Number Theory and Cryptography using PARI/GP

Minh Van Nguyen
nguyenminh2@gmail.com
25 November 2008

This article uses PARI/GP to study elementary number theory and the RSA public key cryptosystem. Various PARI/GP commands will be introduced that can help us to perform basic number theoretic operations such as greatest common divisor and Euler's phi function. We then introduce the RSA cryptosystem and use PARI/GP's built-in commands to encrypt and decrypt data via the RSA algorithm. It should be noted that our brief exposition on RSA and cryptography is for educational purposes only. Readers who require further detailed discussions are encouraged to consult specialized texts such as [2, 5, 6].

1 What is PARI/GP?

PARI/GP [3] is a computer algebra system that is geared towards research in number theory. It can be used as a powerful desktop calculator, as a tool to help undergraduate students study number theory, or as a programming environment for studying number theoretic cryptosystems. PARI/GP is available free of charge and can be downloaded from the following website:

`http://pari.math.u-bordeaux.fr`

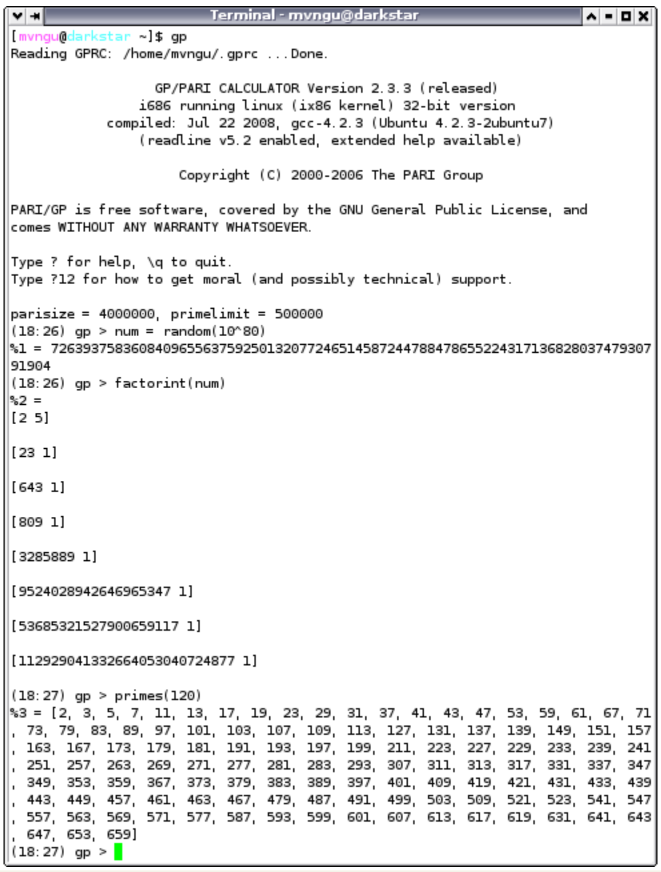
The default interface to PARI/GP is command line based, but there are graphical user interfaces to the software as well. Figure 1 shows sample PARI/GP sessions under Linux and Windows.

2 Elementary number theory

This section reviews basic concepts from elementary number theory, including the notion of primes, greatest common divisors, congruences and Euler's phi function. The number theoretic concepts and PARI/GP commands introduced will be referred to in later sections when we present the RSA algorithm.

Prime numbers

Public key cryptography uses many fundamental concepts from number theory, such as prime numbers and greatest common divisors. A positive integer $n > 1$ is said to be *prime* if its factors are exclusively 1 and itself. In PARI/GP, we can obtain the first 50 prime numbers using the command `primes`:


Terminal - mvngu@darkstar
```
[mvngu@darkstar ~]$ gp
Reading GPRC: /home/mvngu/.gprc ...Done.

                  GP/PARI CALCULATOR Version 2.3.3 (released)
                i686 running linux (ix86 kernel) 32-bit version
            compiled: Jul 22 2008, gcc-4.2.3 (Ubuntu 4.2.3-2ubuntu7)
                (readline v5.2 enabled, extended help available)

                     Copyright (C) 2000-2006 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 ?12 for how to get moral (and possibly technical) support.

parisize = 4000000, primelimit = 500000
(18:26) gp > num = random(10^80)
%1 = 72639375836084096556375925013207724651458724478847865522431713682803747930
791904
(18:26) gp > factorint(num)
%2 =
[2 5]

[23 1]

[643 1]

[809 1]

[3285889 1]

[9524028942646965347 1]

[5368532152790065911 7 1]

[112929041332664053040724877 1]

(18:27) gp > primes(120)
%3 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71
, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157
, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241
, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347
, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439
, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547
, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643
, 647, 653, 659]
(18:27) gp >
```

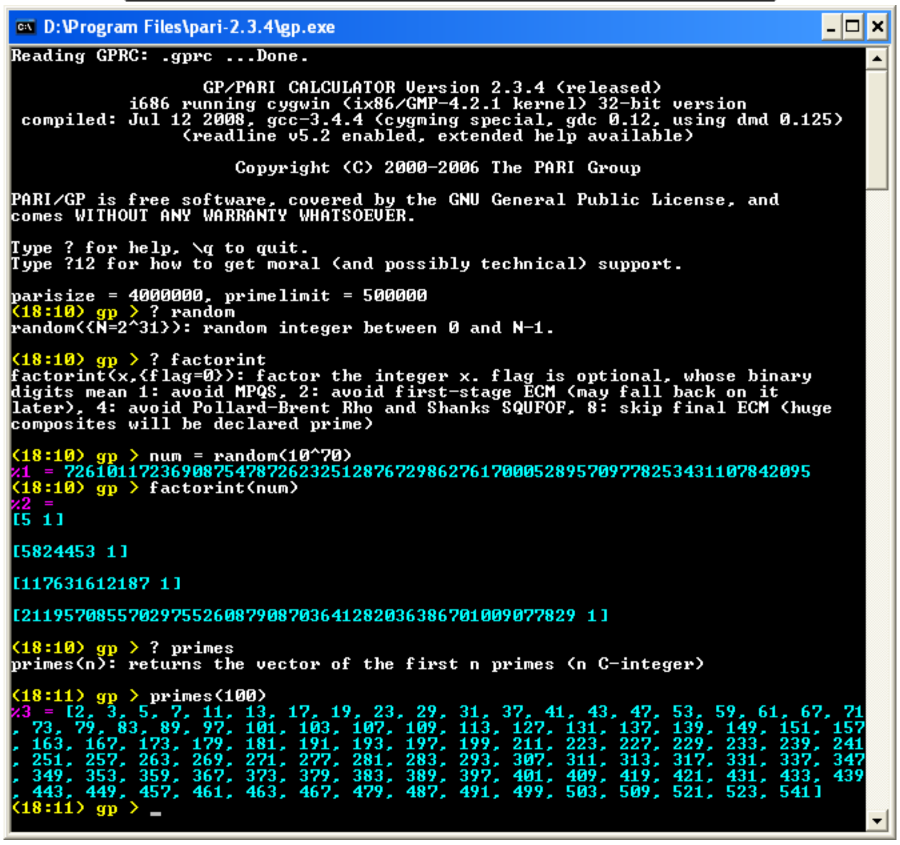

D:\Program Files\pari-2.3.4\gp.exe
```
Reading GPRC: .gprc ...Done.

                  GP/PARI CALCULATOR Version 2.3.4 (released)
           i686 running cygwin (ix86/GMP-4.2.1 kernel) 32-bit version
 compiled: Jul 12 2008, gcc-3.4.4 (cygming special, gdc 0.12, using dmd 0.125)
                (readline v5.2 enabled, extended help available)

                     Copyright (C) 2000-2006 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 ?12 for how to get moral (and possibly technical) support.

parisize = 4000000, primelimit = 500000
(18:10) gp > ? random
random({N=2^31}): random integer between 0 and N-1.

(18:10) gp > ? factorint
factorint(x,{flag=0}): factor the integer x. flag is optional, whose binary
digits mean 1: avoid MPQS, 2: avoid first-stage ECM (may fall back on it
later), 4: avoid Pollard-Brent Rho and Shanks SQUFOF, 8: skip final ECM (huge
composites will be declared prime)

(18:10) gp > num = random(10^70)
%1 = 7261011723690875478726232512876729862761700052895709778253431107842095
(18:10) gp > factorint(num)
%2 =
[5 1]

[5824453 1]

[117631612187 1]

[2119570855702975526087908703641282036386701009077829 1]

(18:10) gp > ? primes
primes(n): returns the vector of the first n primes (n C-integer)

(18:11) gp > primes(100)
%3 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71
, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157
, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241
, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347
, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439
, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
(18:11) gp > _
```

Figure 1: PARI/GP sessions under Linux (top) and Windows (bottom).

PARI/GP
```
gp > primes(50)
%1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199,
211, 223, 227, 229]
```

Greatest common divisors

Let $\mathbb{Z}$ be the set of integers. If $a, b \in \mathbb{Z}$, then the *greatest common divisor* (GCD) of a and b is the largest positive factor of both a and b. We use $\gcd(a, b)$ to denote this largest positive factor. The GCD of any two distinct primes is 1, and the GCD of 18 and 27 is 9. We can use the command `gcd` to compute the GCD of any two integers:

PARI/GP
```
gp > gcd(3, 59)
%2 = 1
gp > gcd(18, 27)
%3 = 9
```

If $\gcd(a, b) = 1$, we say that a is *coprime* (or relatively prime) to b. In particular, $\gcd(3, 59) = 1$ so 3 is coprime to 59.

Congruences

When one integer is divided by a non-zero integer, we usually get a remainder. For example, upon dividing 23 by 5, we get a remainder of 3; when 8 is divided by 5, the remainder is again 3. The notion of congruence helps us to describe the situation in which two integers have the same remainder upon division by a non-zero integer. Let $a, b, n \in \mathbb{Z}$ such that $n \neq 0$. If a and b have the same remainder upon division by n, then we say that a is *congruent* to b modulo n and denote this relationship by

$$a \equiv b \pmod{n}$$

This definition is equivalent to saying that n divides the difference of a and b, that is, $n \mid (a - b)$. Thus $23 \equiv 8 \pmod{5}$ because when both 23 and 8 are divided by 5, we end up with a remainder of 3. The command `Mod` allows us to compute such a remainder:

PARI/GP
```
gp > Mod(23, 5)
%4 = Mod(3, 5)
gp > Mod(8, 5)
%5 = Mod(3, 5)
```

Euler's phi function

Consider all the integers from 1 to 20, inclusive. List all those integers that are coprime to 20. In other words, we want to find those integers n, where $1 \leq n \leq 20$, such that $\gcd(n, 20) = 1$. The latter task can be easily accomplished with some PARI/GP programming:

PARI/GP
```
gp > for (n = 1, 20, if (gcd(n,20) == 1, print1(n, " ")))
1 3 7 9 11 13 17 19
```

The above programming statement can be saved into a text file named, for example, `/home/mvngu/totient.gp`, organizing it as in Listing 1 to enhance readability.

```
for (n = 1, 20,
    if (gcd(n,20) == 1,
        print1(n, " ")
    )
)
```

Listing 1: Finding all integers $1 \leq n \leq 20$ such that $\gcd(n, 20) = 1$.

The command `\r` can be used to read the file containing our programming statements into PARI/GP:

PARI/GP
```
gp > \r /home/mvngu/totient.gp
1 3 7 9 11 13 17 19
```

From the latter list, there are 8 integers in the closed interval $[1, 20]$ that are coprime to 20. Without explicitly generating the list

$$1 \quad 3 \quad 7 \quad 9 \quad 11 \quad 13 \quad 17 \quad 19 \tag{1}$$

how can we compute the number of integers in $[1, 20]$ that are coprime to 20? This is where Euler's phi function comes in handy. Let $n \in \mathbb{Z}$ be positive. Then *Euler's phi function* counts the number of integers a, with $1 \leq a \leq n$, such that $\gcd(a, n) = 1$. This number is denoted by $\varphi(n)$. The command `eulerphi` implements Euler's phi function. To compute $\varphi(20)$ without explicitly generating the list (1), we proceed as follows:

PARI/GP
```
gp > eulerphi(20)
%27 = 8
```

3 How to keep a secret?

Cryptography is the science (some might say art) of concealing data. Imagine that we are composing a confidential email to someone. Having written the email, we can send it in one of two ways. The first, and usually convenient, way is to simply press the send button and not care about how our email will be delivered. Sending an email in this manner is similar to writing our confidential message on a postcard and post it without enclosing our postcard inside an envelope. Anyone who can access our postcard can see our message. On the other hand, before sending our email, we can scramble the confidential message and then press the send button. Scrambling our message is similar to enclosing our postcard inside an envelope. While not 100% secure, at least we know that anyone wanting to read our postcard has to open the envelope.

In cryptography parlance, our message is called *plaintext*. The process of scrambling our message is referred to as *encryption*. After encrypting our message, the scrambled version is called *ciphertext*. From the ciphertext, we can recover our original unscrambled message via *decryption*. Figure 2 illustrates the processes of encryption and decryption. A *cryptosystem* is comprised of a pair of related encryption and decryption processes.

$$\begin{tikzcd}\boxed{\text{plaintext}} \arrow[r, "\text{encrypt}"] & \boxed{\text{ciphertext}} \arrow[r, "\text{decrypt}"] & \boxed{\text{plaintext}}\end{tikzcd}$$

Figure 2: A schema for encryption and decryption.

Table 1 provides a very simple method of scrambling a message written in English and using only upper case letters, excluding punctuation characters. Formally, let $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \dots, \mathtt{Z}\}$ be the set of capital letters of the English alphabet. Furthermore, let $\Phi = \{65, 66, 67, \dots, 90\}$ be the American Standard Code for Information Interchange (ASCII) encodings of the upper case English letters. Then Table 1 explicitly describes the mapping $f : \Sigma \longrightarrow \Phi$. (Those who are familiar with ASCII would recognize that f is a common process for *encoding* elements of Σ, rather than a cryptographic "scrambling" process *per se*.) To scramble a message written using the alphabet Σ, we simply replace each capital letter of the message with its corresponding ASCII encoding. However, the scrambling process described in Table 1 provides very little to no cryptographic security at all, and we strongly discourage its use in practice as a method of encryption.

A	B	C	D	E	F	G	H	I	J	K	L	M
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
65	66	67	68	69	70	71	72	73	74	75	76	77
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
78	79	80	81	82	83	84	85	86	87	88	89	90

Table 1: ASCII encodings of English capital letters.

4 Keeping a secret with two keys

The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a *public key cryptosystem*. RSA uses a *public key* to encrypt messages and decryption is performed using a corresponding *private key*. We can distribute our public keys, but for security reasons we should keep our private keys to ourselves. The encryption and decryption processes draw upon techniques from elementary number theory. Algorithm 1 [6, p.165] outlines the RSA procedure for encryption and decryption.

The next two sections will step through the RSA algorithm, using PARI/GP to generate public and private keys, and perform encryption and decryption based on those keys.

```
1  Choose two primes $p$ and $q$ and let $n = pq$.
2  Let $e \in \mathbb{Z}$ be positive such that $\gcd\big(e, \varphi(n)\big) = 1$.
3  Compute a value for $d \in \mathbb{Z}$ such that $de \equiv 1 \pmod{\varphi(n)}$.
4  Our public key is the pair $(n, e)$ and our private key is the triple $(p, q, d)$.
5  For any non-zero integer $m < n$, encrypt $m$ using $c \equiv m^e \pmod{n}$.
6  Decrypt $c$ using $m \equiv c^d \pmod{n}$.
```

Algorithm 1: The RSA algorithm for encryption and decryption.

5 Generating public and private keys

Positive integers of the form $M_m = 2^m - 1$ are called *Mersenne numbers*. If p is prime and $M_p = 2^p - 1$ is also prime, then M_p is called a *Mersenne prime*. For example, 31 is prime and $M_{31} = 2^{31} - 1$ is a Mersenne prime, as can be verified using the command `isprime(p)`. This command returns `1` if its argument `p` is prime; otherwise it returns `0` when `p` is composite. Indeed, the number $M_{61} = 2^{61} - 1$ is also a Mersenne prime. We can use M_{31} and M_{61} to work through step 1 in Algorithm 1:

PARI/GP
```
gp > p = (2^31) - 1
%6 = 2147483647
gp > isprime(p)
%7 = 1
gp > q = (2^61) - 1
%8 = 2305843009213693951
gp > isprime(q)
%9 = 1
gp > n = p * q
%10 = 4951760154835678088235319297
```

A word of warning is in order here. In the above code example, the choice of p and q as Mersenne primes, and with so many digits far apart from each other, is a very bad choice in terms of cryptographic security. However, we shall continue to use the above chosen numeric values for p and q for the remainder of this article, always bearing in mind that they have been chosen for educational purposes only. The reader is encouraged to refer to [2, 5, 6] for in-depth discussions on the security of RSA, or consult other specialized texts.

For step 2, we need to find a positive integer that is coprime to $\varphi(n)$. The command `random(n)` returns a pseudo-random integer within the closed interval $[0, n-1]$. Using a programming loop, we can compute the required value of e as follows:

PARI/GP
```
gp > e = random(eulerphi(n));
gp > while (gcd(e, eulerphi(n)) != 1, e = random(eulerphi(n)))
gp > e
%12 = 4048613754158036308925952619
```

To calculate a value for d in step 3 of Algorithm 1, we use the extended Euclidean algorithm. By definition of congruence, $de \equiv 1 \pmod{\varphi(n)}$ is equivalent to

$$de - k \cdot \varphi(n) = 1$$

where $k \in \mathbb{Z}$. From steps 1 and 2, we know the numeric values of e and $\varphi(n)$. The extended Euclidean algorithm allows us to compute d and $-k$. In PARI/GP, this can be accomplished via the command `bezout`. Given two integers x and y, `bezout(x, y)` returns a vector `[u, v, g]` such that $g = \gcd(x, y)$ and $ux + vy = g$. Having computed a value for `d`, we then use the command `Mod(d*e, eulerphi(n))` to check that `d*e` is indeed congruent to 1 modulo `eulerphi(n)`:

PARI/GP

```
gp > bezout(e, eulerphi(n))
%13 = [-2467255312924178983708448021, 2017255175378597045689617010, 1]
gp > Mod(-2467255312924178983708448021, eulerphi(n))
%14 = Mod(2484504839605656093165693679, 4951760152529835076874141700)
gp > d = 2484504839605656093165693679
%15 = 2484504839605656093165693679
gp > Mod(d * e, eulerphi(n))
%16 = Mod(1, 4951760152529835076874141700)
```

Thus, our RSA public key is

$$(n, e) = (4951760154835678088235319297, 4048613754158036308925952619) \qquad (2)$$

and our corresponding private key is

$$(p, q, d) = (2147483647, 2305843009213693951, 2484504839605656093165693679)$$

6 Encryption and decryption

Suppose we want to scramble the message "`HELLOWORLD`" using RSA encryption. From Table 1, our message maps to integers of the ASCII encodings as follows:

H	E	L	L	O	W	O	R	L	D
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
72	69	76	76	79	87	79	82	76	68

Table 2: ASCII encodings of `HELLOWORLD`.

Concatenating all the integers in Table 2, our message can be represented by the integer

$$m = 72697676798779827668 \qquad (3)$$

There are other more cryptographically secure means for representing our message as an integer, as opposed to the method presented in this article. The above process is used for demonstration purposes only and we strongly discourage its use in practice.

To encrypt our message, we raise m to the power of e and reduce the result modulo n. The command `Mod(a^b, n)` first computes `a^b` and then reduces the result modulo `n`. If the exponent `b` is a "large" integer, say with more than 20 digits, then performing modular exponentiation in this naive manner can take quite some time. Naive modular exponentiation is inefficient and, when performed using a computer, can quickly consume a huge quantity of the computer's memory or result in error messages. For instance, if we perform naive modular exponentiation using the command `Mod(m^e, n)`, where `m` is per (3), and `n` and `e` are as in (2), we would get an error message similar to the following:

```
PARI/GP
gp > Mod(m^e, n)
  ***   length (lg) overflow
```

```
/* Modular exponentiation using repeated squaring. */
/* That is, we want to compute a^b mod n. */
modexp(a, b, n) = {
    local(d, bin);
    d = 1;
    bin = binary(b);
    for (i = 1, length(bin),
        d = Mod(d*d, n);
        if (bin[i] == 1,
            d = Mod(d*a, n);
        );
    );
    return(d);
}
```

Listing 2: PARI/GP function for modular exponentiation via repeated squaring.

There is a trick to efficiently perform modular exponentiation, called the method of repeated squaring [1, p.879]. Suppose we want to compute $a^b \mod n$. First, let $d := 1$ and obtain the binary representation of b, say $(b_1, b_2, \dots, b_k)$ where each $b_i \in \mathbb{Z}_2$. For $i := 1, \dots, k$, let $d := d^2 \mod n$ and if $b_i = 1$ then let $d := da \mod n$. This algorithm is implemented in Listing 2, which defines a PARI/GP function called `modexp` to perform modular exponentiation using repeated squaring.

The code in Listing 2 can be saved to a file called, say, `/home/mvngu/modexp.gp`. The file can then be read into PARI/GP using the command `\r`. Let us read the above file into PARI/GP and encrypt our message:

```
PARI/GP
gp > m = 72697676798779827668
%17 = 72697676798779827668
gp > \r /home/mvngu/modexp.gp
gp > modexp(m, e, n)
%18 = Mod(359367672514946173472712696, 4951760154835678088235319297)
gp > c = 359367672514946173472712696
%19 = 359367672514946173472712696
```

Thus $c = 359367672514946173472712696$ is the ciphertext. To recover our plaintext, we raise `c` to the power of `d` and reduce the result modulo `n`. Again, we use modular exponentiation via repeated squaring in the decryption process:

```
PARI/GP
gp > modexp(c, d, n)
%22 = Mod(72697676798779827668, 4951760154835678088235319297)
```

Notice in the last output that the value 72697676798779827668 is the same as the integer that represents our original message. Hence we have recovered our plaintext.

7 What's next?

This article has demonstrated that the computer algebra system PARI/GP can be used to study undergraduate number theory and cryptography. As the reader delves further into number theory or its application to cryptography, software tools such as PARI/GP would be handy in the learning process. Further information about PARI/GP can be obtained from its website at `http://pari.math.u-bordeaux.fr`.

The structure of this article closely follows the tutorial *Number Theory and the RSA Public Key Cryptosystem*, written by the same author for the Sage [4] documentation project. That tutorial can be found online at

`http://wiki.sagemath.org/DocumentationProject`

Many of the invaluable feedbacks by Martin Albrecht and William Stein to that tutorial have been incorporated into this article.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, USA, 2nd edition, 2001.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, FL, USA, 1996.

[3] PARI Group. PARI/GP (version 2.3.4), July 2008.
`http://pari.math.u-bordeaux.fr`.

[4] W. Stein. *Sage: Open Source Mathematics Software (version 3.1.4).* The Sage Group, 03 November 2008.
`http://www.sagemath.org`.

[5] D. R. Stinson. *Cryptography: Theory and Practice.* Chapman & Hall/CRC, Boca Raton, USA, 3rd edition, 2006.

[6] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory.* Pearson Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 2006.