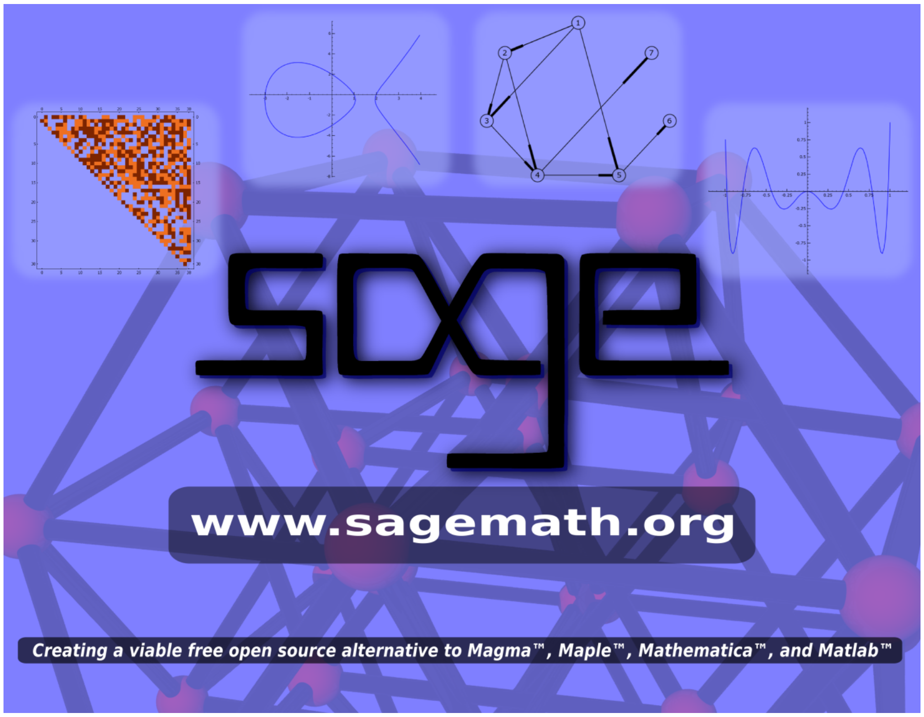
An Introduction to
Algebraic, Scientific, and Statistical Computing:
an Open Source Approach Using Sage

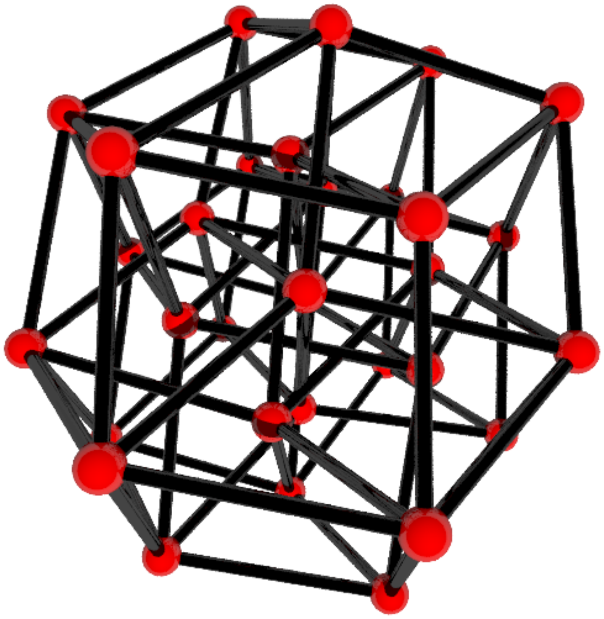

sage
www.sagemath.org
Creating a viable free open source alternative to Magma™, Maple™, Mathematica™, and Matlab™

William A. Stein

April 3, 2008

Abstract

This is an undergraduate textbook about algebraic, scientific, and statistical computing using the free open source mathematical software system Sage.



Contents

Preface

This is an undergraduate textbook about Sage, which is a free open source computational environment for the mathematical sciences. Sage includes optimized full-featured implementations of algorithms for compuations in pure mathematics, applied mathematics, and statistics.

I started the Sage project in late 2004 as a project to provide a viable open source free alternative to the Magma computer algebra system [BCP97]. My main motivation for this was frustration with not being allowed to easily change or understand the internals of Magma, worry about the longterm future of Magma, and concern that students and researchers in number theory could not easily use the Magma-based tools that I had spent six hard years developing. I started Sage as a new project instead of switching from Magma to an existing open source system, since the only free open source software for number theory is PARI [ABC$^+$], whose functionality was far behind that of Magma in several areas of interest to me (exact linear algebra and arithmetic of algebraic curves). Since PARI development moves slowly (or I am a very impatient person!), I didn't think it was likely this would change in the near future. PARI is a superb program – it just doesn't meet my needs, and has design constraints that make it impossible to modify so that it does so.

The Sage mathematical software system takes a fresh approach to mathematical software development and architecture. For instance, one major distinction between Sage and older systems is that Sage uses a standard language. Maple, Mathematica, Magma, Matlab, PARI, Gap, etc., all use their own special purposes language written just for mathematics. In sharp contract, one works with Sage using Python, which is one of the world's most popular *general purpose* scripting languages. This has some drawbacks, e.g., some mathematical expressions can be more difficult to express in Python than in Mathematica (say), but the overall pros greatly outweigh the cons. By using Python, one can use almost *anything* ever written in Python directly in Sage. And there is much useful Python code out there that adddresses a wide range of application areas[1]:

- "Python is fast enough for our site and allows us to *produce maintainable features in record times*, with a minimum of developers," said Cuong Do, Software Architect, **YouTube.com.**

[1] These quotes came from the Python website.

- “Google has made no secret of the fact they use Python a lot for a number of internal projects. Even knowing that, once *I was an employee, I was amazed at how much Python code there actually is in the Google source code system.*”, said Guido van Rossum, **Google**, creator of Python.

- “Python plays a key role in our production pipeline. Without it a project the size of *Star Wars: Episode II* would have been very difficult to pull off. From crowd rendering to batch processing to compositing, *Python binds all things together*,” said Tommy Burnette, Senior Technical Director, **Industrial Light & Magic.**

Instead of writing many of the core libraries from scratch like Maple, Mathematica, Magma, Gap, PARI, Singular, and Matlab did, in Sage I assembled together the best open source software out there, and built on it, always making certain that the complete system was easily buildable from source on a reasonable range of computers. I was able to do this to a large extent with Sage because of fortuitous timing: the components were out there and mature, their code is stable, and their copyright licences are clear and compatible (none of this was the case when the afformentioned math software was started). Of course there are drawbacks to this approach. Some of the upstream libraries can be difficult to understand, are written in a range of languages, and have different conventions than Sage. By strongly encouraging good relations between the Sage project and the projects that create many of the components of Sage, we turn these weakness into strengths.

A wide and vibrant community of developers and users have become involved with Sage. Due to the broad interests of this large community of developers, Sage has grown into a project with the following specific goal:

> **Mission Statement:** Provide a viable free open source alternative to Magma, Maple, Mathematica, and Matlab.

Among many other things, this mission statement implies that in order to succeed Sage should have a graphical user interface, 2D and 3D graphics, support for statistical and numerical computation, and much more.

Sage is starting to be recognized worldwide as a useful tool in mathematics education and research. Sage won first prize in the Scientific Category of the 2007 Tropheés du Libre.

The Sage development model has also matured. There are now regular releases of Sage about once every two weeks, and all code that goes into Sage is peer reviewed.

Sage has also received generous financial support from mathematics institutions (including the Clay Mathematics Institute, MSRI, PIMS, and IPAM), universities such as University of Washington and University of Bristol, and from the National Science Foundation, the Department of Defense, and Microsoft Corporation.

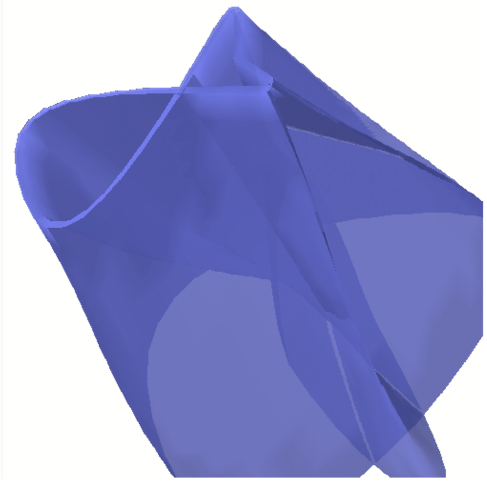
CLAY MATHEMATICS INSTITUTE
Institute for Pure & Applied Mathematics
IPAM
University of California, Los Angeles
MSRI
Mathematical Sciences Research Institute
PIMS
UNIVERSITY OF WASHINGTON
LVX SIT
1861
NSF
Microsoft Research
DEPARTMENT OF DEFENSE
UNITED STATES OF AMERICA

Exercise 0.1.

1. Make a list of 5 free open source programs you have used.
2. For each program, write down a corresponding commercial program, if it exists.
3. List some of the pros *and cons* from your perspective of the free program versus the commercial version.

Chapter 1

Computing with Sage



1.1 Installing and Using Sage

1.1.1 Installing Sage

You are strongly encouraged to follow along with all examples in this book. Get Sage up and running now! Try out everything, modify things, experiment, look under the hood, and do all the exercises!

The website `http://sagemath.org/lists.html` lists the two main Sage mailing list: sage-support and sage-devel. The first is for all questions about using and installing Sage, and for bug reports. The second is for discussion related directly to modifying and improving Sage. Join these lists; note that you will likely want to select *daily digest* for message deliver, so that you receive one email a day instead of 50!

If you find a bug in Sage, *we want to know about it!* Send an email to sage-support describing the bug in as much detail as you can.

The exact details for installing Sage will change over time. The following briefly describes the situation in April 2008; for more details see the Sage installation guide.

You install Sage on any computer either by extracting a pre-compiled binary or building from source.

To install a binary, download the binary (either a .tar.gz file, a .dmg file, or a .7z file) from `http://sagemath.org`, extract the file, and following the directions in the README.txt file. In most cases, you can simply put the extracted `sage-x.y.z` directory somewhere, and run Sage from there.

Installing Sage on Microsoft Windows currently involves installing the VMware player program, then running a virtual machine. This will likely change significantly within a year.

Currently you can only build Sage from source on Linux and OS X (or on Windows in a Linux virtual machine). To build from source, which will take over an hour, first download `sage-x.y.z.tar` from `http://sagemath.org/download.html`. Once you have downloaded Sage, extract the "tarball" as follows:

```
$ tar xvf sage-x.y.z.tar
```

Then run the `make` command in the `sage-x.y.z` directory:

```
$ cd sage-x.y.z
$ make
... 137600 lines of output ...
real    102m57.600s
user    71m10.115s
sys     14m56.116s
To install gap, gp, singular, etc., scripts
in a standard bin directory, start sage and
type something like
   sage: install_scripts(’/usr/local/bin’)
at the SAGE command prompt.

SAGE build/upgrade complete!
```

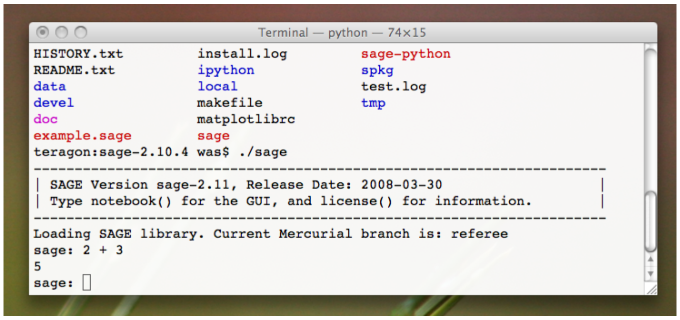
The above should work if you have gcc, g++, make, and a few other prerequisites installed (Sage includes almost all depedencies), and are using a supported architecture and operating system. An advantage of building from source is that if you can build from source, then you can change absolutely any part of Sage and use the modified version. If you run into trouble, email `sage-support`.

Exercise 1.1. Install Sage on a computer.

1.1.2 The command line and the notebook

The command line and the Sage notebook provide two complementary ways for you to interactively work with Sage. The command line provides a simple and powerful way to type Sage commands. The notebook provides a modern AJAX web-browser based graphical interface to Sage. Both the notebook and command line have advantages and disadvantages, and you will want to become familiar with each. For example, the notebook is better for graphics, whereas the command line is better for debugging and profiling code (see Section 1.4).

In Linux or OS X, start the command line by typing `./sage` in the directory where you installed Sage, or just type `sage` if the Sage install directory `sage-x.y.z` is in your PATH. In Windows, after starting the Sage vmware image, type `sage` at the login prompt.

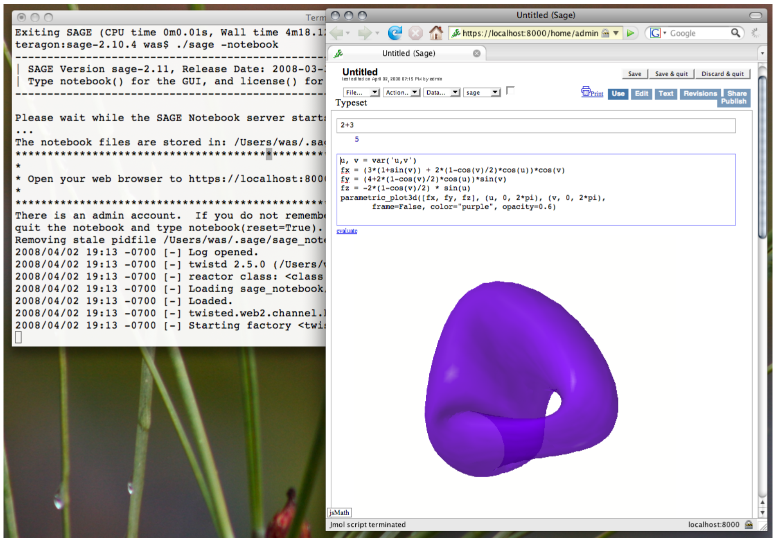


```
HISTORY.txt          install.log          sage-python
README.txt           ipython              spkg
data                 local                test.log
devel                makefile             tmp
doc                  matplotlibrc
example.sage         sage
teragon:sage-2.10.4 was$ ./sage
----------------------------------------------------------------------
| SAGE Version sage-2.11, Release Date: 2008-03-30                   |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
Loading SAGE library. Current Mercurial branch is: referee
sage: 2 + 3
5
sage: 
```

In Linux or OS X, start the notebook by typing `./sage -notebook` in the directory where you installed Sage:

```
teragon:~ was$ sage -inotebook
----------------------------------------------------------------------
| SAGE Version 2.10.4, Release Date: 2008-03-16                      |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

Please wait while the SAGE Notebook server starts...
...
The notebook files are stored in: /Users/was/.sage//sage_notebook
WARNING: Running the notebook insecurely may be dangerous.
Make sure you know what you are doing.
**************************************************
*                                                *
* Open your web browser to http://localhost:8000 *
*                                                *
**************************************************
```

In Windows, type `notebook` at the login prompt, then use your web browser to navigate to the URL that is displayed. You can also use Sage without installing it on your computer by signing up for an account at `https://sagenb.org`.



```
Exiting SAGE (CPU time 0m0.01s, Wall time 4m18.1
teragon:sage-2.10.4 was$ ./sage -notebook
----------------------------------------------------
| SAGE Version sage-2.11, Release Date: 2008-03-
| Type notebook() for the GUI, and license() for
----------------------------------------------------

Please wait while the SAGE Notebook server start
...
The notebook files are stored in: /Users/was/.sa
****************************************************
*
* Open your web browser to https://localhost:800
*
****************************************************
There is an admin account.  If you do not rememb
quit the notebook and type notebook(reset=True).
Removing stale pidfile /Users/was/.sage/sage_not
2008/04/02 19:13 -0700 [-] Log opened.
2008/04/02 19:13 -0700 [-] twistd 2.5.0 (/Users/
2008/04/02 19:13 -0700 [-] reactor class: <class
2008/04/02 19:13 -0700 [-] Loading sage_notebook
2008/04/02 19:13 -0700 [-] Loaded.
2008/04/02 19:13 -0700 [-] twisted.web2.channel.
2008/04/02 19:13 -0700 [-] Starting factory <twi
```

```
2+3
    5
```

```
u, v = var('u,v')
fx = (3*(1+sin(v)) + 2*(1-cos(v)/2)*cos(u))*cos(v)
fy = (4+2*(1-cos(v)/2)*cos(u))*sin(v)
fz = -2*(1-cos(v)/2) * sin(u)
parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
       frame=False, color="purple", opacity=0.6)
```

Exercise 1.2.

1. Using the Sage command line, compute $123 + 456$.

2. Using the Sage notebook, compute $456 + 789$.

Tab completion and help are incredibly useful features of both the command line and notebook, and work in almost the same way in both. This is useful in two ways. First, if you type the first few letters of a command, then press the tab key, you'll see all commands that begin with those first few letters.

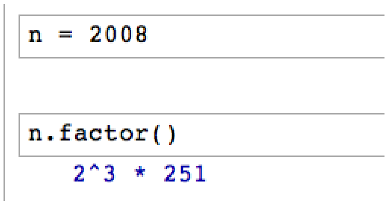
The second way in which tab completion is useful is that it shows you most of the things you can do with something. For example, if n is an integer and you type `n.[tab key]`, you'll see a list of all the functions that you can call on n. For example, the code `n=2008` in Sage sets the *variable* n equal to the integer 2008:

```
sage: n = 2008
```

Then type `n.fa[tab key]` (press the tab key after typing `n.fa`), you'll see that `n.factor` is a command associated to n. Type `n.factor()` to factor n:

```
sage: n = 2008
sage: n.factor()
2^3 * 251
```

Here we are showing all input and output as if they were typed at the command line. If you're using the notebook press shift-enter after typing $n = 2008$ into an input cell. After computing the factorization you will see something like this:



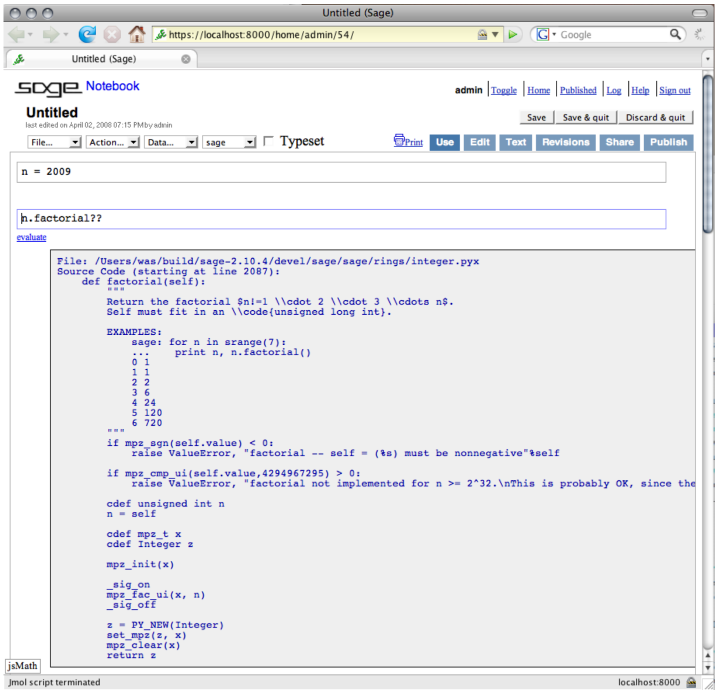
```
n = 2008

n.factor()
    2^3 * 251
```

Exercise 1.3. Use tab completion to determine how to compute the *factorial* of 100.

In the command line every command you type is recorded in the history. Use the up arrow to scroll through previous commands; this history even works if you quit Sage and restart. Likewise, in the notebook previous commands are visibly recorded in input cells, and you can click or use the arrow keys to move to a previous cell and press shift-enter to evaluate it again.

Exercise 1.4. 1. Quit the Sage command line, restart Sage, and press the up arrow until you see `n = 2008`. Change 2008 to 2009 and press enter. Then factor the result, again using the up arrow to select `n.factor()`.

2. In the Sage notebook click and change `n = 2008` to `n = 2009`, then press shift-enter twice to see how 2009 factors.



```
n = 2009

n.factorial??

File: /Users/was/build/sage-2.10.4/devel/sage/sage/rings/integer.pyx
Source Code (starting at line 2087):
    def factorial(self):
        """
        Return the factorial $n!=1 \\cdot 2 \\cdot 3 \\cdots n$.
        Self must fit in an \\code{unsigned long int}.

        EXAMPLES:
            sage: for n in srange(7):
            ...       print n, n.factorial()
            0 1
            1 1
            2 2
            3 6
            4 24
            5 120
            6 720
        """
        if mpz_sgn(self.value) < 0:
            raise ValueError, "factorial -- self = (%s) must be nonnegative"%self

        if mpz_cmp_ui(self.value,4294967295) > 0:
            raise ValueError, "factorial not implemented for n >= 2^32.\nThis is probably OK, since the

        cdef unsigned int n
        n = self

        cdef mpz_t x
        cdef Integer z

        mpz_init(x)

        _sig_on
        mpz_fac_ui(x, n)
        _sig_off

        z = PY_NEW(Integer)
        set_mpz(z, x)
        mpz_clear(x)
        return z
```

If `n` is any object in Sage or `foo` any function (even the function `n.factorial`), type ? after it and press enter to see a description of the command along with examples. In the notebook, you can also type `n.factorial(` and press the tab key for popup help. If you put two question marks instead of one you'll see the help *and source code* of the function or object, i.e., the computer code that defines that function or object.

There are several ways to time how long it takes for something in Sage to run. If the command is just one line, put the word `time` at the beginning of the line, e.g.,

```
sage: time n = factorial(10^5)
CPU time: 0.10 s,  Wall time: 0.10 s
```

Exercise 1.5. Make up a line of input to Sage that takes at least ten seconds to evaluate.

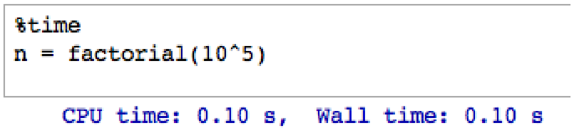
You can time execution of all the code in a notebook cell by putting `%time` at the beginning of the cell. For example:

```
{{{
%time
n = factorial(10^5)
///
CPU time: 0.10 s,  Wall time: 0.10 s
}}}
```

Above we have used the following notation:

```
{{{
INPUT
///
OUTPUT
}}}
```

Thus the above looks like the following in the notebook:



```
%time
n = factorial(10^5)
```

```
CPU time: 0.10 s,  Wall time: 0.10 s
```

You can also time execution of a block of code by typing `t = cputime()` before the block, then after the block typing `cputime(t)`. This will output the number of CPU seconds that elapsed during the computation. For the physical amount of time that actually elapsed on your "wall" clock, type `t = walltime()` then later type `walltime(t)`. This can be useful, especially in complicated programs.

Once you start writing complicated Sage programs, especially when using the command line, you'll want to place code in an external file and edit it with a standard code editor (use the special Python mode if your editor has one). This works very well in Linux and OS X, where you put the code in the file of your choice and type

```
      sage: load filename.sage
```

to execute all the code in `filename.sage`. Under Windows, the situation is currently more complicated – you either have to configure VMware shared folders, or regularly upload the file to the Sage notebook using `Data --> Upload or Create File`. Another separate option[1] is to use the Windows program WinSCP Using WinSCP you can login to the VMware machine (use login name 'login' and password 'sage'). Then you can select edit from the context menu and edit files. WinSCP takes care of automatically uploading and downloading the modified version of the file whenever you change it. If you call the file filename.sage, you would type the following to load the file you're editing:

```
      sage: load /home/login/filename.sage
```

For OS X or Linux users, if you're constantly editing `filename.sage`, and find yourself regularly typing `load filename.sage` into Sage, you should instead *attach* `filename.sage` by typing

```
      sage: attach filename.sage
```

This works exactly like `load filename.sage`, except that if `filename.sage` is changed and you execute a new command, Sage reloads `filename.sage` before executing the command. Try it; you'll like it.

Exercise 1.6. Create a file `hi.sage` that contains the line `print "hello"`. Load that file into Sage using the load command, and see that hello is printed out. If you're using OS X or Linux, attach `hi.sage`, then change "hello" to something else, then type something at the Sage prompt – you should see something besides hello printed out.

1.1.3 Loading and saving variables and sessions

As mentioned above, in Sage you create a new variable by assigning to it. E.g., typing `n = 2008` creates a new variable n and assigns the value 2008 to it. Most

[1] suggested by Lars Fischer; the author has not yet tried this!!

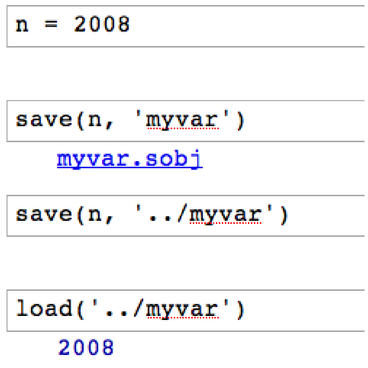
(but not all![2]) individual variables, even incredibly complicated ones, can be easily saved or loaded to disk using the save and load functions. This can be extremely useful if it takes a long time to compute a variable, and you want to save it to use it again later.

First we discuss using save and load from the command line, then from the notebook. In the following session, we set n to 2008, then save n to the variable `myvar.sobj`, quit sage, restart, and reload the value of n from that file.

```
teragon:~ was$ sage
sage: n = 2008
sage: save(n, 'myvar')
sage: quit
Exiting SAGE (CPU time 0m0.02s, Wall time 0m41.28s).
teragon:~ was$ ls -l myvar.sobj
-rw-r--r--  1 was  was  48 Mar 26 22:40 myvar.sobj
teragon:~ was$ sage
sage: load('myvar.sobj')
2008
```

Note that `.sobj` is added to the end of the filename, if it isn't given. Also, all saved objects are compressed to save disk space.

Using save and load from the notebook is a bit different, since every notebook cell is executed in a different subdirectory. In the following example, in the first cell the value of n is saved in the file `myvar.sobj` in the current cell directory. In general, in the notebook every file created in the current (cell) directory is either displayed embedded in the output or a link to it is created – you could right click and download `myvar.sobj` to your hard drive if you wanted. In the second example we save the value of n to the file `myvar.sobj` in the directory above the current cell directory. This directory is fixed independent of the cell we're working in, so we can load `myvar.sobj` in another cell, which we do.



```
n = 2008

save(n, 'myvar')
    myvar.sobj

save(n, '../myvar')

load('../myvar')
    2008
```

Not only can you load and save individual objects, but you can save all

[2] See the Python documentation on the pickle module.

save-able objects defined in the current session.[3] In this example we define two variables $n = 2008$ and $m = -2/3$, save the session, then restart, load the session and see that n and m are again defined.

```
teragon:~ was$ sage
sage: n = 2008
sage: m = -2/3
sage: save_session('session')
sage: quit
Exiting SAGE (CPU time 0m0.08s, Wall time 9m11.00s).
teragon:~ was$ sage
losage: load_session('session')
sage: n
2008
sage: m
-2/3
```

Think of `save_session` as being sort of like "saving your game" in a video game.

Exercise 1.7. (Command line only.) A useful feature of session loading and saving is that you can save two separate sessions and load them at the same time. Instead of the second loaded session overwriting the first completely, the two are merged. Try this out – define several variables in two separate sessions, save both sessions. Quit, load one session, then load the other and see that the sessions are merged.

Behind the scenes session loading and saving works essentially by calling save on every currently defined variable – the objects that can't be saved just don't get saved.

1.2 Python: the Language of Sage

Python has excellent support for basic data structures, including lists, tuples, strings, dictionaries, sets, and user-defined classes. Functions, control flow, and error handling are also all well supported. In this section we just give a very brief overview with exercises of the Python language, which should be enough to get you going. There are many excellent books about the basics of Python; my favoriates are the official Python Tutorial, Dive Into Python, and Python in a Nutshell, and I strongly encourage you to look at them. In fact, the examples below very closely follow those in the Python Tutorial.

1.2.1 Lists, Tuples, Strings, Dictionaries and Sets

The five basic data structures you will use constantly in Python programming are lists, tuples, strings, dictionaries, and sets.

[3] **WARNING:** As of March 30, 2008, `save_session` in the Sage notebook is broken.

Lists

Lists and tuples are the two main sequence types, and both allow you to store and work with arbitrary finite sequences of Python objects. Both are indexed *starting at 0*. The main difference between lists and tuples is that lists are mutable, whereas tuples are immutable; this means that you can easily change lists, which has some subtle pros and cons as we will see below.

You can create a list by explicitly listing its elements, by adding, multiplying, appending to and slicing existing lists:

```
sage: a = ['spam', 'eggs', 100, 1234]; a
['spam', 'eggs', 100, 1234]
sage: a[0]
'spam'
sage: a[3]
1234
sage: a[-2]
100
sage: a[1:-1]
['eggs', 100]
sage: a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
sage: 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
sage: a.append(314); a
['spam', 'eggs', 100, 1234, 314]
```

Absolutely anything can go in a list, e.g., other lists, etc., and you can change the entries:

```
sage: b = [[1,2], 1.234, 3/4, vector([1/3,2,3]), [[1]]]; b
[[1, 2], 1.23400000000000, 3/4, (1/3, 2, 3), [[1]]]
```

Assignment to lists is also quite flexible. For example, if you use a negative index the index counts down from the right. You can also use the : slicing notation to change whole sections of a list at once.

```
sage: b[0] = 5; b
[5, 1.23400000000000, 3/4, (1/3, 2, 3), [[1]]]
sage: b[-1] = 'hello'; b
[5, 1.23400000000000, 3/4, (1/3, 2, 3), 'hello']
sage: b[:2]
[5, 1.23400000000000]
sage: b[:2] = [1,2,3,4]; b
[1, 2, 3, 4, 3/4, (1/3, 2, 3), 'hello']
sage: b[1:4] = []; b
[5, 'hello']
sage: len(b)        # the length of b
2
```

There many list methods such as append, remove, sort, index, etc. Make a list `v` then type `v.[tab key]` to see a list of those methods.

You can also use the beautiful and powerful *list comprehension* construction to make new lists from old lists. This is similar to "set builder notation" in mathematics:

```
sage: vec = [2, 4, 6]
sage: [3*x for x in vec]
[6, 12, 18]
sage: [3*x for x in vec if x > 3]
[12, 18]
sage: [3*x for x in vec if x < 2]
[]
sage: [[x,x^2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
sage: vec1 = [2, 4, 6]
sage: vec2 = [4, 3, -9]
sage: [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
sage: [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
sage: [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Finally use the `del` statement to delete an element of a list or a section of a list:

```
sage: a = [-1, 1, 66.25, 333, 333, 1234.5]; a
[-1, 1, 66.2500000000000, 333, 333, 1234.50000000000]
sage: del a[0]
sage: a
[1, 66.2500000000000, 333, 333, 1234.50000000000]
sage: del a[2:4]
sage: a
[1, 66.2500000000000, 1234.50000000000]
sage: del a[:]
sage: a
[]
```

Note in particular the command `del a[2:4]` which deletes the entries of a in positions 2 and 3. These are exactly the entries you see if you type `a[2:4]`.

Tuples

As mentioned above, tuples are sequence types like lists, except the objects in the tuple can’t be deleted or replaced once the tuple has been created.

```
sage: t = (12345, 54321, ’hello!’)
sage: t[0]
12345
sage: t
(12345, 54321, ’hello!’)
sage: type(t)
<type ’tuple’>
sage: t[0] = 5   # tuples are immutable
Traceback (most recent call last):
...
TypeError: ’tuple’ object does not support item assignment
sage: # Tuples may be nested:
sage: u = (t, (1, 2, 3, 4, 5))
sage: u
((12345, 54321, ’hello!’), (1, 2, 3, 4, 5))
```

There is also an eloquent tuple unpacking system, which makes it simple to assign several variables to the entries of a tuple.

```
sage: t = (12345, 54321, ’hello!’); t
(12345, 54321, ’hello!’)
sage: # tuple UNPACKING
sage: x, y, z = t
sage: print x, y, z
12345 54321 hello!
```

Tuple unpacking also supports having functions to have multiple return variables, without running into the numerous painful gotcha's that other languages (e.g., Matlab and Magma) suffer from having multiple return variables.

```
sage: def foo(a,b):
...       return a+b, a-b
sage: c,d = foo(2,3)    # multiple return values!
sage: print c, d
5 -1
```

1.2.2 Control Flow, Errors and Exceptions

- Loops, functions, control statements; putting code in files and modules (import/load/attach)

1.2.3 Classes and Inheritence

- user-defined classes; object-oriented programming and mathematics, inheritence

1.3 Cython: Compiled Python

1.3.1 The Cython project

1.3.2 How to use Cython: command line, notebook

when to use; when good / when bad

1.4 Debugging and Profiling

The best perspective to take when writing or indeed using code is to assume that it has bugs. Be skeptical! If you read source code of Sage or its components you'll sometimes see things that will make you worry. This is a good thing, e.g., this occurs in a comment in the Sage code by Jonathan Bober for computing the number of partitions of an integer (the `number_of_partitions` command):

```
// Extensive trial and error has found 3 to be the smallest
// value that doesn't seem to produce any wrong answers.
// Thus, to be safe, we use 5 extra bits.
```

Note, by the way, that Bober's code for computing the numer of partitions of an integer is currently faster than anything else in the world, and it is not known to have any bugs.

A healthy amount of skepticism and worry is a good to cultivate when doing computational mathematics. Never listen to anybody who suggests you do otherwise. These same sort of issues occur in closed source systems such as Mathematica, but you don't get to see them – that doesn't make the chance of bugs less likely. In fact, the Mathematica documentation says the following:

> But just as long proofs will inevitably contain errors that go undetected for many years, so also a complex software system such as Mathematica will contain errors that go undetected even after millions of people have used it

There can even be bugs in the implementation of basic arithmetic, even at the hardware level! [[reference the pentium hardware bug from the 1990s that was found by a number theorist trying to debug his code]] So be skeptical when you do computations. Always think about what mathematical computations tell you, and try to find ways to *double check* results.

It almost goes without saying, but beautifully written, well-documented code that has been around for a long time and used a great deal is generally much less likely to be seriously buggy than newly written code. A great example of such high quality – by aging – code is NTL (the Number Theory Library [[url]]). Quality code that has been used for years, and thus likely has few bugs is like gold – treat it that way; don't just toss it out and assume that something new that you sit down and write is likely to be better. That said, if you persist you can and will write beautiful code that equals or surpasses anything ever done before. When you do this, please consider contributing you code to the Sage project.

There are several good reasons to write new code. One excellent reason is that you simply want to understand an algorithm well, perhaps one you're learning about in a course, a book, a paper, or that you just designed. Implementing an algorithm correctly forces you to understand every detail, which can provide new insight into the algorithm. If you're implementing a nontrivial algorithm that is described in a book or paper, the chances that it is wrong in some way (e.g., a typo in a formula, a fundamental mistake in the algorithm, whatever) is quite high – so you will definitely learn something, and possibly improve the mathematical literature while you're at it.

Another great reason to write new code is to implement an algorithm that isn't available in Sage. When you do this, make sure to be skeptical about the correctness of your code; always test it extensively, document it, etc., just to increase the chances it might always work correctly. And if there is any way to independently verify correctness of the output of code, attempt to implement this too. If the algorithm is available in other mathematical software such as Maple or Mathematica, and you have that program, use the interfaces described in Section 1.6 to write code that automatically tests the output of your implementation against the output of the implementation in that other system. Include such test code in your final product.

1.4.1 Using Print Statements

Yes, you *can* use print statements for debugging your code. There is no shame in this! Especially when using Python where you do not have to recompile every time, this can be a particularly useful technique.

specific techniques for using print statements (how to figure out things)

use attach for .py or .sage files.

1.4.2 Debugging Python using pdb

Use command line and do `%pdb`

1.4.3 Debugging Cython using gdb

how to trace things through

bt is enough to get pretty far

multiple processes can be confusing

1.5 Source Control Management

1.5.1 Distributed versus Centralized

1.5.2 Mercurial

1.6 Using Mathematica, Matlab, Maple, etc., from Sage

A distinctive feature of Sage is that Sage supports using Maple, Mathematica, Octave, Matlab, and many other programs from Sage, assuming you have the relevant program (there are some caveats). This makes it much easier to combine the functionality of these systems with each other and Sage.

Before discussing interfaces in more detail, we make a few operating system dependent remarks.

what works on all os's; in particular gap, singular, gp, maxima, always there.

what works on linux

on os x

on windows

—-

example of using gp to do something.

example of using mathematica

example of using maple

example of using matlab/octave

—-

eval versus call.

Discussion of what goes on behind the scenes. Files used for large inputs – named pipes for small.

—-

Warning– multiple processes; complicated; can get parallel, which is harder to think about...

Chapter 2

Algebraic Computing

Algebraic computing is concerned with computing with the algebraic objects of mathematics such as arbitrary precision integer and rational numbers, groups, rings, fields, vector spaces and matrices, and other object. The tools of algebraic computing support research and education in pure mathematics, underly the design of error correcting codes and cryptographical systems, and play a role in scientific and statistical computing.

2.1 Groups, Rings and Fields

2.1.1 Groups

2.1.2 Rings

2.1.3 Fields

2.2 Number Theory

2.2.1 Prime numbers and integer factorization

2.2.2 Elliptic curves

2.2.3 Public-key cryptography: Diffie-Hellman, RSA, and Elliptic curve

2.3 Linear Algebra

2.3.1 Matrix arithmetic and echelon form

Matrix multiplication using a numerical BLAS (in both mod p and over ZZ cases)

2.3.2 Vector spaces and free modules

2.3.3 Solving linear systems

Applicatin: computing determinants over ZZ

2.4 Systems of polynomial equations

2.5 Graph Theory

2.5.1 Creating graphs and plotting them

2.5.2 Computing automorphisms and isomorphisms

2.5.3 The genus and other invariants

Chapter 3

Scientific Computing

Scientific computing is concerned with constructing mathematical models and using numerical techniques to solve scientific, social, and engineering problems.

3.1 Floating Point Numbers

3.1.1 Machine precision floating point numbers

3.1.2 Arbitrary precision floating point numbers

3.2 Interval arithmetic

3.3 Root Finding and Optimization

3.3.1 Single variable: max, min, roots, rational root isolation

3.3.2 Multivariable: local max, min, roots

3.4 NumericalSolution of Linear Systems

3.4.1 Solving linear systems using LU factorization

3.4.2 Solving linear systems iteratively

3.4.3 Eigenvalues and eigenvectors

3.5 Symbolic Calculus

3.5.1 Symbolic Differentiation and integration

3.5.2 Symbolic Limits and Taylor series

3.5.3 Numerical Integration

Chapter 4

Statistical Computing

4.1 Introduction to R and Scipy.stats

4.1.1 The R System for Statistical Computing

4.1.2 The Scipy.stats Python Library

4.2 Descriptive Statistics

4.2.1 Mean, standard deviation, etc.

4.3 Inferential Statistics

4.3.1 Simple Inference

4.3.2 Conditional Inference

4.4 Regression

4.4.1 Linear regression

4.4.2 Logistic regression

Bibliography

[ABC$^+$] B. Allombert, K. Belabas, H. Cohen, X. Roblot, and I. Zakharevitch, `PARI/GP`, `http://pari.math.u-bordeaux.fr/`.

[BCP97] W. Bosma, J. Cannon, and C. Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3–4, 235–265, Computational algebra and number theory (London, 1993). MR 1 484 478