# 1 Sage

Sage is an open source computer algebra package. It can be downloaded for free from `www.sagemath.org/` or it can be accessed directly online at the website `https://sagecell.sagemath.org/`. The computer computations in this book can be done in Sage, especially by those comfortable with programming in python. In the following, we give examples of how to do some of the basic computations. Much more is possible. See `www.sagemath.org/` or search the Web for other examples. Another good place to start learning Sage in general is [Bard] (there is a free online version).

**Shift ciphers.**

Suppose you want to encrypt the plaintext `This is the plaintext` with a shift of 3. We first encode it as an alphabetic string of capital letters with the spaces removed. Then we shift each letter by 3 positions:

```
S=ShiftCryptosystem(AlphabeticStrings())
P=S.encoding("This is the plaintext")
C=S.enciphering(3,P);C
```

When this is evaluated, we obtain the ciphertext

```
WKLVLVWKHSODLQWHAW
```

To decrypt, we can shift by 23 or do the following:

```
S.deciphering(3,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Suppose we don't know the key and we want to decrypt by trying all possible shifts:

```
S.brute_force(C)
```

Evaluation yields

```
0:  WKLVLVWKHSODLQWHAW,
1:  VJKUKUVJGRNCKPVGZV,
2:  UIJTJTUIFQMBJOUFYU,
3:  THISISTHEPLAINTEXT,
4:  SGHRHRSGDOKZHMSDWS,
5:  RFGQGQRFCNJYGLRCVR,
6:  etc.

24:  YMNXNXYMJUQFNSYJCY,
25:  XLMWMWXLITPEMRXIBX
```

**Affine ciphers.**

Let's encrypt the plaintext `This is the plaintext` using the affine function $3x + 1 \bmod 26$:

```
A=AffineCryptosystem(AlphabeticStrings())
P=A.encoding("This is the plaintext")
C=A.enciphering(3,1,P);C
```

When this is evaluated, we obtain the ciphertext

```
GWZDZDGWNUIBZOGNSG
```

To decrypt, we can shift by 23 or do the following:

```
A.deciphering(3,1,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

We can also find the decryption key:

```
A.inverse_key(3,1)
```

This yields

```
(9, 17)
```

Of course, if we "encrypt" the ciphertext using $9x+17$, we obtain the plaintext:

```
A.enciphering(9,17,C)
```

Evaluate to obtain

```
THISISTHEPLAINTEXT
```

**Vigenère ciphers.**

Let's encrypt the plaintext `This is the plaintext` using the keyword `ace` (that is, shifts of 0, 2, 4). Since we need to express the keyword as an alphabetic string, it is efficient to add a symbol for these strings:

```
AS=AlphabeticStrings()
V=VigenereCryptosystem(AS,3)
K=AS.encoding("ace")
P=V.encoding("This is the plaintext")
C=V.enciphering(K,P);C
```

The "3" in the expression for `V` is the length of the key. When the above is evaluated, we obtain the ciphertext

```
TJMSKWTJIPNEIPXEZX
```

To decrypt, we can shift by $0, 24, 22$ (= `ayw`) or do the following:

```
V.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

Now let's try the example from Section 2.3. The ciphertext can be cut and pasted from ciphertexts.m in the MATLAB files (or, with a little more difficulty, from the Mathematica or Maple files). A few control symbols need to be removed in order to make the ciphertext a single string.

```
vvhq="vvhqwvvrhmusgjgthkihtssejchlsfcbgvwcrlryqtfs . . . czvile"
```

(We omitted part of the ciphertext in the above in order to save space.) Now let's computed the matches for various displacements. This is done by forming a string that displaces the ciphertext by $i$ positions by adding $i$ blank spaces at the beginning and then counting matches.

```
for i in range(0,7):
C2 = [" "]*i + list(C)
count = 0
for j in range(len(C)):
if C2[j] == C[j]:
count += 1
print i, count
```

The result is

```
0 331
1 14
2 14
3 16
4 14
5 24
6 12
```

The `331` is for a displacement of 0, so all 331 characters match. The high number of matches for a displacement of 5 suggests that the key length is 5. We now want to determine the key.

First, let's choose every fifth letter, starting with the first (counted as 0 for Sage). We extract these letters, put them in a list, then count the frequencies.

```
V1=list(C[0::5])
dict((x, V1.count(x)) for x in V1)
```

The result is

```
C: 7,
D: 1,
E: 1,
F: 2,
G: 9,
I: 1,
J: 8,
K: 8,
N: 3,
P: 4,
Q: 5,
R: 2,
T: 3,
U: 6,
V: 5,
W: 1,
Y: 1
```

Note that A, B, H, L, M, O. S, X, Z do not occur among the letters, hence are not listed. As discussed in Subsection 2.3.2, the shift for these letters is probably 2. Now, let's choose every fifth letter, starting with the second (counted as 1 for Sage). We compute the frequencies:

```
V2=list(C[1::5])
dict((x, V2.count(x)) for x in V2)
```

```
A: 3,
B: 3,
C: 4,
F: 3,
G: 10,
H: 6,
M: 2,
O: 3,
P: 1,
Q: 2,
R: 3,
S: 12,
T: 3,
U: 2,
V: 3,
W: 3,
Y: 1,
Z: 2
```

As in Subsection 2.3.2, the shift is probably 14. Continuing in this way, we find that the most likely key is $\{2, 14, 3, 4, 18\}$, which is `codes`. let's decrypt:

```
V=VigenereCryptosystem(AS,5)
K=AS.encoding("codes")
P=V.deciphering(K,C);P
```

```
THEMETHODUSEDFORTHEPREPARATIONANDREADINGOFCODEMES . . .  ALSETC
```

**Hill ciphers.**

Let's encrypt the plaintext `This is the plaintext` using a $3\times 3$ matrix. First, we need to specify that we are working with such a matrix with entries in the integers mod 26:

```
R=IntegerModRing(26)
M=MatrixSpace(R,3,3)
```

Now we can specify the matrix that is the encryption key:

```
K=M([[1,2,3],[4,5,6],[7,8,10]]);K
```

Evaluate to obtain

```
[ 1 2  3]
[ 4 5  6]
[ 7 8 10]
```

This is the encryption matrix. We can now encrypt:

```
H=HillCryptosystem(AlphabeticStrings(),3)
P=H.encoding("This is the plaintext")
C=H.enciphering(K,P);C
```

If the length of the plaintext is not a multiple of 3 (= the size of the matrix), then extra characters need to be appended to achieve this. When the above is evaluated, we obtain the ciphertext

```
ZHXUMWXBJHHHLZGVPC
```

Decrypt:

```
H.deciphering(K,C)
```

When this is evaluated, we obtain

```
THISISTHEPLAINTEXT
```

We could also find the inverse of the encryption matrix mod 26:

K1=K.inverse();K1

This evaluates to

```
[ 8 16  1]
[ 8 21 24]
[ 1 24  1]
```

When we evaluate

```
H.enciphering(K,C):C
```

we obtain

```
THISISTHEPLAINTEXT
```

**LFSR.**

Consider the recurrence relation $x_{n+4} \equiv x_n + x_{n+1} + x_{n+3} \bmod 2$, with initial values $1, 0, 0, 0$. We need to use 0s and 1s, but we need to tell Sage that they are numbers mod 2. One way is to define "`o`" (that's a lower-case "oh") and "`l`" (that's an "ell") to be 0 and 1 mod 2:

```
F=GF(2)
o=F(0); l=F(1)
```

We also could use `F(0)` every time we want to enter a `0`, but the present method saves some typing. Now we specify the coefficients and initial values of the recurrence relation, along with how many terms we want. In the following, we ask for 20 terms:

```
s=lfsr_sequence([l,l,o,l],[l,o,o,o],20);s
```

This evaluates to

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

Suppose we are given these terms of a sequence and we want to find what recurrence relation generates it:

```
berlekamp_massey(s)
```

This evaluates to

```
x^4 + x^3 + x + 1
```

When this is interpreted as $x^4 \equiv 1 + 1x + 0x^2 + 1x^3 \bmod 2$, we see that the coefficients 1, 1, 0, 1 of the polynomial give the coefficients of recurrence relation. In fact, it gives the smallest relation that generates the sequence.

*Note:* Even though the output for `s` has 0s and 1s, if we try entering the command `berlekamp_massey([1,1,0,1,1,0])` we get `x^3 - 1`. If we instead enter `berlekamp_massey([l,l,o,l,l,o])`, we get `x^2 + x + 1`. Why? The first is looking at a sequence of integers generated by the relation $x_{n+3} = x_n$ while the second is looking at the sequence of integers mod 2 generated by $x_{n+2} \equiv x_n + x_{n+1} \bmod 2$. Sage defaults to integers if nothing is specified. But it remembers that the 0s and 1s that it wrote in `s` are still integers mod 2.

**Number Theory.**

To find the greatest common divisor, type the following first line and then evaluate:

```
gcd(119, 259)
7
```

To find the next prime greater than or equal to a number:

```
next_prime(1000)
1009
```

To factor an integer:

```
factor(2468)
2^2 * 617
```

Let's solve the simultaneous congruences $x \equiv 1 \pmod 5$, $x \equiv 3 \pmod 7$:

```
crt(1,3,5,7)
31
```

To solve the three simultaneous congruences $x \equiv 1 \pmod 5, x \equiv 3 \pmod 7, x \equiv 0 \pmod{11}$:

```
a= crt(1,3,5,7)
crt(a,0,35,11)
66
```

Compute 123ˆ456 (mod 789):

```
mod(123,789)^456
699
```

Compute $d$ so that $65d \equiv 1 \pmod{987}$:

```
mod(65,987)^(-1)
410
```

Let's check the answer:

```
mod(65*410, 987)
1
```

**RSA.**

Suppose someone unwisely chooses RSA primes $p$ and $q$ to be consecutive primes:

```
p=nextprime(987654321*10^50+12345); q=nextprime(p+1)
n=p*q
```

Let's factor the modulus

$$n = pq = 975461057789971041000000000000000000000000000000000000002507 \\ 6543210190000000000000000000000000000000000000000161156941$$

without using the `factor` command:

```
s=N(sqrt(n), digits=70)
p1=next_prime(s)
p1, q1
(98765432100000000000000000000000000000000000000000000012773,
98765432100000000000000000000000000000000000000000000012617)
```

Of course, the fact that $p$ and $q$ are consecutive primes is important for this calculation to work. Note that we needed to specify 70-digit accuracy so that round-off error would not give us the wrong starting point for looking for the next prime. These factors we obtained match the original $p$ and $q$, up to order:

```
p, q
(98765432100000000000000000000000000000000000000000000012617,
98765432100000000000000000000000000000000000000000000012773)
```

**Lagrange interpolation.**

Suppose we want to find the polynomial of degree at most 3 that passes through the points $(1,1),(2,2),(3,21),(5,12)$ mod the prime $p = 37$. We first need to specify that we are working with polynomials in $x$ mod 37. Then we compute the polynomial:

```
R=PolynomialRing(GF(37),"x")
f=R.lagrange_polynomial([(1,1),(2,2),(3,21), (5,12)]);f
```

This evaluates to

```
22*x^3 + 25*x^2 + 31*x + 34
```

If we want the constant term:

```
f(0)
43
```

**Elliptic curves.**

Let's set up the elliptic curve $y^2 \equiv x^3 + 2x + 3 \bmod 7$:

```
E=EllipticCurve(IntegerModRing(7),[2,3])
```

The entry `[2,3]` gives the coefficients $a, b$ of the polynomial $x^2 + ax + b$. More generally, we could use the vector `[a,b,c,d,e]` to specify the coefficients of the general form $y^2 + axy + cy \equiv x^3 + bx^2 + dx + e$. We could also use `GF(7)` instead of `IntegerModRing(7)` to specify that we are working mod 7. We could replace the `7` in `IntegerModRing(7)` with a composite modulus, but this will sometimes result in error messages when adding points (this is the basis of the elliptic curve factorization method).

We can list the points on $E$:

```
E.points()
[(0:1:0), (2:1:1), (2:6:1), (3:1:1), (3:6:1), (6:0:1)]
```

These are given in projective form. The point `(0:1:0)` is the point at infinity. The point `(2:6:1)` can also be written as `[2, 6]`. We can add points:

```
E([2,1])+E([3,6])
(6 :  0 :  1)
```

```
E([0,1,0])+E([2,6])
(2 :  6 :  1)
```

In the second addition, we are adding the point at infinity to the point $(2, 6)$ and obtaining $(2, 6)$. This is an example of $\infty + P = P$. We can multiply a point by an integer:

```
5*E([2,1])
(2 :  6 :  1)
```

We can list the multiples of a point in a range:

```
for i in range(10):
   print(i,i*E([2,6]))

(0, (0 :  1 :  0))
(1, (2 :  6 :  1))
(2, (3 :  1 :  1))
(3, (6 :  0 :  1))
(4, (3 :  6 :  1))
(5, (2 :  1 :  1))
(6, (0 :  1 :  0))
(7, (2 :  6 :  1))
(8, (3 :  1 :  1))
(9, (6 :  0 :  1))
```

The indentation of the `print` line is necessary since it indicates that this is iterated by the `for` command. To count the number of points on $E$:

```
E.cardinality()
6
```

Sage has a very fast point counting algorithm (due to Atkins, Elkies, and Schoof; it is much more sophisticated than listing the points, which would be infeasible). For example,

```
p=next_prime(10^50)
E1=EllipticCurve(IntegerModRing(p),[2,3])
n=E1.cardinality()
p, n, n-(p+1)
(100000000000000000000000000000000000000000000000151,
```

```
99999999999999999999999999112314733133761086232032,
-887685266866238913768120)
```

As you can see, the number of points on this curve (the second output line) is close to $p+1$. In fact, as predicted by Hasse's theorem, the difference $n-(p+1)$ (on the last output line) is less in absolute value than $2\sqrt{p} \approx 2 \times 10^{25}$.