The Electronic Journal of Mathematics and Technology, Volume 6, Number 1, ISSN 1933-2823

Implementing the RSA cryptosystem with Maxima CAS

Juan Monterde
monterde@uv.es
Facultat de Matemàtiques
Universitat de València
Spain

José A. Vallejo
jvallejo@fc.uaslp.mx
Facultad de Ciencias
Universidad Autónoma de San Luis Potosí
México

Abstract

We show a possible implementation of the RSA algorithm with Maxima CAS. Theoretical background and numerous examples are given, along with the code. We also implement a simplified version of the digital signature method and comment on some ideas for using this material in the classroom.

1 Introduction

A basic problem in cryptography is the reduction of the number of operations needed to deal with big numbers. This is usually done through the use of congruences. An example is the RSA algorithm, in which calculations are done modulo n, where n is the product of two (big) primes p and q. In practical applications, the size in bits of n is 1024, 2048 or even 4096 (corresponding to 309, 617 and 1234 decimal digits, respectively). Thus, the computational cost if we were to use these numbers "as they are" would be prohibitive.

One possibility is to make use of the Chinese remainder theorem, so the calculations can be transferred from the ring $\mathbb{Z}_n$ to the ring $\mathbb{Z}_p \times \mathbb{Z}_q$. The sum of the bit length of p and q is the bit length of n, so p and q can be taken considerably smaller than n, and the calculations are speeded up notably. The reason behind these properties is that the Chinese remainder theorem allows one to represent big integers as $n-$tuples of long (Euclidean) divisions remainders. In this form, operations such as sums or multiplications can be done in parallel in real time. However, comparison or division are not so direct, and the inverse problem (given n factoring it into p and q) is computationally intractable for large n. Precisely, this fact lies at the foundation of the RSA algorithm [7]. However, when using RSA encryption one has to compute expressions of the form $a^b \bmod c$, with large a, b. For these, the Chinese remainder theorem is not so practical, and other algorithms are preferred, such as the repeated squaring method. From a mathematical point of view, the other essential ingredients of the RSA cryptosystem are Fermat's Little Theorem and its generalization, Euler's theorem on congruences. These facts will be reviewed in Section 2.

Our aim in this paper is to show how to implement the RSA cryptosystem with the Maxima CAS [4], which we do in Section 3. This choice of software is motivated by pedagogical as well as practical reasons: the source code is freely available, the students can get a copy at no cost (thus, they can work at home with the same program they use at the classroom), the developers are very active and accessible, and the commands and syntax are very intuitive (some of the Maxima syntax is merely a translation of the plain English used to describe the mathematical operations, such as `integrate(x^2,x,0,1)`: "integrate x^2 with respect to x between 0 and 1"). These reasons almost coincide with the ones expressed by A. McAndrew in [5], a paper which describes in great detail the use of Maxima and Axiom in teaching a course on cryptography (for an interesting alternative, see also [6], where SAGE is used instead of Maxima). Indeed, the present paper is based on a $10-$hour summer course given by one of the authors (JM) to students of mathematics, physics and engineering, ranging from first-year to post-graduates. Our conclusions are also similar to those of McAndrew in [5] (Maxima is particularly well suited for basic cryptography), so we refer to his paper for a justification of the educational aspects and focus our attention on the technical details: the mathematical foundation of the RSA cryptosystem and its practical implementation on Maxima.

In Section 4 we show how to apply the RSA techniques to guarantee the authorship of a message, through the use of digital signatures. We include sample codes for all the functions described in the text along with examples of their usage, so the paper is relatively self-contained. Information about the use of Maxima can be found in the page `http://maxima.sourceforge.net/documentation.html`.

Section 5 offers some comments about how to use the present material in actual cryptography courses.

Some technical details about the software used:

The version of Maxima was 5.24.0, with the interface wxMaxima (version 11.04.0). Both can be downloaded for free from `http://maxima.sourceforge.net/download.html` and `http://andrejv.github.com/wxmaxima`, respectively.

The code examples assume that the messages are written in pure ASCII (not extended), to avoid some problems with the XML rendering of non printable characters in wxMaxima. When expressing the path to a file, we assume that the user has a Linux® or Mac OSX® system. For MS Windows®, little changes are required.

The source code for the programs presented in this paper, suppressed outputs and some related material, can be found at `http://galia.fc.uaslp.mx/~jvallejo/Software.html`.

2 Algebraic preliminaries

2.1 Modular arithmetic

We begin by recalling Euclid's algorithm. To compute the greatest common divisor (gcd) of two integers $a, b \in \mathbb{Z}$, first divide a by b to find the quotient q_1 and the remainder r_1:

$$a = q_1 \cdot b + r_1.$$

Then, repeat the process substituting a by b and b by r_1:

$$b = q_2 \cdot r_1 + r_2.$$

Repeat again, this time using r_1 and r_2, to find:

$$r_1 = q_3 \cdot r_2 + r_3.$$

The algorithm continues this way until one of the r_i divides r_{i-1}. In this case, $\gcd(a,b) = r_{i-1}$. Note that the process ends after a finite number of steps, as in each one of them we have $r_i < r_{i-1}$.

Example 1 *To compute* $\gcd(1547, 560)$*, we proceed as follows:*

$$\begin{aligned} 1547 &= 2 \cdot 560 + 427; \\ 560 &= 1 \cdot 427 + 133; \\ 427 &= 3 \cdot 133 + 28; \\ 133 &= 4 \cdot 28 + 21; \\ 28 &= 1 \cdot 21 + 7. \end{aligned}$$

As $7|21$*, we are done:* $\gcd(1547, 560) = 7$.

Maxima has a function to compute the gcd:

```
(%i1) gcd(1547,560);
```

```
(%o1)  7
```

A basic fact from Number Theory is the following: If $d = \gcd(a,b)$, with $a > b$, there exist two integers u, v such that

$$d = ua + vb.$$

This is called the Bézout identity (see [2], Section 1.2).

Example 2 *We have just seen that* $\gcd(1547, 560) = 7$*. Let us find* u, v *such that* $7 = u \cdot 1547 + v \cdot 560$*. We can use the intermediate steps in Euclid's algorithm, but reversing direction (from end to start):*

$$\begin{aligned} 7 &= 28 - 1 \cdot 21 \\ &= 28 - (133 - 4 \cdot 28) = -133 + 5 \cdot 28 \\ &= -133 + 5 \cdot (427 - 3 \cdot 133) = 5 \cdot 427 - 16 \cdot 133 \\ &= 5 \cdot 427 - 16 \cdot (560 - 427) = -16 \cdot 560 + 21 \cdot 427 \\ &= -16 \cdot 560 + 21 \cdot (1547 - 2 \cdot 560) = 21 \cdot 1547 - 58 \cdot 560. \end{aligned}$$

Thus,

$$7 = 21 \cdot 1547 - 58 \cdot 560.$$

The Maxima command `gcdex` does the same:

```
(%i2) gcdex(1547,560);
```

```
(%o2)  [21, -58, 7]
```

As a direct application of the Bézout identity, we have that the invertible elements for the product in[1] $\mathbb{Z}/m\mathbb{Z}$, are precisely those coprimes with m. In other words: the integers a for which there exists a b such that $ab \equiv 1 \pmod{m}$, are precisely those a satisfying $\gcd(a,m) = 1$.

[1]This is the ring formed by the equivalence classes determined by the binary relation "congruence modulo m": two integers $a, b \in \mathbb{Z}$ are congruent modulo m iff their difference $a - b$ is a multiple of m. In this case we write $a \equiv b \pmod{m}$.

Example 3 *Modulo* 841*, the integer* 160 *is invertible. Indeed,* $841 = 29^2$ *while* $160 = 2^5 \cdot 5$*. Thus,* $\gcd(841, 160) = 1$*. Its inverse is* 205*, since*

$$160 \cdot 205 - 1 = 32800 - 1 = 32799 = 39 \cdot 841.$$

Maxima can compute the inverse of n modulo m directly with the command `inv_mod`:

```
(%i3) inv_mod(160,841);
```

```
(%o3)  205
```

The properties of modular arithmetic are similar to those of integer arithmetic. To name a few (see [1], Chapter 5):

1. $a \equiv a \pmod{m}$, for all $a \in \mathbb{Z}$.
2. $a \equiv b \pmod{m}$ if and only if $b \equiv a \pmod{m}$, for all $a, b \in \mathbb{Z}$.
3. If $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$, for all $a, b, c \in \mathbb{Z}$.
4. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a + c \equiv b + d \pmod{m}$ and $a \cdot c \equiv b \cdot d \pmod{m}$, for all $a, b, c, d \in \mathbb{Z}$.
5. If $a \equiv b \pmod{m}$, then if $a \equiv b \pmod{d}$ for each divisor of m, d.
6. If $a \equiv b \pmod{m}$, and $a \equiv b \pmod{n}$, with m, n coprimes, then $a \equiv b \pmod{m \cdot n}$.

We now recall a classical result (see [1], Chapter 5). Fermat's Little Theorem says that if p is a prime number, then for any integer a we have

$$a^p \equiv a \pmod{p},$$

and, for every integer a not divisible by p:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{1}$$

An immediate consequence is the following. If a is not divisible by p and $n \equiv m \pmod{(p-1)}$, then

$$a^n \equiv a^m \pmod{p}.$$

Example 4 *Let us find the last digit of* $2^{1000000}$ *when written in base* 7*. This is just the remainder of* $2^{1000000}$ *modulo* 7*. In order to apply the result above, we must reduce the exponent. If we take* $p = 7$*,* $n = 1000000$*, we must find a suitable* m *such that* $1000000 \equiv m \pmod{6}$*. Since* $1000000 = 166666 \cdot 6 + 4$*, this means that we can take* $m = 4$*. Applying now the corollary to Fermat's Little Theorem, we find*

$$2^{1000000} \equiv 2^4 = 16 \equiv 2 \pmod{7}.$$

Thus, the last digit of $2^{1000000}$ *when written in base* 7 *is* 2*.*

2.2 The Chinese remainder theorem

The original form of the theorem, appearing in a book written by the Chinese mathematician Qin Jiushao and published in 1247, is a result related to systems of congruences. It is possible to find a precedent in the *Sunzi suanjing*, a book from the third century written by Sun Zi:

> Han Xing asks how many soldiers are in his army. If you let them parade in rows of 3 soldiers, two soldiers will be left. If you let them parade in rows of 5, 3 will be left, and in rows of 7, 2 will be left. How many soldiers are there?.

The modern formulation of the problem is the following. Let $m_1, m_2, ..., m_k$ be integers that are greater than one and pairwise coprime, and let $a_1, a_2, ..., a_k$ be any integers. Then there exists an integer x such that $x \equiv a_i \pmod{m_i}$ for each $i \in \{1, 2, ..., k\}$. Furthermore, for any other integer y that satisfies all the congruences, $y \equiv x \pmod{M}$ where $M = m_1 \cdots m_k$. Note that there is only one solution in $\{0, 1, ..., M\}$.

The proof of the theorem (which can be found in Theorem 5.26 of [1]) gives an algorithm to find x. The following code implements it in Maxima (it returns the lowest positive solution modulo M):

Beginning of code

```
chinese_remainder(a,k):=
 block([K,L,x],
    K:makelist(apply("*",delete(k[i],k)),i,1,length(k)),
    L:makelist(first(gcdex(K[i],k[i])),i,1,length(k)),
    x:mod(sum(a[i]*K[i]*L[i],i,1,length(k)),apply("*",k)),
    x
 );
```

End of code

To use it, just write `chinese_remainder([a1,...,ar],[m1,...,mr])`. For the Han Xing's example:

```
(%i4)  chinese_remainder([2,3,2],[3,5,7]);
```

(%o4) 23

A small army, indeed. Surely, this is one example in which the solution is not the one in $\{0, 1, ..., M\}$ (as in this case Han Xing would have not needed to ask for the number of soldiers, being easy to count them by himself), but one of the infinite numbers congruent with 23 modulo $M = 7 \cdot 5 \cdot 3 = 105$, that is: 128, 233, etcetera.

2.3 Euler's theorem

Consider Example 4 again. What if we are asked for the last digit in base 77?. We can not apply Fermat's theorem here, because 77 is composite. Euler found a generalization of Fermat's theorem to this case, introducing his totient function $\varphi(n)$ (which gives the number of positive integers less than or equal to n that are coprime to n).

In Maxima we have the command `totient`:

```
(%i5) totient(77);
```

```
(%5)   60
```

Now, Euler's theorem says that for every $a \in \mathbb{Z}$:

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Example 5 *Let us compute the last digit of* $2^{1000000}$ *in base* 77. *As* $\varphi(77) = 60$*, we know that* $2^{60} \equiv 1 \pmod{77}$. *As* $1000000 = 16666 \cdot 60 + 40$*, then* $2^{1000000} \equiv 2^{40} \pmod{77}$. *Now, it is easy to compute* $2^{40} \pmod{77}$. *For instance*

$$2^{10} = 1024 = 13 \cdot 77 + 23 \equiv 23 \pmod{77},$$

and $2^{40} = 2^{10 \cdot 4} = 23^4 \equiv 23 \pmod{77}$. *Thus, the last digit of* $2^{1000000}$ *in base* 77 *is* 3.

2.4 The Maxima function power_mod

As we have seen in Example 5, a basic computation one often encounters in modular arithmetic is the modular exponentiation when both the exponent and the modulus are very large. As mentioned, the Chinese remainder theorem or the Euler's theorem can be of some help to compute modular exponentiation. Nevertheless, there is in Maxima a simple way to compute such exponentiation without using these results.

The modular power function `power_mod(a,b,c)` gives exactly the result of $a^b \pmod{c}$ for $b > 0$:

```
(%i6) power_mod(2,40,77);
```

```
(%o6)   23
```

However, `power_mod` is much more efficient than the Chinese remainder or Euler's theorems, because it avoids generating the full form of a^b. The algorithm to compute modular exponentiation follows the repeated squaring method: Instead of repeated multiplication of a by itself, what this algorithm does is to reduce any partial result modulo c, immediately after a multiplication is performed. In that way we never encounter any integers greater than c^2 (see [8] for details on this method). In our implementation of the RSA cryptosystem we will use the `power_mod` function, and so, the repeated squaring method implicitly.

3 The RSA cryptosystem

3.1 Caesar cipher

The first documented cryptosystem is very simple. It just replaces each letter in the plain text with the one sitting three positions down the alphabet, for example, we would replace A by D, B by E, and so on (the last three letters, X, Y, Z, are replaced by A, B, C, respectively). It seems that this method was used by Julius Caesar to send military instructions to his generals.

This idea can be translated to numbers, for an easier use. If we replace each letter of the alphabet by a natural number (adding the space between words character) as in

	a	b	c	d	e	f	g	h	i	j	k	l	···	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	···	18	19	20	21	22	23	24	25	26

then the encryption function is simply

$$f(x) = x + 3 \pmod{27},$$

while the decryption function is, obviously, the inverse:

$$g(x) = f^{-1}(x) = x - 3 \pmod{27}.$$

These simple properties make the Caesar cipher very easy to use. However, they are also the origin of its weakness. To break a cryptosystem means to deduce the encryption/decryption functions from a sample of ciphertext (the encrypted message); in this case, the method of attack is a simple frequency analysis of word repetitions (in English, for instance, the most repeated number corresponds to the most used letter, the *e*, the second is the *t*, and so on).

Several schemes were devised to improve the Caesar cipher. For instance, one could try more complicated bijections, such as

$$f(x) = a \cdot x + b \pmod{27},$$

with a an integer such that $\gcd(a, 27) = 1$, and $a, b \in \{1, 2, ..., 26\}$. Another possibility consists in taking groups of contiguous letters, instead of one by one; for example, digraphs (groups of two letters) or trigraphs (groups of three letters). In this case, an encryption function could have the form

$$f(x) = a \cdot x + b \pmod{m},$$

where $a, b \in \{1, 2, ..., m - 1\}$ and m is a number big enough to allow for the enumeration (in a one-to-one and onto manner) of every possible combination of three characters. Both parameters, a and m, are necessary for the encryption as well as the decryption processes, and are called the *key* of the cryptosystem.

Anyway, these systems are too weak to be used in practice. They can be broken in a number of ways, very easily with the use of a modern computer. We have cited them here just for the sake of illustration of the concepts involved.

3.2 Public key cryptosystems

A revolution happened when the so called public key cryptosystems appeared, in the late seventies. These systems are based on the following fact: there are bijective (one-to-one and onto) functions for which the calculation of the inverse is very difficult, computationally impossible. Then, we can make publicly available the keys used to encrypt messages whenever the keys used for decryption are to be found by one of these functions. One example is the factorization of a given number into a product of primes, and this fact is the foundation of the RSA method (see [7]).

In a public key cryptosystem, each user has a public key, that is, an encryption function f_{pub}, and a private key, a decryption function f_{pri}, which are inverses one of another: $f_{pri} = f_{pub}^{-1}$. Thus, if a user A wants to send a message (say, "*text*", translated to numbers as in the preceding section to "2052420") to a user B, she must look in the public keys listing the one corresponding to B (say, f_{pub}^{B}), apply it to the message (obtaining $f_{pub}^{B}(2052420)$), and send it to B. Even in the unfortunate event that someone intercepts the ciphered text, only B knows the decryption function. Indeed, user B can read the original message by just applying $f_{pri}^{B} = \left(f_{pub}^{B}\right)^{-1}$ to $f_{pub}^{B}(2052420)$; thus:

$$f_{pri}^{B}\left(f_{pub}^{B}(2052420)\right) = \left(f_{pub}^{B}\right)^{-1}\left(f_{pub}^{B}(2052420)\right) = 2052420.$$

3.3 The RSA algorithm

As stated in the preceding subsection, to implement the RSA cryptosystem we only need to give the pair of functions (f_{pub}, f_{pri}) for each user, A, of the system. The procedure is the following:

1. User A chooses two large prime numbers, p_A and q_A.
2. Calculate the product $n_A = p_A \cdot q_A$.
3. Evaluate the totient function[2] on n_A, $\Phi(n_A)$
4. Choose a number $e_A \in \{1, 2, ..., \Phi(n_A)\}$, coprime with $\Phi(n_A)$.

With these choices, $\gcd(e_A, \Phi(n_A)) = 1$, so e_A is an invertible element of the ring $\mathbb{Z}/\Phi(n_A)\mathbb{Z}$. Let d_A be an integer such that its equivalence class is the inverse of that of e_A, that is,

$$e_A \cdot d_A \equiv 1 \pmod{\Phi(n_A)}.$$

5. The encryption function, $f_{pub}^{A} : \mathbb{Z}/n_A\mathbb{Z} \to \mathbb{Z}/n_A\mathbb{Z}$, is given by

$$f_{pub}^{A}(m) = m^{e_A} \pmod{n_A}.$$

[2]Note that in this case, with n_A *semiprime* (that is, the product of two primes), we have

$$\Phi(n_A) = (p_A - 1) \cdot (q_A - 1) = n_A - p_A - q_A - 1.$$

6. The decryption function, f^A_{pri}, is the inverse of f^A_{pub}, that is:

$$f^A_{pri}(m) = m^{d_A} \pmod{n_A}.$$

It is instructive to check that these functions are indeed inverses of each other.

Theorem 6 *For every $m \in \mathbb{Z}/n_A\mathbb{Z}$,*

$$f^A_{pri}(f^A_{pub}(m)) \equiv f^A_{pri}(m^{e_A}) \equiv m^{e_A \cdot d_A} \equiv m \pmod{n_A}.$$

Proof. For simplicity, we will drop the subscripts A. We can assume that m is not divisible[3] by p. Then, by Fermat's Little Theorem (1),

$$m^{p-1} \equiv 1 \pmod{p}.$$

Let us take the power of $q-1$ on both sides of this equation:

$$m^{(p-1)\cdot(q-1)} = \left(m^{p-1}\right)^{q-1} \equiv 1^{q-1} = 1 \pmod{p}.$$

A similar computation, interchanging p and q, gives

$$m^{(p-1)\cdot(q-1)} = \left(m^{q-1}\right)^{p-1} \equiv 1^{p-1} = 1 \pmod{q}.$$

We can apply the property 6 of modular arithmetic, to get

$$m^{(p-1)\cdot(q-1)} = 1 \pmod{pq}.$$

Note that for any integer k we will also have

$$m^{k\cdot(p-1)\cdot(q-1)} = \left(m^{(p-1)\cdot(q-1)}\right)^k \equiv 1^k = 1 \pmod{pq}.$$

Now, recall that $e \cdot d \equiv 1 \pmod{\Phi(n) = (p-1)\cdot(q-1)}$. That means

$$e \cdot d = 1 + k \cdot (p-1) \cdot (q-1).$$

Thus:

$$m^{e\cdot d} = m^{1+k\cdot(p-1)\cdot(q-1)} = m \cdot m^{k\cdot(p-1)\cdot(q-1)} \equiv m \cdot 1 = m \pmod{pq},$$

so $m^{e\cdot d} \equiv m \pmod{n}$, as we intended to prove. ■

Why is this method secure?. In the RSA cryptosystem, the *public key* of user A is the pair of integers (e_A, n_A), while her *private key* is the pair (d_A, n_A). To find the private key knowing the public key is equivalent to computing the integer d_A knowing e_A and n_A. Thus, if we want to break the cryptosystem, we must first calculate $\Phi(n_A) = (p_A - 1)(q_A - 1)$, that is, we must factor n_A as the product of two primes. This is an extremely difficult problem from the computational point of view; indeed, a joint effort of several researchers using hundreds of dedicated computers only could factor a $232-$digit number after two years of computation (see [3]). Note that this number corresponds to a $768-$bit encryption, the problem of factoring a $1024-$bit key is a thousand times harder! Thus, although breaking the RSA cryptosystem is theoretically possible, it is practically impossible with our current state of knowledge.

[3]That p divide m is highly improbable, given the magnitude order of the prime p.

3.4 Implementation in Maxima CAS

The following function generates a list (e, n, d) suitable for use in the RSA algorithm.

```
                        Beginning of code
(
 load(distrib)
);

generate_random_key(l):=
block([p,q,n,r,e,d,K],
p:next_prime(floor(float(random_continuous_uniform(2^l,2^(l+1))))),
q:next_prime(floor(float(random_continuous_uniform(2^l,2^(l+1))))),
n:p*q,
r:(p-1)*(q-1),
e:next_prime(floor(float(random(floor(r/2))))),
d:inv_mod(e,r),
K:[e,n,d],
K
);
                          End of code
```

To use it write `generate_random_key(l)` where l determines the number of digits of the integers e, n, d (they will be pseudo-randomly generated between 2^l and 2^{l+1}). A suggested value (to speed up computations while keeping practical interest for applications) is $l = 125$. Note that the construction of e is based on the obvious consequence of the Prime Number Theorem[4] that the number of primes between $r/2$ and r (for large r) is approximately $r/\log(r/2)$, so these are relatively abundant.
For example:

```
(%i7) key:generate_random_key(125);

(%o7) [671258217454686505293403265373669737061769832160804966205145423006710890659,
6524380046621073928456206976859918286968722163998541593398840097117692758107,
4117416708352497196549341588732382167090260042357291528574523538009688319619]
```

Once we have the list (e, n, d) as the output of `generate_random_key`, we can create our public key (e, n):

```
(%i8) public_key:rest(key,-1);

(%o8) [671258217454686505293403265373669737061769832160804966205145423006710890659,
```

[4]Let $\pi(x)$ be the function giving the number of primes less than or equal to x. Then, the Prime Number Theorem says that

$$\lim_{x\to+\infty} \frac{\pi(x)}{x/\ln(x)} = 1.$$

$6524380046621073928456206976859918286968722163998541593398840097117692758107]$

and our private key (d, n):

```
(%i9) private_key:reverse(rest(key,1));
```

(%o9) $[4117416708352497196549341588732382167090260042357291528574523538009688319619,$
$6524380046621073928456206976859918286968722163998541593398840097117692758107]$

The next function `rsa_encrypt` reads a text from a file and encrypts it in blocks of length k.

```
Beginning of code
(
 load(stringproc)
);

file_to_string(u):=
 block([a,i:0,text_string,tmp],
    a:openr(u),
    while stringp(tmp[i]:readline(a)) do i:i+1,
    text_string:simplode(makelist(tmp[j],j,0,i-1)," "),
    text_string
);

rsa_encrypt_console(s,k,P):=
 block([tmp,added_char,encrypted_text,encrypted_text_added],
  local(fill_string,string_to_number,rsa_encode),
  tmp:smake(k-(slength(s)-k*floor(slength(s)/k)),ascii(126)),
  added_char:[slength(tmp)],
  fill_string(s,k):=concat(s,tmp),
  string_to_number(s):=sum(
    (cint(charat(s,i))-32)*95^(i-1),i,1,slength(s)
    ),
  rsa_encode(a,b,c):=power_mod(a,b,c),
  encrypted_text:makelist(
     rsa_encode(
      string_to_number(
       substring(fill_string(s,k), 1+k*(i-1), 1+k*i)
       ),P[1],P[2]
      ), i, 1, slength(fill_string(s,k))/k
    ),
  encrypted_text_added:append(added_char,encrypted_text)
);

rsa_encrypt_file(s,k,P,p):=
 block([LL],
    LL:rsa_encrypt_console(s,k,P),
    stringout(p,LL)
);

rsa_encrypt(q,k,P,[p]):=
 block([t],
    t:file_to_string(q),
    if p=[] then rsa_encrypt_console(t,k,P)
    else rsa_encrypt_file(t,k,P,first(p))
);
```

End of code

The result can be shown in the console or, if a path to a file is given, written to that file. For the purpose of this example, we have stored the following text in `/root/hysteria.txt`:

> As she laughed I was aware of becoming involved in her laughter and being part of it, until her teeth were only accidental stars with a talent for squad-drill. I was drawn in by short gasps, inhaled at each momentary recovery, lost finally in the dark caverns of her throat, bruised by the ripple of unseen muscles. An elderly waiter with trembling hands was hurriedly spreading a pink and white checked cloth over the rusty green iron table, saying: "If the lady and gentleman wish to take their tea in the garden, if the lady and gentleman wish to take their tea in the garden ..." I decided that if the shaking of her breasts could be stopped, some of the fragments of the afternoon might be collected, and I concentrated my attention with careful subtlety to this end.

We can encrypt the text in blocks of length[5] $k = 20$ with the following commands (the output is suppressed because of its length):

```
(%i10) load(rsa_encrypt)$

(%i11) rsa_encrypt("/root/hysteria.txt",20,public_key);
```

Or we can redirect the output to a file, in this example `/root/hysteria-encrypted.txt`:

```
(%i12) rsa_encrypt("/root/hysteria.txt",20,public_key,
       "/root/hysteria-encrypted.txt");
```

(%o12) $/root/hysteria-encrypted.txt$

The function given below, `rsa_decrypt`, reads encrypted text from a file and decrypts it.

Beginning of code

```
(
 load(basic)
);
```

[5]Here, the particular value $k = 20$ is chosen just as an example. In Section 5, $k = 3$ is taken instead. What one should keep in mind when choosing the value of k is that the numerical representation of each block must have a value less than the modulus if the message is to be decrypted successfully.

```
rsa_decrypt_console(M,k,Q):=
 block([clean_message],local(number_to_string,rsa_decode),
    number_to_string(z,k):=makelist(
     ascii(mod(floor(z/(95^(i-1))),95)+32),i,1,k
     ),
    rsa_decode(a,b,c):=power_mod(a,b,c),
    clean_message:simplode(
     makelist(
      simplode(
       number_to_string(
        rsa_decode(M[i],Q[1],Q[2]),k
        )
       )
     ,i,1,length(M))
    ),
    clean_message
);

rsa_decrypt(s,k,Q,[p]):=
 block([prelist,AA,BB,len],
    prelist:read_list(s,comma),
    AA:rest(rest(prelist,-2),1),
    BB:pop(AA),
    len:slength(rsa_decrypt_console(AA,k,Q)),
    if p=[] then
     substring(rsa_decrypt_console(AA,k,Q),1,len-(BB-1))
    else
     stringout(
      first(p),substring(
       rsa_decrypt_console(AA,k,Q),1,len-(BB-1)
       )
      )
);
```

End of code

Again, the function can show the clean text in the console or, if a path to a file is given, to store the decrypted text in that file. In the example, we decode the text that was previously encrypted and stored in `/root/hysteria-encrypted.txt`. Note also that the length of the blocks ($k = 20$) is passed as an argument:

```
(%i13) rsa_decrypt("/root/hysteria-encrypted.txt",20,private_key);
```

(%o13) *As she laughed I was aware of becoming involved in her laughter and being part of it, until her teeth were only accidental stars with a talent for squad-drill. I was drawn in by short gasps, inhaled at each momentary recovery, lost finally in the dark caverns of her throat, bruised by the ripple of unseen muscles. An elderly waiter with trembling hands was hurriedly spreading a pink and white checked cloth over the rusty green iron table, saying: "If the lady and gentleman wish to take their* *tea in the garden, if the lady and gentleman wish to take their tea in the garden ..." I decided that if the shaking of her breasts could be stopped, some of the fragments of the afternoon might be collected, and I concentrated my attention with careful subtlety to this end.*

To redirect the output to a file, we give the path to that file, as in:

```
(%i14) rsa_decrypt("/root/hysteria-encrypted.txt",20,
       private_key,"/root/hysteria-decrypted.txt");
```

(%o14) $/root/hysteria - decrypted.txt$

4 Digital signatures

Although the RSA cryptosystem, as described in the preceding sections, has some nice properties (it is reliable and secure), it also has a serious drawback: it does not provide any method to check the authenticity of the message; in other words, we do not know if the person claiming to be the sender of the message is actually that person, as the recipient gets the message encrypted with his own public key, which is available to everybody.

However, using the same ideas of the RSA method, a solution to this problem can be given. It consists in the following: the sender (user A) signs the message in such a way that anybody (for instance, the receiver, user B) can deduce that only A is able to produce that signature. To implement this "digital signature", a possible method[6] follows these steps: user A adds to the message she wants to send (say *text*) her signature, another piece of text such as *I am user A*, that identifies her, encrypted with her *private* key, to get $text\ + f^A_{pri}(I\ am\ user\ A)$. Then, the sender (user A) encrypts the result with the public key of the recipient B, $f^B_{pub}(text\ + f^A_{pri}(I\ am\ user\ A))$. When user B decrypts the message, he will find two pieces, the message *text* and the encrypted chunk $f^A_{pri}(I\ am\ user\ A)$. To be sure that the message actually comes from A, all he has to do is to apply her public key to it, as the result is then

$$f^A_{pub}(f^A_{pri}(I\ am\ user\ A)) = I\ am\ user\ A.$$

Note that the idea behind the digital signature is that only A is able to produce a text such that, when applying f^A_{pub}, the result is legible.

The following function, `rsa_encrypt_ds`, is a slight modification of `rsa_encrypt` capable of including a signature. It takes a file with path a containing some text to encrypt, an integer k (the chunk length), a list $P = [e, n]$ (the public key of the recipient, user B), a list $Q = [d, n]$ (the private key of the sender, user A), a path q to a file storing a signature, and, optionally, a path p to a file where the output is written as its arguments. When p is absent, the output is written in the Maxima console.

Beginning of code

```
load("stringproc");
```

[6]The procedure we describe here is not the one used in an actual situation, where A first gets a digest of the message, with a certain hash function such as MD5, and then encrypt this digest with her own private key. But the example we describe retains the basic idea while being far more easy to implement.

```
load("basic");

file_to_string(u):=
 block([a,i:0,tmp,text_string],
    a:openr(u),
    while stringp(tmp[i]:readline(a)) do i:i+1,
    text_string:simplode(makelist(tmp[j],j,0,i-1)," "),
    text_string
);

rsa_encrypt_console(s,k,P):=
block([tmp,added_char,encrypted_text,encrypted_text_added],
local(fill_string,string_to_number,rsa_encode),
tmp:smake(k-(slength(s)-k*floor(slength(s)/k)),ascii(126)),
added_char:[slength(tmp)],
fill_string(s,k):=concat(s,tmp),
string_to_number(s):=sum((cint(charat(s,i))-32)*95^(i-1),i,1,slength(s)),
rsa_encode(a,b,c):=power_mod(a,b,c),
encrypted_text:makelist(
        rsa_encode(
         string_to_number(
          substring(fill_string(s,k), 1+k*(i-1), 1+k*i)
                ),P[1],P[2]
              ), i, 1, slength(fill_string(s,k))/k
             ),
encrypted_text_added:append(added_char,encrypted_text)
);

rsa_encrypt_ds(a,k,P,Q,q,[p]):=
 block([t,v,L,M,l,m,S],
    t:file_to_string(a),
    v:file_to_string(q),
    L:rsa_encrypt_console(t,k,P),
    M:rsa_encrypt_console(v,k,Q),
    l:length(L),
    m:length(M),
    S:[l,m,L,M],
    if p=[] then S
    else stringout(first(p),S)
);
```

End of code

To use this function, we prepare a file `/root/signature.txt`, with the following content:

T. S. Eliot
Prufrock and Other Observations
1917

Then we can make the following to encrypt and sign the file `/root/hysteria.txt` using the following public-private keys for users A and B:

```
(%i15) public_keyA;
```

(%o15) [1650059172879816791260876129953328603907602666352067869083685909461393736909,
3708150289863286994535783817852033198889096484604423097296248436592325166991]

```
(%i16) private_keyA;
```

(%o16) [379527932593144827812732692164393824004791535303438931019254825655253733685,
3708150289863286994535783817852033198889096484604423097296248436592325166991]

```
(%i17) public_keyB;
```

(%o17) [146879466713946901827530582303760087656241476871443053164266198403926458463,
3040216715027414087962274325454256602848802215847097074502322427275375547837]

```
(%i18) private_keyB;
```

(%o18) [2843378208968070601455100396976753790872008736270032956043745072607798170907,
3040216715027414087962274325454256602848802215847097074502322427275375547837]

```
(%i19) rsa_encrypt_ds("/root/hysteria.txt",20,public_keyB,
       private_keyA,"/root/signature.txt")$
```

(The output here was suppressed, it can be seen at `http://galia.fc.uaslp.mx/~jvallejo/Software.html`). Or, we can redirect the output to a file, as before. In this example, the file is `/root/hysteria-signed.txt`:

```
(%i20) rsa_encrypt_ds("/root/hysteria.txt",20,public_keyB,
       private_keyA,"/root/signature.txt",
       "/root/hysteria-signed.txt");
```

(%o20) $/root/hysteria - signed.txt$

To decrypt the file *and* the signature, we have the following function, `rsa_decrypt_ds`:

Beginning of code

```
(
 load(stringproc)
);

rsa_decrypt_console(M,k,T):=
 block([clean_message],local(number_to_string,rsa_decode),
    number_to_string(z,k):=makelist(
        ascii(mod(floor(z/(95^(i-1))),95)+32),i,1,k
        ),
    rsa_decode(a,b,c):=power_mod(a,b,c),
    clean_message:simplode(
    makelist(simplode
     (number_to_string(rsa_decode(M[i],T[1],T[2]),k)),i,1,length(M)
     )
    ),
    clean_message
);


rsa_decrypt_ds(s,k,y,z,[p]):=
block([prelist,ML,SL,BB,CC,lenm,lens],
prelist:read_list(s,comma),
ML:rest(rest(prelist,-prelist[3]-5),4),
SL:rest(rest(prelist,-3),prelist[2]+6),
BB:pop(ML),
CC:pop(SL),
lenm:slength(rsa_decrypt_console(ML,k,z)),
lens:slength(rsa_decrypt_console(SL,k,y)),
if p=[] then
simplode([substring(rsa_decrypt_console(ML,k,z),1,lenm-(BB-1)),"
--- Signature begins ---
",substring(rsa_decrypt_console(SL,k,y),1,lens-(CC-1))]," ")
else
stringout(
first(p), simplode([substring(rsa_decrypt_console(ML,k,z),1,lenm-(BB-1)
),"
--- Signature begins ---
",substring(rsa_decrypt_console(SL,k,y),1,lens-(CC-1))]," "))
);
```

End of code

Applying it to the output of the function rsa_encrypt_ds, we have:

```
(%i21) rsa_decrypt_ds("/root/hysteria-signed.txt",20,
       public_keyA,private_keyB);
```

(%o21) *As she laughed I was aware of becoming involved in her laughter and being part of it, until her teeth were only accidental stars with a talent for squad-drill. I was drawn in by short gasps, inhaled at each momentary recovery, lost finally in the dark caverns of her throat, bruised by the ripple of unseen muscles. An elderly waiter with trembling hands was hurriedly spreading a pink and white checked cloth over the rusty green iron table, saying: "If the lady and gentleman wish to take their tea in the garden, if the lady and gentleman wish to take their tea in the garden ..." I decided that if the shaking of her breasts could be stopped, some of the fragments of the afternoon might be collected, and I concentrated my attention with careful subtlety to this end. — Signature begins — T. S. Eliot Prufrock and Other Observations 1917*

Of course, we can redirect the output to a file, instead than the console:

```
(%i22) rsa_decrypt_ds("/root/hysteria-signed.txt",20,public_keyA,
       private_keyB,"hysteria-signed-decrypted.txt");
```

(%o22) $/root/hysteria-signed-decrypted.txt$

Note that in order to get the text in the file `/root/hysteria-signed-decrypted.txt` nicely formatted, one has to write the code above for `rsa_encrypt_ds` with the appropriate new line breaks and long line expressions, as it contains some string arguments in `simplode` that will be written "as they are"; the version appearing here does not have these features to fit within the page width.

5 Working at the classroom

When working out examples with the students at the classroom, the teacher can produce a message, sign it and send it encrypted to the students for decrypting it. To do this efficiently, it is a good idea to have a file with the names of the students ordered (following any criteria), and another one where their public keys are stored *with the same ordering*. In this way, the teacher can send the same message to different students, digitally signed with his own private key, but encrypted with the public key corresponding to each student.

Suppose for example that we have seven students: Cordelia, Curan, Edmund, Fool, Goneril, Lear, and Regan. We create a text file with each name in one line, alphabetically ordered, and store it at `/root/students.txt`. Also, we let the students create their own public-private keys, and ask them to hand us the public pairs. All these public pairs are written on another text file, each pair e, n in one line (separated by commas). For instance, we could have:

```
1119871,3560093
623521,2735539
745231,2543987
1321063,3038941
140759,1981211
1019747,2155823
609043,2827757
```

We store the file at `/root/public-keys.txt`. For later use, note that the public key of the fourth student, Fool, is $[1321063, 3038941]$. Actually, his private one is $[734395, 3038941]$ (both were generated with `generate_random_key(10)`).
The teacher wants to send an encrypted message, contained in the file `/root/KingLear.txt`. His own digital signature:

> The teacher
> William Shakespeare
> King Lear

is contained in another file, `/root/digsig.txt`. Finally, the public-private keys of the teacher are:

```
(%i23) teacher_public;
```

(%o23) $[429881, 2294543]$

```
(%i24) teacher_private;
```

(%o24) $[872729, 2294543]$

With these data, the following function `rsa_encrypt_group` signs the message with the teacher's signature, encrypts the signature with the teacher's private key, the whole message with each student's public key and generates a file with the name of the student containing the encrypted message, stored in a path given by the user (in the example, `/root/messages/`). Note also the k value for the text chunks ($k = 3$ in the example).

Beginning of code

```
(
 load(numericalio)
);

rsa_encrypt_group(s,k,t,u,r,Q,q):=
 block([students,publickeys,name,path],
  students:read_list(t),
  publickeys:read_nested_list(u,comma),
  for j:1 thru length(students) do
   (name:students[j],
    path:simplode([r,string(name),".txt"]),
    rsa_encrypt_ds(s,k,publickeys[j],Q,q,path)
   )
);
```

End of code

```
(%i25) rsa_encrypt_group("/root/KingLear.txt",3,
       "/root/students.txt","/root/public-keys.txt",
       "/root/messages/",teacher_private,"/root/digsig.txt");
```

(%o25) *done*

Now, we have seven files in `/root/messages/`. These are: `Cordelia.txt`, `Curan.txt`, `Edmund.txt`, `Fool.txt`, `Goneril.txt`, `Lear.txt`, and `Regan.txt`. Suppose that the teacher sends each one to its corresponding recipient. When Fool receives his message, he can decrypt it using the function `rsa_decrypt_ds` of the preceding section, as he knows the teacher's public key (to decipher the signature) and his own private key (to decipher the message):

```
(%i26) rsa_decrypt_ds("/root/messages/Fool.txt",3,
       teacher_public,[734395,3038941]);
```

(%o26) *This is the excellent foppery of the world, that, when we are sick in fortune,–often the surfeit of our own behavior,–we make guilty of our disasters the sun, the moon, and the stars: as if we were villains by necessity; fools by heavenly compulsion; knaves, thieves, and treachers, by spherical predominance; drunkards, liars, and adulterers, by an enforced obedience of planetary influence; and all that we are evil in, by a divine thrusting on: an admirable evasion of whoremaster man, to lay his goatish disposition to the charge of a star! — Signature begins — The teacher William Shakespeare King Lear*

6 Acknowledgements

JM has been partially supported by a Spanish Ministry of Science and Innovation grant MTM2009-08933. JAV was supported by a CONACyT grant CB-J2-2007 code 78791.

References

[1] T. Apostol: *Introduction to analytic number theory*. Springer Verlag, New York (1976).

[2] G. A. Jones, and J. M. Jones: *Elementary Number Theory*. Springer Verlag, London (1998).

[3] T. Kleinjung et al.: *Factorization of a* $768-$*bit RSA modulus*, version 1.4, February 18 (2010). Available on-line at http://eprint.iacr.org/2010/006.pdf.

[4] Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.24.0 (2011). http://maxima.sourceforge.net.

[5] A. McAndrew: *Teaching cryptography with open-source software*. SIGCSE Bull. **40** 1 (2008) $325 - 329$.

[6] A. McAndrew: *Introduction to Cryptography with Open-Source Software*. CRC Press, Boca Ratón (2011).

[7] R. Rivest, A. Shamir; L. Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM **21** 2 (1978) $120 - 126$.

[8] B. Schneier: *Applied Cryptography: Protocols, Algorithms, and Source Code in C* (2nd ed.). Wiley (1996).