Algebra and Cryptography with SageMath

Sebastiano Tronto - sebastiano.tronto@uni.lu

2021-04-23

This lecture's notes are in a different format: the presentations for the LaTeX part were made with LaTeX, so this one is made with Sage, or rather with the Jupyter Notebook.

1 The Jupyter Notebook

Reference: [1]

The Jupyter Notebook is one of the default interfaces for SageMath, along with the command line interface. You can access it via web browser, but it is running locally on your device (notice the strange url: `http://localhost:8888/notebooks...`).

You can create a new notebook by clicking on `New > SageMath 9.2`. You can also create a Python 3 notebook to write Python code.

Jupyter saves and reads files in the `.ipynb` format. If you download the file for this lecture you can open it and follow the examples interactively.

1.1 Cells

The notebook contains one or more *interactive cells* that you can run, like this one below:

[2]:
```
# Exercise: modify this cell to use the print() command
2+2
2/5
```

[2]:
```
2/5
```

If you are reading this from Jupyter rather than from the pdf file, you can edit the cell above and run it again. You can also add more cells by selecting `Insert` from the menu bar.

Notice that only the last statement produces an output. You can force anything to be written as output with the `print()` command, which works like in Python. As an exercise, try to modify the cell above to provide more output!

1.2 Markdown

Markdown is a simple markup language - think of LaTeX or html, but much simpler. You can add text to your notebook with Markdown cells by selecting `Cell > Cell Type > Markdown`.

You can also include some LaTeX code in Markdown cells, with dollar signs $ or align environments:

$$\frac{(x+y)^2}{x+1} = \frac{x^2+y^2}{x+1}$$

When you are done writing a Markdown cell, you can run it to see the well-formatted text. To edit the text again, double-click on the cell. Try doing it now to fix the formula above!

2 Symbolic expressions

Reference: [2]

Now, let's get started with Sage. One thing you might want to do is manipulating symbolic expressions, like the following:

[3]:
```
f =  x^2 + 2*x - 5 == 0
solve(f,x)
```

[3]:
```
[x == -sqrt(6) - 1, x == sqrt(6) - 1]
```

Notice that the single = is part of an assignment, as in Python: we are *assigning* to the variable `f` the value `x^2 + 2*x - 5 >= 0`, which in this case is an equation, so it contains the symbol ==. Keep in mind the difference between the two!

Exercise: change the code above to solve the corresponding inequality $x^2 + 2x - 5 \geq 0$.

2.1 Mathematical variables

Last time we saw what *variables* are in Python, and that they are a little bit different from the *Mathematical variables* that you use in Mathematics. In Sage, both concepts are present, but they are still distinct. For example in the cell above `f` is a variable in the sense of computer science, while `x` is a Mathematical variable.

If you want to use Mathematical variables other than `x`, you first need to *declare* them with the `var()` command:

[14]:
```
var('y')
solve(y^2 + (x+1)*y - 2 == 0, y)
```

[14]:
```
[y == -1/2*x - 1/2*sqrt(x^2 + 2*x + 9) - 1/2, y == -1/2*x + 1/2*sqrt(x^2 + 2*x +
9) - 1/2]
```

Try removing the first line in the cell above and see what error you get!

Here is another example:

[16]:
```
var('a', 'b')
f = x^2+a*x+b
solve(f,x)
```

[16]:
```
[x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

Some common constants are already defined in Sage:

[17]:
```
e^(pi*I)
```

[17]:
```
-1
```

We will study symbolic expressions more in detail next time, in the context of calculus/analysis.

3 Basic rings and fields

References: [3] [4] [5]

As you should know, a *field* is a Mathematical structure with two operations, addition and multiplication, which respect certain rules (distributivity, associativity, commutativity...). Some examples of fields are the Rational numbers $\mathbb{Q}$, the Real numbers $\mathbb{R}$ and the Complex numbers $\mathbb{C}$, but there are many more. As you should also know, a *(commutative) ring* is like a field, except not all elements different from 0 need have a multiplicative inverse. For example the integers $\mathbb{Z} = \{\dots, -1, 0, 1, 2, \dots\}$ are a ring, but not a field.

These structures are already implemented in Sage. Some of the most common are listed in the following table:

Mathematical object	Math symbol	Sage name
Integers	$\mathbb{Z}$	`ZZ`
Rational numbers	$\mathbb{Q}$	`QQ`
Real numbers	$\mathbb{R}$	`RR`
Complex numbers	$\mathbb{C}$	`CC`
Integers modulo n	$\mathbb{Z}/n\mathbb{Z}$	`Integers(n)`
Finite fields	$\mathbb{F}_p$	GF(p)
...	...	...

If you write a number or an expression, Sage will figure out where it "lives", choosing the most restrictive interpretation possible. For example `3` will be interpreted to be an integer, even if it is also a rational number, a real number and a complex number.

3.1 Parents and coercion

Reference: [6]

You can check where an object "lives" with the `parent()` command. It works more or less like the Python command `type()`, but it gives a more Mathematically inclined answer. Check the reference link [6] above if you want more details.

[18]:
```
#Edit this cell to find out the type of other objects that we used
parent(3/5)
```

[18]:
```
Rational Field
```

Sometimes Sage does not give you the best possible interpretation, so you can force something to be interpreted as living in a smaller ring as follows:

[4]:
```
minus_one = e^(pi*I)
minus_one_coerced = ZZ(e^(pi*I))  # coercion
print(parent(minus_one))
print(parent(minus_one_coerced))
```

```
Symbolic Ring
Integer Ring
```

Remark. Notice that there is a fundamental difference between the rings `RR` and `CC` and all the others in the table above: the real and complex numbers are *approximated*.

[1]:
```
print(QQ(3))
print(RR(3))
```

```
3
3.00000000000000
```

You can also choose the precision of this approximation using the alternative name `RealField`.

[4]:
```
print(RR)
print(RealField(prec=1000))
```

```
Real Field with 53 bits of precision
Real Field with 1000 bits of precision
```

4 Polynomial rings

Reference: [7]

If you want to work with polynomials over a certain ring it is better to use this specific construction, rather than the symbolic expressions introduced above.

[5]:
```
polring.<x,y,z> = RR[]  # Alternative: polring.<x,y,z> = PolynomialRing(RR)
polring
```

[5]:
```
Multivariate Polynomial Ring in x, y, z over Real Field with 53 bits of
precision
```

You can use as many variables as you like, and you can replace `RR` with any ring. In the example above `polring` is just the name of the variable (in the computer science sense) associated with this polynomial ring.

4.1 Operations on polynomials

The usual Mathematical operations are available on polynomial rings, including Euclidean division `//` and remainder `%`. There is also the single-slash division `/`, but the result may not be a polynomial anymore.

Exercise: use the `parent()` command to find out what the quotient of two polynomials is.

Question: what happens if you remove the first line in the cell below? What if we used the variable y instead of x?

[6]:
```
polring.<x> = QQ[]
p = x^2 + 2*x - 3   # Don't forget * for multiplication!
q = p // (x+1)
r = p %  (x+1)
f = p /  (x+1)
print(q)
print(r)
print(f)
```

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

You can do more complex operations. Try out `roots()` and `factor` in the cell below.

Remark. Notice how the result can change substantially if you change the base ring.

Remark. Factorizations are a particular object in Sage. They are kinda like a list, but not really. You can get a list of pairs (factor, power) with `list(factor(f))`.

[7]:
```
polring_onevar.<t> = QQ[]

f = t^5 + t^4 - 2*t^3 - 2*t^2 - 3*t - 3
print(factor(f))
print(f.roots())  # Result: list of pairs (root,multiplicity)

polring_manyvar.<x,y,z> = QQ[]
factor(x*y+x)

# The following line gives an error, because the polynomial
# is understood to possibly have many variables:
#(x^2-1).roots()
```

```
(t + 1) * (t^2 - 3) * (t^2 + 1)
[(-1, 1)]
```

[7]:
```
(y + 1) * x
```

5 Matrices and vectors

References: [8], but in particular the subections [9] and [10]

In Sage you can easily manipulate matrices and vectors

[77]:
```
A = matrix([[1,2,3],[0,0,1],[4,-3,22/7]])
B = matrix([[1/2,0,0],[7,0,0],[1,1,1]])
v = vector([3,4,-1])
```

```
print(A, "\n")  # \n just means "newline"
print(B, "\n")
print(B*v, "\n")
print(A^2 + 2*B - A*B, "\n")

print("Rank of A =", rank(A)) # You can also use A.rank()
print("Rank of B =", rank(B))
```

```
[   1    2    3]
[   0    0    1]
[   4   -3 22/7]

[1/2   0   0]
[  7   0   0]
[  1   1   1]

(3/2, 21, 6)

[  -7/2    -10   80/7]
[    17     -4   15/7]
[ 241/7  -18/7 869/49]

Rank of A = 3
Rank of B = 2
```

Exercise: in the cell above, compute the determinant, inverse and characteristic polynomial of the matrix `A`. *Hint: look at the reference [10] above (the functions are listed in alphabetic order).*

As for polynomials, you can specify where a matrix or a vector lives

[57]:
```
M = matrix(CC, [[0,1],[1,0]])
parent(M)
```

[57]:
```
Full MatrixSpace of 2 by 2 dense matrices over Complex Field with 53 bits of
precision
```

You can also solve linear systems and compute eigenvalues and eigenvectors of a matrix

Warning. In linear algebra there are distinct concepts of *left* and *right* eigenvalues (and eigenvector). The one you know is probably that of **right** eigen-{value,vector}, that is an element λ of the base field and a non-zero vector $\mathbf{v}$ with $A\mathbf{v} = \lambda\mathbf{v}$. The other concept corresponds to the equality $\mathbf{v}^T A = \lambda\mathbf{v}$.

[60]:
```
A = Matrix(RR, [[sqrt(59),32],[-1/4,3]])
v = vector(RR, [3,0])
A.solve_right(v) # Solve Ax=v. Alternative: A \ v
```

[60]:
```
(0.289916349448506, 0.0241596957873755)
```

[64]:
```
A = Matrix(QQ, [[1,2],[3,4]])
A.eigenspaces_right() # Also: A.eigenvalues(), A.eigenvectors_right()
```

[64]:
```
[
(-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?])
]
```

We can also extract a specific submatrix by selecting only some rows and columns, with a syntax similar to that of Python's lists. Check out more examples in the reference [9] above, and try them in the cell below.

[94]:
```
A = MatrixSpace(ZZ, 7).random_element()
print(A, "\n")
print(A[1:3,2:5], "\n") # Rows from 1 to 3, columns from 2 to 5
print(A[0,0:], "\n")    # First row, all columns
print(A[[0,5,2],0:5])   # Rows 0, 5 and 2 (in this order) and columns 0 to 5
```

```
[-14   2   0  -1   1  -2  -1]
[  0  -8   0   9  -2  11   1]
[  0   3   1  -1   1   1 221]
[ -1   2   1 -25 -10   4   0]
[ -3   0   0   2  16  -1  -2]
[  1  -3   3 -41   1   0   0]
[ -2   1   0   0  -6   2  12]

[ 0  9 -2]
[ 1 -1  1]

[-14   2   0  -1   1  -2  -1]

[-14   2   0  -1   1]
[  1  -3   3 -41   1]
[  0   3   1  -1   1]
```

Exercise: write a sage function that computes the determinant of an $n \times n$ matrix $A = (a_{ij})$ using Laplace's rule by the first row, that is

$$\det A = \sum_{j=1}^{n} (-1)^j a_{0j} M_{0j}$$

where M_{0j} is the determinant of the $(n-1) \times (n-1)$ matrix obtained by removing the 0-th row and the j-th column from A.

[91]:
```
def my_det(A):
    if not A.is_square():
        print("Error: matrix is not square")

    n = A.nrows()  # size of the matrix

    # Continue from here!
```

6 Number Theory

Reference: [11]

Sage includes a large library of functions for computing with the integers, see the link above.

[8]:
```
n = 123456789
m = 987654321
p = 3607

print(factor(n))
print(is_prime(p))
print(p.divides(n))
print(euler_phi(m))
print(gcd(n, m))
print(lcm(n, m))
```

```
3^2 * 3607 * 3803
True
True
619703040
9
13548070123626141
```

6.1 Primes

Reference: [12]

The set of prime numbers is called `Primes()`. It is like an infinite list: for example you can get the one-millionth prime number or you can use this list to create other lists. You can also check what the first prime number larger than a given number is.

[9]:
```
PP = Primes()
print(PP)
print(PP[10], PP[10^6])
print(PP.next(44))

First_Thousand_Primes = PP[0:1000]
print([p for p in First_Thousand_Primes if p < 100 and p > 75])
```

```
Set of all prime numbers: 2, 3, 5, 7, ...
31 15485867
47
[79, 83, 89, 97]
```

6.2 The Chinese remainder theorem (CRT)

We say that two integers a and b are *congruent* modulo another integer $n > 0$ if they have the same remainder when divided by n. We denote this by $a \equiv b \pmod{n}$, or in Python/Sage syntax `a % n == b % n`.

The Chinese remainder theorem states that if $a, b \in \mathbb{Z}$ and $n, m \in \mathbb{Z}_{>0}$ are such that $\gcd(n, m) = 1$ then the system of congruences

$$\begin{cases} x \equiv a \pmod{n} \\ x \equiv b \pmod{m} \end{cases}$$

has exactly one solution modulo mn. This means that there is one and only one number x with $0 \leq x < mn$ such that $x \equiv a \pmod{n}$ and $x \equiv b \pmod{m}$.

The procedure to find such a number is not too hard to describe (you might see it in an algebra or number theory course), but it can be a bit long. Luckily, Sage can do this for you:

[10]:
```
a = 2
b = -1
n = 172
m = 799

if gcd(n,m) != 1:
    print("The numbers are not comprime, I can't solve this!")
else:
    x = crt(a, b, n, m)
    print(x, x%n, x%m)
```

```
74306 2 798
```

Exercise. There is a more general version of the Chinese remainder theorem which says that if $a_0, a_1, \ldots, a_k \in \mathbb{Z}$ and $n_0, n_2, \ldots, n_k \in \mathbb{Z}_{>0}$ are such that $\gcd(n_i, n_j) = 1$ for $i \neq j$, then the system of congruences

$$\begin{cases} x \equiv a_0 \pmod{n_0} \\ x \equiv a_1 \pmod{n_1} \\ \ldots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

has exactly one solution modulo $\prod_{i=0}^{k} n_i$. Use the `crt()` function to find a solution to such a system. *Hint: start by running the command `help(crt)`.

[127]:
```
#help(crt)
```

7 Cryptography: RSA

Cryptography is the discipline that studies methods to communicate secrets in such a way that any unauthorized listener would not be able to understand the message.

A simple cryptographic protocol could be changing every letter of your text following a fixed scheme (or *cypher*), for example by turning every A into a B, every B into a C and so on. However this is not a very secure method, for many reasons. One of them is that at some point the people who want to communicate need to agree on what method to use, and anyone listening to that conversation would be able to decypher every subsequent conversation. A public-key cryptographic protocol solves this problem.

7.1 Public-key cryptography

Public-key cryptographic protocols, such as RSA, work like this: there are two keys, a *private* key that is only known to person A (traditionally called Alice in every example), and a *public* key that does not need to be secret.

The public key is used to *encrypt* the message (that is to "lock" it, or "hyde" it), but one needs the private key to *decrypt* it. Imagine having two keys for your door, but one can only be used to lock it, while the other only to open it.

The message exchange works like this: suppose that person B (Bob) wants to send a secret message to Alice. Then Alice secretely generates a private and a public key and sends only the public one to Bob. Now Bob encrypts the message and sends it to Alice, who can use her private key to decrypt it. Even if Eve (short for *eavesdropper*, an unauthorized listener) listens to every message exchanged, she won't be able to decypher the secret: the private key has never left Alice's house!

Notice that such a protocol is *asymmetric*: if Alice wanted to send a secret to Bob in reply, Bob would need to generate a pair of keys of his own.

Let's see how we can do this in practice, using number theory!

7.2 RSA

As many other cryptography protocols, RSA is based on a Mathematical process that is easy to do in one direction, but very hard to invert. In this case the hard process is integer factorization, that is decomposing an integer number as a product of primes.

[2]:
```
p = 10000310001910004310005710069
q = 10014465531244957205984532844 3
n = p*q
print(is_prime(p), is_prime(q), is_prime(p*q))

# Use the command below to see how long it takes
#timeit("factor(n)", number=1, repeat=1)
```

```
True True False
```

In order to generate the keys, Alice picks a number n which is the product of two large primes p and q of more or less the same size. Finding such primes is relatively easy compared to factoring the number n she obtained. Then she computes the Euler totient $\varphi(n) = (p-1)(q-1)$ of n, which she can do because she knows that $n = pq$ - it would be impossible otherwise!

Then Alice can compute two integers (d, e) such that $de \equiv 1 \pmod{\varphi(n)}$. She will send the numbers n and d to Bob and keep e secret. In this case the public key is the pair (n, d), while e is the private key.

Of course, she does all of this using Sage!

[105]:
```
def two_large_primes():
    p, q = 0, 0
    # We make sure that they are different
    while p == q:
        p = Primes()[randint(10^6, 2*10^6)]
        q = Primes()[randint(10^6, 2*10^6)]
    return p, q

def random_unit_mod(N):
    R = Integers(N)
    d = R(0)
    # We make sure that it is invertible
    while not d.is_unit():
        d = R.random_element()
    return d

def Alice_generate_keys():
    p, q = two_large_primes()
    n = p*q
    phi_n = (p-1)*(q-1) # euler_phi(n) is slow!

    d = random_unit_mod(phi_n)
    e = d^-1
    return n, d, e

Alice_generate_keys()
```

[105]:
```
(419199544978969, 235530823946467, 80799425863927)
```

Now, how does Bob encrypt his message? Let's say he wants to send to Alice the number m with $1 < m < n$ (In practice he would like to send her some text with emojis, or maybe a voice message; but for computers everything is a number, and there are different ways to translate any sort of information to a number. He just chooses one of the many standard methods that already exist, no cryptography is needed in this step. If the message m is too long, he can split it up in some pieces and repeat the process multiple times.)

Now he computes $m^d \pmod{n}$ and sends it back to Alice.

[3]:
```
def Bob_encrypt(m, n, d):
    R = Integers(n)
    return R(m)^d  # Assume that n is large enough

message = 42424242
Bob_encrypt(message, 419199544978969, 235530823946467)
```

[3]:
```
149461597163501
```

Since $de \equiv 1 \pmod{\varphi(n)}$, it follows that $(m^d)^e \equiv m \pmod{n}$ (see Wikipedia: Euler's theorem). So for Alice it is very easy to get back the original message:

[108]:
```
def Alice_decrypt(m_encrypted, n, e):
    R = Integers(n)
    return R(m_encrypted)^e

Alice_decrypt(149461597163501, 419199544978969, 80799425863927)
```

[108]:
```
42424242
```

Another assumption on which RSA relies is that even if one knows $M = m^e$ and e, extracting the e-th root of M modulo n (and thus obtaining m) is very hard. Currently the best known way to do this is by factorizing n first, which is considered to be a very hard problem. However, there is no proof that faster algorithms can't be devised.

Moreover, one day we will overcome the current technological difficulties and quantum computers will be available. Quantum computers are not just "more powerful" than classical hardware, but they work based on completely different logical foundations and they make the factorization problem much easier to solve: for example Shor's algorithm takes advantage of this different logic and can factorize numbers quickly, if run on a quantum computer.

To this day the largest number factorized with a quantum computer is $21 = 3 \times 7$. Nonetheless, quantum-safe cryptography protocols (i.e. based on problems that are hard to solve also with quantum computers) have already been developed.