Sage and GP: A (Very) Short Introduction

Mathematical software is part of the standard toolkit of mathematicians. Throughout this course we will use the programs *Sage* and GP to do computations. The goal of this handout is to provide some basic orientation on the two programs, including where to get them. Both programs are free, both can be used online or installed on your own machine, both are very powerful, and both take some time to learn.

The two programs are different in several ways. GP is mostly intended for use by number theorists, while *Sage* wants to tackle all kinds of mathematics. Indeed, *Sage* has incorporated the functionality of many other free mathematics programs, including Maxima, Octave, R, GAP, and even GP itsef. *Sage* runs in a browser window or a terminal, while GP is more old-fashioned and runs in a terminal window. I learned GP first, and it's still my go-to program for quick computations, but lately I have been trying to teach my students to use *Sage*.

Of course, there are other mathematical software tools. Both *Maple* and *Mathematica* are well-known and quite powerful. The people who make *Mathematica* are also behind Wolfram Alpha, which can do lots of mathematics as well. These can all do some of the things we need to do, but I have decided to focus on GP and *Sage* because they are free and I know how to use them.

Pari and GP

The interactive calculator GP was designed to serve the needs of people working in number theory. It is actually a front end for a software library called Pari, which can be used to create mathematical programs in C and C++. (Such programs are faster than using GP, but for most things GP is fast enough.)

Pari and GP were created by Henri Cohen and his team in 1985 and it has continued to grow since them. The current chief developer is Karim Belabas, who has many collaborators. The Pari-GP system is still actively developed, with new features being added and bugs being fixed all the time. The home web site for Pari-GP is where you go to download it, but you can also find a lot of information there.

The normal way to use GP is to download and install it on your computer. The installer will usually create an icon on your desktop; clicking that icon opens a terminal window where you will see something like this.

```
        GP/PARI CALCULATOR Version 2.11.2 (released)
amd64 running mingw (x86-64/GMP-6.1.2 kernel) 64-bit version
  compiled: Apr 28 2019, gcc version 6.3.0 20170516 (GCC)
                  threading engine: single
       (readline v6.2 enabled, extended help enabled)

            Copyright (C) 2000-2018 The PARI Group
```

```
PARI/GP is free software, covered by the GNU General Public License,
and comes WITHOUT ANY WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?17 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000
gp >
```

Some things to note:

- The last line is a *prompt*: GP expects you to type in some instruction and then hit the enter/return key. Prompts are configurable; on my machine it is the current time followed by `gp >`
- If you type `\q` and hit enter/return you will exit the program.
- If you type a question mark or a question mark followed by a number you will get (some) help.

Let's try something simple.

```
gp > 25!
%1 = 15511210043330985984000000
```

Notice that GP understands the factorial notation and is not afraid of computing large numbers. Also notice that it uses `%1` to label the result. You can later refer to this number by that name if you want to. Or you can give it a name.

```
gp > a=44/5
%2 = 44/5
```

The basic philosophy of GP is that it assumes the numbers you enter are in the simplest setting that makes sense. In this case, GP assumes that you mean the *rational number* 44/5. If you want the decimal version (i.e., the *real number*), you can either make it a decimal from the beginning or multiply it by 1.0.

```
gp > b=44.0/5
%3 = 8.8000000000000000000000000000000000000
gp > a*1.0
%4 = 8.8000000000000000000000000000000000000
```

Real numbers are presented with a default precision, in this case 38 significant digits. You can enter polynomials in the usual way.

```
gp > p=x^2-2*x+1
%5 = x^2 - 2*x + 1
gp > q = x^2 - 3*x + 2
%6 = x^2 - 3*x + 2
gp > p*q
%7 = x^4 - 5*x^3 + 9*x^2 - 7*x + 2
gp > p/q
%8 = (x - 1)/(x - 2)
```

Notice that the quotient of two polynomials is a rational function, given in lowest terms. What if we try something strange?

```
gp > log(p)
%9 = -2*x - x^2 - 2/3*x^3 - 1/2*x^4 - 2/5*x^5 - 1/3*x^6 - 2/7*x^7
    - 1/4*x^8 - 2/9*x^9 - 1/5*x^10 - 2/11*x^11 - 1/6*x^12 - 2/13*x^13
    - 1/7*x^14 - 2/15*x^15 + O(x^16)
```

As usual, GP makes the most reasonable interpretation of what you want, and returns the Taylor series. (The output is actually one long line, which I have broken up to make it easier to read.)

Factoring is straightforward.

```
gp > factor(42)
%10 =
[2 1]
[3 1]
[7 1]
```

Notice that the answer is a matrix whose rows give the prime factors and then the powers. Hard(er) factorizations are fine too:

```
gp > factor(23432432)
%11 =
[   2 4]
[ 313 1]
[4679 1]
```

One computation you will be doing a lot is finding the greatest common divisor:

```
gp > gcd(1421541,243234)
%12 = 9
```

There's also `gcdext`, which will be even more useful.

If the input of `factor` is a polynomial, that's fine too.

```
gp > factor(x^3+x)
%13 =
[      x 1]

[x^2 + 1 1]
```

We can enter numbers modulo m (if you don't know what those are, don't worry, you soon will) and do operations with them.

```
gp > Mod(234,37)
%14 = Mod(12, 37)
gp > %10^16
%15 = Mod(9, 37)
gp > 1/%10
%16 = Mod(34, 37)
gp > %10^36
%17 = Mod(1, 37)
```

All GP constructions are iterative, so you can construct polynomials with coefficients in $\mathbb{Z}/37\mathbb{Z}$ and so on. Things can break if you overdo it (for example, power series in x whose coefficients are power series in y are problematic), but generally it all works well.

What if we ask for help?

```
gp > ?
Help topics: for a list of relevant subtopics, type ?n for n in
   0: user-defined functions (aliases, installed and user functions)
   1: PROGRAMMING under GP
   2: Standard monadic or dyadic OPERATORS
   3: CONVERSIONS and similar elementary functions
   4: functions related to COMBINATORICS
   5: NUMBER THEORETICAL functions
   6: POLYNOMIALS and power series
   7: Vectors, matrices, LINEAR ALGEBRA and sets
   8: TRANSCENDENTAL functions
   9: SUMS, products, integrals and similar functions
  10: General NUMBER FIELDS
  11: Associative and central simple ALGEBRAS
  12: ELLIPTIC CURVES
  13: L-FUNCTIONS
  14: MODULAR FORMS
  15: MODULAR SYMBOLS
  16: GRAPHIC functions
  17: The PARI community
Also:
  ? functionname (short on-line help)
  ?\             (keyboard shortcuts)
  ?.             (member functions)
Extended help (if available):
  ??          (opens the full user's manual in a dvi previewer)
  ??  tutorial / refcard / libpari
```

```
            (tutorial/reference card/libpari manual)
??  refcard-ell
            (or -lfun/-mf/-nf: specialized reference card)
??  keyword (long help text about "keyword" from the user's manual)
??? keyword (a propos: list of related functions).
```

The next step is to make a more specific request for help, say with `?5` or `?gcd`. As that list shows, GP can do a lot.

While the default way to use GP is in a terminal window, there is also a (rather new) browser-based version. It is also possible to use the Sage Cell Server (see below) in GP-mode to do quick computations.

There is a lot more to say, but that should get you started. At the Pari-GP home page you can find a tutorial and a user's manual. There are also email lists where you can ask for help if necessary.

Sage

Sage is an ambitious attempt to create powerful mathematical software that is free and open-source. The ultimate ambition, says the Sage home page, is to create "a viable free open source alternative to *Magma*, *Maple*, *Mathematica* and MATLAB." (I'd say that they are very close to that, and in some aspects well beyond.) The development approach emphasizes openness: while William Stein is the leader of the team, contributions have come from across the mathematical community.

Sage can be used through a web interface, without needing to download and install the program. There are two ways to do that: either the Sage Cell Server or the more elaborate interface based on projects and notebooks offered by *CoCalc*. It is also possible to download and install the program on your own computer. When you do, you can run *Sage* in a terminal window or you can run it in your browser. The latter is much like using *CoCalc*, but the program is running on your local machine.

Of the two web interfaces, the Sage Cell Server is particularly easy to use for small computations. It presents the user with a big blank rectangle where one can type in Sage commands. Below is a button labeled "Evaluate," which does exactly that. The output appears below. The downside is that Sage will not remember what you did, so if you define a symbol and then press the "Evaluate" button, you cannot use it again without repeating the definition.

There are two very nice features of the Cell Server that deserve note. First, it works on a tablet or phone. Second, because Sage incorporates GP and other open-source mathematical software, the Cell Server can be put into GP mode to do computations in GP. I have also used it in R mode.

CoCalc requires creating an account. Once you log in to your account, you can create projects, and each project can contain many notebooks. Notebooks allow you to enter lines of Sage code, which are evaluated when you hit Shift-Enter. Definitions and results are remembered within each session.

If you are going to use the program a lot, then of course the right thing to do is to download and install it. It takes quite a bit of space, but having it on your own machine avoids connectivity issues.

In mathematical terms, there is an important philosophical difference between GP and *Sage*. In GP, as we noted, the program assumes (or guesses) the mathematical context for the objects you create. *Sage*, on the other hand, prefers to be told. You do that by creating a ring or field (or something else) in which you are operating.

Time to show some examples. You can find many more in A Tour of Sage, which is available online.

The basic "calculator" commands work as expected. Entering

```
3 + 5
57.1^100
```

into the Sage Cell Server and hitting "Evaluate" will produce

```
8
4.609043686613944033110074777753591036 9 E175
```

You can get some space between the two lines of output by adding `print(" ")` between the two commands. Here we find the inverse of a matrix:

```
matrix([[1,2],[3,4]])^(-1)
```

results in

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

Notice that you can enter a matrix by providing a list in brackets containing the rows as lists in brackets. (There are other ways; most things in *Sage* can be done in several different ways.) Let's try some calculus:

```
x = var('x')
f=integrate(sqrt(x)*sqrt(1+x), x)
```

That may surprise you: there would be no output at all. *Sage* has been told `x` is a variable and to assign the symbol `f` to the answer. It does, but doesn't print anything out. To see the answer you need to say

```
x = var('x')
f=integrate(sqrt(x)*sqrt(1+x), x)
print(f)
```

The result now is

```
1/4*((x + 1)^(3/2)/x^(3/2) + sqrt(x + 1)/sqrt(x))/((x + 1)^2/x^2
- 2*(x + 1)/x + 1) - 1/8*log(sqrt(x + 1)/sqrt(x) + 1)
+ 1/8*log(sqrt(x + 1)/sqrt(x) - 1)
```

That answer is very hard to read. It's good to know that you can also do it like this:

```
x = var('x')
f=integrate(sqrt(x)*sqrt(1+x), x)
show(f)
```

That gives something like:

$$\frac{\frac{(x+1)^{\frac{3}{2}}}{x^{\frac{3}{2}}}+\frac{\sqrt{x+1}}{\sqrt{x}}}{4\left(\frac{(x+1)^2}{x^2}-\frac{2\,(x+1)}{x}+1\right)}-\frac{1}{8}\log\left(\frac{\sqrt{x+1}}{\sqrt{x}}+1\right)+\frac{1}{8}\log\left(\frac{\sqrt{x+1}}{\sqrt{x}}-1\right)$$

There is also `latex(f)`, which is what I did to get the LaTeX code to typeset the result. While `show` produces output that is nicer to look at, the output of `print` is easier to cut-and-paste. In general, it's best to ask *Sage* to either `print` or `show` the outputs you want to see.

I should remark that there are subtle differences between writing `f=` as above and writing `f(x)=`. The Sage Tutorial is helpful on this issue.

As mentioned above, *Sage* likes to know in what ring it is working. Indeed, it will assume one if you don't give it one.

```
M=matrix([[1,2], [3,4]])
print M.base_ring()
```

results in

```
Integer Ring
```

That is, *Sage* has assumed your matrix lives in $M_2(\mathbb{Z})$. You can tell it otherwise by adding the name of a ring to the command that creates the matrix:

```
M=matrix(CC,[[1,2], [3,4]])
print M.base_ring()
print M
```

results in

```
Complex Field with 53 bits of precision
[1.00000000000000 2.00000000000000]
[3.00000000000000 4.00000000000000]
```

Notice that a complex number is $a+bi$ with $a, b \in \mathbb{R}$, so that both a and b will be printed as real numbers, i.e., as decimal expansions.

Many commands in *Sage* use the object-oriented `A.something()` format as above; sometimes you need to put something into the parentheses, but often they just need to be there. One advantage of *CoCalc* and the terminal interface is that if you type `A.` and then hit the Tab key, you will get a list of possible continuations. Here's a selection of matrix commands:

```
M=matrix(QQ,[[1,2], [3,4]])
print(M)
print("  ")
M.characteristic_polynomial()
print("  ")
M.column_space()
print("  ")
M.determinant()
print("  ")
M.eigenspaces_right()
```

The `QQ` tells *Sage* to think of this as a matrix with rational coefficients. The output is

```
[1 2]
[3 4]

x^2 - 5*x - 2

Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]

-2

[
(-0.3722813232690144?, Vector space of degree 2 and dimension 1
over Algebraic Field
User basis matrix:
[                   1 -0.6861406616345072?]),
(5.372281323269015?, Vector space of degree 2 and dimension 1
over Algebraic Field
User basis matrix:
[                 1 2.186140661634508?])
]
```

Without the `QQ`, the `column_space` function would return a free $\mathbb{Z}$-module of rank two rather than a vector space. But notice that when you ask for eigenspaces *Sage* will move to the real or complex numbers if necessary. It's also worth noticing that *Sage* can think of matrices acting on vectors on the right or on the left, so you should use things like `kernel_right()` rather than just `kernel()`.

That's probably enough to start. To learn more, start with the Tour of Sage, then go on to theSage Tutorial. See also *Sage for Undergraduates*, which is avaliable online as well as in print. There's a lot more helpful documentation online.