic steps for installing the most recent version of Sage (which is the “source” version, not the “binary”).

1. Download `sage-*.tar` (where `*` denotes the version number) from the website and save into a directory, say `HOME`. Type `tar zxvf sage-*.tar` in `HOME`.
2. `cd sage-*` (we call this `SAGE_ROOT`) and type `make`. Now be patient because this process make take 2 hours or so.
3. Optional: When the compilation is finished, type on the command line in the Sage home directory:

```
./sage -i database_jones_numfield
./sage -i database_gap-4.4.8
./sage -i database_cremona_ellcurve-2005.11.03
./sage -i gap_packages-4.4.8_1
```

This last package loads the GAP GPL'd packages `braid`, `ctbllib`, `DESIGN`, `FactInt`, `GAPDoc`, `GRAPE`, `LAGUNA`, `SONATA 2.3`, and `TORIC`. It also compiles (automatically) the C programs in `GUAVA` and `GRAPE`.

Other optional packages to install are at <http://modular.math.washington.edu/sage/packages/optional/>.

Another way: download packages from <http://sage.scipy.org/sage/packages/optional/> and save to the directory `SAGE_ROOT`. Type

```
/sage -i sage-package.spkg
```

for each `sage-package` you download (use `sage -f` if you are reinstalling.) This might be useful if you have a CD of these packages but no (or a very slow) internet connection.

4. If you want to build the documentation, `cd devel/doc` and type `./rebuild`. This requires having `latex` and `latex2html` installed.

15.9 Python language program code for Sage commands

Let's say you want to know what the Python program is for the Sage command to compute the center of a permutation group. Use Sage's help interface to find the file name:

```
sage: PermutationGroup.center?
Type:          instancemethod
Base Class:    <type 'instancemethod'>
String Form:   <unbound method PermutationGroup.center>
Namespace:    Interactive
File:         /home/wdj/sage/local/lib/python2.4/site-packages/sage/groups/permgrou.py
Definition:   PermutationGroup.center(self)
```

Now you know that the command is located in the `permgrou.py` file and you know the directory to look for that Python module. You can use an editor to read the code itself.

15.10 "Special functions" in Sage

Sage has several special functions:

- Bessel functions and Airy functions
- spherical harmonic functions
- spherical Bessel functions (of the 1st and 2nd kind)
- spherical Hankel functions (of the 1st and 2nd kind)
- Jacobi elliptic functions
- complete/incomplete elliptic integrals
- hyperbolic trig functions (for completeness, since they are special cases of elliptic functions)

and orthogonal polynomials

- `chebyshev_T(n, x)`, `chebyshev_U(n, x)` - the Chebyshev polynomial of the first, second kind for integers $n > -1$.
- `laguerre(n, x)`, `gen_laguerre(n, a, x)` - the (generalized) Laguerre poly. for $n > -1$.
- `legendre_P(n, x)`, `legendre_Q(n, x)`, `gen_legendre_P(n, x)`, `gen_legendre_Q(n, x)` - the (generalized) Legendre function of the first, second kind for integers $n > -1$.
- `hermite(n, x)` - the Hermite poly. for integers $n > -1$.
- `jacobi_P(n, a, b, x)` - the Jacobi polynomial for integers $n > -1$ and a and b symbolic or $a > -1$ and $b > -1$.
- `ultraspherical(n, a, x)` - the ultraspherical polynomials for integers $n > -1$. The ultraspherical polynomials are also known as Gegenbauer polynomials.

In Sage, these are restricted to numerical evaluation and plotting but via maxima, some symbolic manipulation is allowed:

```
sage: maxima.eval("f:bessel_y (v, w)")
'bessel_y(v, w)'
sage: maxima.eval("diff(f,w)")
'(bessel_y(v-1,w)-bessel_y(v+1,w))/2'
sage: maxima.eval("diff (jacobi_sn (u, m), u)")
'jacobi_cn(u,m)*jacobi_dn(u,m)'
sage: jsn = lambda x: jacobi("sn",x,1)
```

```
sage: P = plot(jsn,0,1, plot_points=20); Q = plot(lambda x:bessel_Y(1, x), 1/2,1)
sage: show(P)
sage: show(Q)
```

In addition to `maxima`, `pari` and `octave` also have special functions (in fact, some of `pari`'s special functions are wrapped in Sage).

Here's an example using Sage's interface (located in `sage/interfaces/octave.py`) with `octave` (<http://www.octave.org/doc/index.html>).

```
sage: octave("atanh(1.1)")    ## requires optional octave
(1.52226, -1.5708)
```

Here's an example using Sage's interface to `pari`'s special functions.

```
sage: pari('2+I').besselk(3)
0.0455907718407551 + 0.0289192946582081*I
sage: pari('2').besselk(3)
0.0615104584717420
```

The last command can also be executed using the command

```
sage: bessel_K(3,2)
0.647385390948634
sage: bessel_K(3,2,prec=100)
0.64738539094863415315923557097
```

15.11 What is Sage?

Sage is a framework for number theory, algebra, and geometry computation that is initially being designed for computing with elliptic curves and modular forms. The long-term goal is to make it much more generally useful for algebra, geometry, and number theory. It is open source and freely available under the terms of the GPL. The section titles in the reference manual gives a rough idea of the topics covered in Sage.

15.11.1 History of Sage

Sage was started by William Stein while at Harvard University in the Fall of 2004, with version 0.1 released in January of 2005. That version included Pari, but not GAP or Singular. Version 0.2 was released in March, version 0.3 in April, version 0.4 in July. During this time, support for Cremona's database, multivariate polynomials and large finite fields was added. Also, more documentation was written. Version 0.5 beta was released in August, version 0.6 beta in September, and version 0.7 later that month. During this time, more support for vector spaces, rings, modular symbols, and windows users was added. As of 0.8, released in October 2005, Sage contained the full distribution of GAP, though some of the GAP databases have to be added separately, and Singular. Adding Singular was not easy, due to the difficulty of compiling Singular from source. Version 0.9 was released in November. This version went through 34 releases! As of version 0.9.34 (definitely by version 0.10.0), Maxima and clisp were included with Sage. Version 0.10.0 was released January 12, 2006. The release of Sage 1.0 was made early February, 2006. As of February 2008, the latest release is 2.10.2.

Many people have contributed significant code and other expertise, such as assistance in compiling on various OS's. Generally code authors are acknowledged in the AUTHOR section of the Python docstring of their file and the credits section of the Sage website.

CONTRIBUTIONS TO THIS DOCUMENT

Besides William Stein, contributions to this part of the documentation were made by Gary Zablackis.

[CF] {CF} A. Campillo and J. I. Farran, Symbolic Hamburger-Noether expressions of plane curves and applications to AG codes', *Math. Comp.*, vol 71(2002)1759-1780. <http://www.ams.org/mcom/2002-71-240/S0025-5718-01-01390-4/home.html>

[CO] {CO} H. Cohen, J. Oesterlé, Dimensions des espaces de formes modulaires, p. 69-78 in *Modular functions in one variable VI. Lecture Notes in Math.* 627, Springer-Verlag, New York, 1977.

[GAP] {GAP4} The GAP Group, *GAP - Groups, Algorithms, and Programming, Version 4.4*; 2005, <http://www.gap-system.org>

[G] {G} Solomon Golomb, *Shift register sequences*, Aegean Park Press, Laguna Hills, Ca, 1967

[Sing] {GPS05} G.-M. Greuel, G. Pfister, and H. Schönemann. *Singular 3.0. A Computer Algebra System for Polynomial Computations*. Centre for Computer Algebra, University of Kaiserslautern (2005). <http://www.singular.uni-kl.de>

[Pari] {PARI2} PARI/GP, version 2.1.5, Bordeaux, 2004, <http://pari.math.u-bordeaux.fr/>

[M] {M} James L. Massey, Shift-Register Synthesis and BCH Decoding, *IEEE Trans. on Information Theory*, vol. 15(1), pp. 122-127, Jan 1969.

[SAGE] {SJ} William Stein, David Joyner, *SAGE: System for Algebra and Geometry Experimentation*, *Comm. Computer Algebra* 39(2005)61-64. (*SIGSAM Bull.* June 2005) <http://sagemath.org/>
<http://sage.sourceforge.net/>

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

A

- adjacency matrix
 - graph, 31
- algebraic-geometric
 - codes, 64
- attach into Sage, 71

B

- background, running Sage in, 67
- Brauer
 - character, 35

C

- calculus
 - critical points, 4
 - differentiation, 3
 - integration, 4
- center
 - groups, 17
- character
 - Brauer, 35
 - modular representation, 35
- characteristic polynomial
 - matrix, 23
- check matrix
 - codes, 27
- ciphers
 - connection polynomial, 29
- class_number
 - number field, 55
- codes
 - algebraic-geometric, 64
 - check matrix, 27
 - dual, 27
 - generator matrix, 27
 - Golay, 27
- conjugacy classes
 - group, 16
- connection polynomial
 - ciphers, 29
- cosets of Γ_0 , 49

- critical points
 - calculus, 4
- cyclotomic
 - number field, 57

D

- database
 - number field, 55
- differentiation
 - calculus, 3
- discrete logs, 47
- discriminant
 - number field, 55
- dual
 - codes, 27

E

- eigenvalues, 21
- eigenvectors, 21
- elliptic curve
 - modular form, 54
 - point counting, 59
- elliptic curves, 51, 53
- evaluation
 - polynomial, 43

F

- factorization
 - polynomial, 42
- Frobenius normal form, 21

G

- GAP
 - referencing, 68
- gcd
 - polynomial, 42
- generator matrix
 - codes, 27
- Golay
 - codes, 27
- graph

- adjacency matrix, 31
- group
 - conjugacy classes, 16
 - normal subgroups, 17
 - permutation, 15
 - Rubik's cube, 15
- groups
 - center, 17

H

- Hecke operators, 50
- help in Sage, 69
- Hermite normal form, 21
- history
 - Sage, 73

I

- importing into Sage, 70
- installation of packages, 71
- installation of Sage, 71
- integral basis
 - number field, 57
- integration
 - calculus, 4

L

- LaTeX output, 69
- linear equations
 - solve, 24
- load into Sage, 71
- logging Sage, 68

M

- matrix
 - characteristic polynomial, 23
 - ring, 37
- modular form
 - elliptic curve, 54
- modular forms, 48
- modular representation
 - character, 35
- modular symbols, 50

N

- normal subgroups
 - group, 17
- number field
 - class_number, 55
 - cyclotomic, 57
 - database, 55
 - discriminant, 55
 - integral basis, 57

P

- p-adics, 38
- PARI
 - referencing, 67
- permutation
 - group, 15
- plot
 - a function, 12
 - a parametric curve, 12
 - curve using surf, 10
- point counting
 - elliptic curve, 59
- polynomial
 - evaluation, 43
 - factorization, 42
 - gcd, 42
 - powers, 41
 - quotient ring, 38
 - ring, 37
 - roots, 43
 - symbolic manipulation, 44
- power series, 4
- powers
 - polynomial, 41
- Python and Sage, 71

Q

- quadratic residues, 48
- quotient ring
 - polynomial, 38

R

- rational canonical form, 21
- referencing
 - GAP, 68
 - PARI, 67
 - Sage, 67
 - Singular, 68
- Riemann-Roch space, 63, 65
- ring
 - matrix, 37
 - polynomial, 37
- roots
 - polynomial, 43
- Rubik's cube
 - group, 15

S

- Sage
 - history, 73
 - referencing, 67
- Singular
 - referencing, 68

Smith normal form, 21

solve

 linear equations, 24

special functions in Sage, 72

symbolic manipulation

 polynomial, 44

T

Taylor series, 4