MA1s11

Computer Algebra

MA1S1

Tristan McLoughlin

References:

http://www.sagenb.org

http://www.sagemath.org/doc/tutorial/introduction.html

http://linear.ups.edu/download/fcla-3.11-sage-5.12-primer.pdf

If you can't read this please move toward the front!

Introduction

We will follow very closely and take most of our examples from the online sage tutorial and the

Sage for Linear Algebra, A Supplement to A First Course in Linear Algebra, by Robert A. Beezer

listed above. See those sources for further details and examples.

Sage is Python based so if you know anything about Python it will help you with Sage, however it's not necessary at all. Enter commands into a *compute cell* and then use the evaluate button or SHIFT-RETURN. For example:

```
2+6
```

```
8
```

You can see that we get the correct result.

Sage uses $=$ for assignment. It uses $==$, $\leq$, $\geq$, $<$ and $>$ for comparison.

```
c=10
```

```
a=7
```

Note that even when there is no output Sage has performed the assignement.

```
b=a
```

```
a==b
```

```
True
```

```
a<=b
```
```
    True
```
```
a>c
```
```
    False
```

Consider the following cell:

```
b=b+10
```

Note that again there is no output but something has happened. The value assigned to b has increased by 10. In fact each time we evaluate this cell 10 will be added to the previously stored value. If we now create a cell which does produce an output of b we can see this.

```
b
```
```
    27
```
```
2**5 #** means exponent
```
```
    32
```
```
2^3 # ^ gives the same as **
```
```
    8
```

We can add comments which the evaluate ignores by # ..... We can also add this type of text cells by holding the SHIFT key while creating a compute cell. Some other operations are:

```
10%3 # for integer arguments this mod i.e. gives the remainder after
dividing 10 by 3
```
```
    1
```
```
10//3 # for integer arguments this gives the quotient
```
```
    3
```

The order in which operations are carried out can be important. See

http://www.sagemath.org/doc/tutorial/appendix.html#section-precedence

```
3^2*4 + 2%5
```
```
    38
```

Can force the order by using brackets.

```
3^(2*4) + 2%5
```
```
    6563
```
```
3^(2*4 + 2)%5
```
```
    4
```
```
3^(2*4 + 2%5)
```
```
    59049
```
```
((3^2)*4) + (2%5)
```
```
    38
```

Sage has many standard functions

```
factorial(4)
```

```
    24
```

```
factorial(301)
```

```
    9212331117714863144664650885047785744596523005624667682444979911910 2\
    4364861917771071201027181657495783316835144992196370281434362354599 4\
    9422279721299999342512208868556011858335806411676948651905233131045 5\
    7242795554340630456089077555410892657751109794706183593699037725625 2\
    0210202137329025175537294741851587322812858600194003982650386368946 0\
    9948101518542666555738140487271373780036453646943966310833635567915 2\
    0110804482395647939340402799702704855430360391496776632374361062371 5\
    8051802378811522644765756328226783230030267320765286025481259319296 0\
    0000000000000000000000000000000000000000000000000000000000000000000 0\
    00000
```

```
sin(3.2)
```

```
    -0.0583741434275801
```

```
sin(pi/4)
```

```
    1/2*sqrt(2)
```

Some functions are evaluated exactly, to get a numerical value one can use the function n() also called numercial_approx(). The default precision is 56 bits.

```
n(sin(pi/4))
```

```
    0.707106781186548
```

but this can be modified by adding an optional argument to the function prec=...

```
n(sin(pi/4),prec=200)
```

```
    0.70710678118654752440084436210484903928483593768847403658834
```

A potential source of confusion in Python and so in Sage is that an integer literal that begins with a zero is treated as an octal number, i.e.,

```
011
```

```
    9
```

Other functions can act on arguments in the fashion .function()

```
1914.factor()
```

```
    2 * 3 * 11 * 29
```

Getting Help

Sage has extensive built-in documentation, which can be accessed by typing the name of a function followed by a question mark:

```
tan?
```

File: /Applications/Sage-5.11-OSX-64bit-10.8.app/Contents/Resources/sage/local/lib/python2.7/site-packages/sage/functions/trig.py

Type: <class ‘sage.functions.trig.Function_tan’>

Definition: tan(coerce=True, hold=False, dont_call_method_on_arg=False, *args)

Docstring:

The tangent function

EXAMPLES:

```
sage: tan(pi)
0
sage: tan(3.1415)
-0.0000926535900581913
sage: tan(3.1415/4)
0.999953674278156
sage: tan(pi/4)
1
sage: tan(1/2)
tan(1/2)
sage: RR(tan(1/2))
0.546302489843790
```

We can prevent evaluation using the `hold` parameter:

```
sage: tan(pi/4,hold=True)
tan(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = tan(pi/4,hold=True); a.simplify()
1
```

TESTS:

```
sage: conjugate(tan(x))
tan(conjugate(x))
```

Multiple commands can be put on a single line separated by a semi-colon

```
b=4*3;b
```

```
    20
```

Matrices

Matrices are basic objects which we have seen before. Lets again look at an example:

```
A = matrix(QQ, 2, 3, [[1, 2, 3], [4, 5, 6]]);A
```

```
    [1 2 3]
    [4 5 6]
```

QQ is the set of all rational numbers (fractions with an integer numerator and denominator), 2 is the number of rows, 3 is the number of columns. Sage understands a list of items as delimited by brackets ([,]) and the items in the list can again be lists themselves. So [[1, 2, 3], [4, 5, 6]] is a list of lists, and in this context the inner lists are rows of the matrix.

Sage knows a wide variety of sets of numbers in addition to rational numbers. These are known as “rings” or “fields” to mathematicians, but we called them “number systems”.

Examples include: ZZ is the integers $\mathbb{Z}$, QQ is the rationals $\mathbb{Q}$, RR is the real numbers, $\mathbb{R}$, though of course Sage can only work to finite precision, and CC is the complex numbers, $\mathbb{C}$, with some reasonable precision

There are various shortcuts you can employ when creating a matrix. For example, Sage is able to infer the size of the matrix from the lists of entries.

```
B = matrix(QQ, [[1, 2, 3], [4, 5, 6],[7,8,9]]);B
```

```
[1 2 3]
[4 5 6]
[7 8 9]
```

Note that Sage nearly always starts counting from zero, like many computer scientists, but unlike sensible people and mathematicians. So we can find the entries of a matrix as follows

```
B[1,1]
```

```
5
```

but not this would normally be called the (2,3) element of the matrix. While

```
B[0,0]
```

```
1
```

Sage also has a function vector(), which is similar in many regards to a 1 row matrix but is delimited by round brackets ( )

```
v=vector(QQ,3,[1,2,3]);v
```

```
(1, 2, 3)
```

We can easily multiply matrices and vectors if they have the right shape.

```
A*A
```

```
Traceback (click to the left of this block for traceback)
...
ArithmeticError: self must be a square matrix
```

```
B*B
```

```
[ 30  36  42]
[ 66  81  96]
[102 126 150]
```

```
B*v
```

```
(14, 32, 50)
```

```
A*v
```

```
(14, 32)
```

```
v*A
```

```
Traceback (click to the left of this block for traceback)
...
```

```
NameError: name 'v' is not defined
```

We can find the shape of matrix by using nrows() and ncols()

```
A.nrows(),A.ncols()
```

```
(2, 3)
```

so A is a 2×3 matrix, and the length of vector by using degree()

```
v.degree()
```

```
3
```

So we can see that A times v, where v is on the right and interpreted as a column vector, a 3×1 matrix, multiplication makes sense. While when it is on the left and so interpreted and a row vector, a 1×3, multiplication doesn't make sense.

Solving equations

Sage can solve equations using the solve function. First one must specify some variables using the var() function; then the arguments to the solve function are an equation (or a system of equations), together with the variables for which to solve:

```
x=var('x');
solve(x+2==3,x)
```

```
[x == 1]
```

We don't have to consider only linear equations:

```
solve(x^2+2*x-3==0,x)
```

```
[x == -3, x == 1]
```

We can also consider a system of linear equations in a number of variables:

```
var('x1,x2,x3')
```

```
(x1, x2, x3)
```

```
eq1=   x1+2*x2+3*x3==1;
eq2= 2*x1+3*x2+  x3==2;
eq3= 2*x1+  x2+  x3==1;
```

```
solve([eq1,eq2,eq3],x1,x2,x3)
```

```
[[x1 == (3/10), x2 == (1/2), x3 == (-1/10)]]
```

These equations are being solved symbolically and so exactly but Sage is capabable of using numerical, and approximate, methods for example with find_root function, however we won't consider these here.

We can of course write down the coefficient matrix for this system of linear equations, write the variables as a vector of variables and the right-hand side of the equation also as a vector.

```
A=Matrix(QQ,[[1,2,3],[2,3,1],[2,1,1]]);
```

```
x=vector([x1,x2,x3]);
b=vector([1,2,1]);
```

We can then see then linear equations as a matrix equation, $Ax - b = 0$, with

```
A*x-b
```

```
    (x1 + 2*x2 + 3*x3 - 1, 2*x1 + 3*x2 + x3 - 2, 2*x1 + x2 + x3 - 1)
```

Sage can in fact solve such matrix equations, where the unknowns are a vector (or matrix) on the right of A.

```
A.solve_right(b)
```

```
    (3/10, 1/2, -1/10)
```

Perhaps obviously .solve_left(b) will solve the equation $xA = b$.

We can also follow more directly our methods in class and form an augmented matrix.

```
B1=A.augment(b);B1
```

```
    [1 2 3 1]
    [2 3 1 2]
    [2 1 1 1]
```

or even include the markers of the last column

```
B2=A.augment(b,w subdivide=True);B2
```

```
    [1 2 3|1]
    [2 3 1|2]
    [2 1 1|1]
```

We can perform row operations (remember Sage's method of counting from zero)

```
D = B2.with_swapped_rows(0,1);B2
```

```
    [1 2 3|1]
    [2 3 1|2]
    [2 1 1|1]
```

the command with "with" creates a new matrix with the rows swapped while the command swap_rows changes the matrix it is acting on.

```
B2.swap_rows(0,1);B2
```

```
    [2 3 1|2]
    [1 2 3|1]
    [2 1 1|1]
```

We can rescale rows by constants

```
B2.rescale_row(0, 1/2);B2
```

```
    [  1 3/2 1/2|  1]
    [  1   2   3|  1]
    [  2   1   1|  1]
```

and add multiples of rows to other rows

```
B2.add_multiple_of_row(1, 0, -1);B2
```

```
[   1 3/2 1/2|   1]
[   0 1/2 5/2|   0]
[   2   1   1|   1]
```

```
B2.add_multiple_of_row(2, 0, -2);B2
```

```
[   1 3/2 1/2|   1]
[   0 1/2 5/2|   0]
[   0  -2   0|  -1]
```

While we could of course continue in this fashion and repeat all the steps of the Gauss-Jordon method, Sage has a built in function to find the reduced row echelon form of a matrix.

```
B1.rref()
```

```
[     1      0      0   3/10]
[     0      1      0    1/2]
[     0      0      1  -1/10]
```

or

```
B2.rref()
```

```
[     1      0      0|  3/10]
[     0      1      0|   1/2]
[     0      0      1| -1/10]
```

or equivalently

```
B2.echelon_form()
```

```
[     1      0      0|  3/10]
[     0      1      0|   1/2]
[     0      0      1| -1/10]
```

There were some special matrices we mentioned in class. Sage has most of these stored automatically, for example the identity matrix is given by

```
id5 = identity_matrix(QQ, 5);id5
```

```
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

All of the matrix operations we discussed are also built into Sage

```
A
```

```
[1 2 3]
[2 3 1]
[2 1 1]
```

```
A.transpose()
```

```
[1 2 2]
```

```
[2 3 1]
[3 1 1]
```

And we can check a number of our tutorial problems, for example

```
(A.transpose())^2==(A^2).transpose()
```

```
True
```

Similarly Sage can tell the properties of the matrices

```
A.is_symmetric()
```

```
False
```

```
B = matrix(QQ, [[ 1, 2, -1],[ 2, 3, 4],[-1, 4, -6]]);B.is_symmetric()
```

```
True
```

Note that we have so far almost always considered rational numbers. As these are ratios of integers which are stored exactly the calculations are exact. In fact Sage can work exactly not just with rational numbers but with the set of all numbers which are the roots of polynomial equations with rational coefficients. However for real numbers some approximation must be made as we have discussed.

RDF and CDF, which are comprised of “double precision” floating point numbers, the first limited to just the reals, the second for the complex numbers. Double-precision refers to the use of 64 bits to store the sign, mantissa and exponent in the representation of a real number. This gives 53 bits of precision. For numerical calculations RDF and CDF are better than using just RR or CC for which some things are not defined.

Nonetheless it is easy to cook up examples where the precision is insufficient. For example from the Beezer book:

```
A = matrix(CDF, [[1.0, 0.0], [0.0, 1.0]]);A.is_symmetric()
```

```
True
```

```
A[0,1]=0.000000000002;A.is_symmetric()
```

```
False
```

```
A[0,1]=0.000000000001;A.is_symmetric()
```

```
True
```

```
A[0,1]
```

```
1e-12
```

Here 1×10^{-12} is being confused with zero in the test for symmetry.

However will let us set our own required precision

```
A.is_symmetric(tol=1.0e-13)
```

```
False
```

One way of performing mathematical "experiments" is to assign e.g. matrices with some random values and then check if conjectured property holds. For example a random matrix with integer entries:

```
A=random_matrix(ZZ, 5, 5);A
```

```
[ -1   1  -1   0   2]
[  1   0   1 -86  -1]
[  0   0   1   1  -1]
```

```
    [ -3   1   0 -71   1]
    [ -1   0   1  -1  -3]
```

```
B=random_matrix(ZZ, 5, 5)
```

We can now check whether matrix multiplication cares about order.

```
A*B==B*A
    False
```

It's possible we could have happened upon two matrices that commute, but it's very unlikely and just in case you can always run the "experiment" a bunch of times.

One important calculation we did was to find the inverse of a matrix. We can start from some matrix with rational entries:

```
A = random_matrix(QQ,5,5);A
    [ -1   2  -1   0   0]
    [  1   2  -1 1/2   2]
    [ -2   0   0   0 1/2]
    [ -2  -1   0 1/2  -1]
    [  2   0   1  -2   2]
```

Sage can actually check whether the matrix has an inverse (it can of course calculate the inverse but we'll get to that in a bit)

```
A.is_singular()
    False
```

Following the method we studied in class we append or augment this matrix by the 5×5 identity matrix:

```
A.augment(identity_matrix(5))
    [ -1   2  -1   0   0   1   0   0   0   0]
    [  1   2  -1 1/2   2   0   1   0   0   0]
    [ -2   0   0   0 1/2   0   0   1   0   0]
    [ -2  -1   0 1/2  -1   0   0   0   1   0]
    [  2   0   1  -2   2   0   0   0   0   1]
```

and then find the reduced row echelon form

```
(A.augment(identity_matrix(5))).rref()
    [      1       0       0       0       0   -1/17    2/17   -8/17
    2/17    1/17]
    [      0       1       0       0       0   -1/17  -15/17   26/17
    -49/17  -16/17]
    [      0       0       1       0       0  -18/17  -32/17   60/17
    -100/17  -33/17]
    [      0       0       0       1       0  -14/17   -6/17   24/17
    -40/17  -20/17]
    [      0       0       0       0       1   -4/17    8/17    2/17
    8/17    4/17]
```

The inverse matrix is the last 5 columns. So we want to make a new matrix from these.

```
Ainv=
((A.augment(identity_matrix(5))).rref()).matrix_from_columns(range(5,10));Ainv
```

```
[  -1/17    2/17   -8/17    2/17    1/17]
[  -1/17  -15/17   26/17  -49/17  -16/17]
[ -18/17  -32/17   60/17 -100/17  -33/17]
[ -14/17   -6/17   24/17  -40/17  -20/17]
[  -4/17    8/17    2/17    8/17    4/17]
```

Let us check that this is indeed the inverse matrix:

```
A*Ainv
```

```
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Of course Sage has a command that straightforwardly calculates the inverse.

```
Ainv - A.inverse()
```

```
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```

What happens if we make a matrix that doesn't have an inverse. We can form two row vectors.

```
row0=vector([1,2,3]);
row1=vector([4,5,6]);
```

Then we make a matrix from these vector, but the third row is simply some linear combination of the first two. This would end up being a row of zeros in Gauss-Jordon and so there would be no inverse.

```
C = matrix([row0, row1, 2*row0 - 4*row1])
```

```
C.inverse()
```

```
Traceback (click to the left of this block for traceback)
...
ZeroDivisionError: input matrix must be nonsingular
```

```
C.is_singular()
```

```
True
```

Calculus

We have mostly focused on aspects of Sage relevant to linear algebra but it can also be used in calculus. For example we can define a variable u and then a function of u say $\sin(u)$ and differentiate this function with respect to u.

```
u=var('u');
diff(sin(u), u)
```

```
    cos(u)
```

We can take more derivatives ...

```
diff(diff(sin(u),u),u)
```

```
    -sin(u)
```

or

```
diff(sin(u),u,2)
```

```
    -sin(u)
```

We can define a function of a variable: $f(x) = 4x^3 + 2x$ for example

```
x=var('x');f(x)=4*x^3+2*x;
```

It is slightly subtle what exactly f is

```
f
```

```
    x |--> 4*x^3 + 2*x
```

that is f is the function, or the map, not a specific value. We act on a value to get another value.

```
f(1)
```

```
    6
```

$f(x)$ is a specific value, but one which depends on the variable x.

```
f(x)
```

```
    4*x^3 + 2*x
```

We can then take derivatives

```
diff(f(x),x)
```

```
    12*x^2 + 2
```

This the derivative of the function evaluated at the point x. The derivative of the function f is itself a function:

```
diff(f,x)
```

```
    x |--> 12*x^2 + 2
```

We can take multiple derivative:

```
diff(f,x,2)
```

```
    x |--> 24*x
```

```
diff(f,x,3)
```

```
    x |--> 24
```

```
diff(f,x,4)
```

```
x |--> 0
```

We can also perform integrations and evaluate both indefinite

```
integral(x*sin(x^2), x)
```

```
-1/2*cos(x^2)
```

and definite integrals

```
integral(x*sin(x^2), x,0,pi)
```

```
-1/2*cos(pi^2) + 1/2
```

For functions the integral can be considered as the inverse of the derivative

```
integral(f,x)
```

```
x |--> x^4 + x^2
```

```
integral(f(x),x)
```

```
x^4 + x^2
```