Montgomery reduction with even modulus

Ç.K. Koç

1 Montgomery reduction algorithm

In Reference 1, P.L. Montgomery introduced an efficient algorithm for computing

\[ c = a \cdot b \pmod{n} \]  

(1)

where \( a \), \( b \), and \( n \) are \( k \)-bit binary numbers. The algorithm is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) that are capable of performing fast arithmetic modulo a power of two. The Montgomery reduction algorithm computes the resulting \( k \)-bit number \( c \) in eqn. 1 without performing a division by the modulus \( n \). Through an ingenious representation of the residue class modulo \( n \), this algorithm replaces division by \( n \) operation with division by a power of two. This operation is easily accomplished on a computer since the numbers are represented in binary form.

Assuming the modulus \( n \) is a \( k \)-bit number, i.e. \( 2^k - 1 \leq n < 2^k \), let \( r = 2^k \). The Montgomery reduction algorithm requires that \( r \) and \( n \) be relatively prime; i.e. \( \gcd(r, n) = \gcd(2^k, n) = 1 \). This requirement is satisfied if \( n \) is odd. The following summarises the basic idea behind the Montgomery reduction algorithm. Given as an integer \( a < n \), its \( n \)-residue with respect to \( r \) is defined

\[ \bar{a} = a \cdot r \pmod{n} \]  

(2)

It is straightforward to show that the set

\[ \{i \cdot r \mod{n} \mid 0 \leq i \leq n - 1 \} \]

is a complete residue system, i.e. it contains all numbers between 0 and \( n - 1 \). Thus, there is one-to-one correspondence between the numbers in the range 0 and \( n - 1 \) and the numbers in the set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the \( n \)-residue of the product of the two integers whose \( n \)-residues are given. Given two \( n \)-residues \( \bar{a} \) and \( \bar{b} \), the Montgomery product is defined as the \( n \)-residue

\[ \bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \]  

(3)

where \( r^{-1} \) is the inverse of \( r \) modulo \( n \), i.e. the number with the property

\[ r^{-1} \cdot r = 1 \pmod{n} \]

The resulting number \( \bar{c} \) in eqn. 3 is indeed the \( n \)-residue of the product

\[ c = a \cdot b \pmod{n} \]

since

\[ \bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \]

\[ = a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \]

\[ = a \cdot b \cdot r \pmod{n} \]

To describe the Montgomery reduction algorithm, an additional quantity is needed \( n' \), which is the integer with the property

\[ r \cdot r^{-1} - n \cdot n' = 1 \]

The integers \( r^{-1} \) and \( n' \) can both be computed by the extended Euclid algorithm [2]. The Montgomery product computation is

function MonProd(\bar{a}, \bar{b})
Step 1 \( t := \bar{a} \cdot \bar{b} \)
Step 2 \( m := t \cdot n' \pmod{r} \)
Step 3 \( u := (t + m \cdot n)/r \)
Step 4 if \( u > n \) then return \( u - n \) else return \( u \)

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo \( r \) and divisions by \( r \), both of which are intrinsically fast operations, since \( r \) is a power two.

Since the preprocessing operations (conversion from ordinary residue to \( n \)-residue, computation of \( n' \), and converting the result back to ordinary residue) are rather time consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation, i.e. the computation of \( a^e \pmod{n} \). Using the binary method for computing the powers [2], replace the exponentiation

---

This research was supported in part by RSA Data Security, Inc., Redwood City, California.
operation by a series of square and multiplication operations modulo \( n \). This is where the Montgomery product operation finds its best use.

Let the binary expansion of the exponent \( e \) be \( (e_{k-1}, e_{k-2}, \ldots, e_0) \). The following summarises the modular exponentiation operation which makes use of the Montgomery product function \( \text{MonPro} \).

function \( \text{ModExp}(a, e, n) \) \{ \( n \) is odd \}
\begin{align*}
\text{Step 1} & \quad \text{Compute } n' \text{ using extended Euclid algorithm} \\
\text{Step 2} & \quad \hat{a} := a \cdot r \pmod{n} \\
\text{Step 3} & \quad \hat{x} := 1 \cdot r \pmod{n} \\
\text{Step 4} & \quad \text{for } i = k - 1 \text{ down to } 0 \text{ do} \\
\text{Step 5} & \quad x := \text{MonPro}(\hat{x}, \hat{x}) \\
\text{Step 6} & \quad \text{if } e_i = 1 \text{ then } x := \text{MonPro}(\hat{a}, x) \\
\text{Step 7} & \quad x := \text{MonPro}(x, 1) \\
\text{Step 8} & \quad \text{return } x
\end{align*}

Thus, start with the ordinary residue \( a \) and obtain its \( n \)-residue \( \hat{a} \) using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, steps 2 and 3 require divisions. However, once the preprocessing has been completed, the inner loop of the binary exponentiation method uses the Montgomery product operation which performs only multiplications modulo \( 2^w \) and divisions by \( 2^w \). When the binary method finishes, one obtains the \( n \)-residue \( x \) of the argument \( x = a^e \pmod{n} \). The ordinary residue number is obtained from the \( n \)-residue by executing the \( \text{MonPro} \) function with arguments \( \hat{x} \) and 1. This is easily shown to be correct, since

\[
\hat{x} = x \cdot r \pmod{n}
\]

immediately implies that

\[
x = \hat{x} \cdot r^{-1} \pmod{n} = \hat{x} \cdot 1 \cdot r^{-1} \pmod{n} = \text{MonPro}(\hat{x}, 1)
\]

The resulting algorithm is quite fast and efficient, as demonstrated by many researchers and engineers who have implemented it [3–6]. However, the algorithm can be refined and made more efficient, particularly when the computing machine involves multiprecision integers. The paper by Dusse and Kaliski [3] describes improved algorithms, including a simple and efficient method for computing \( n' \).

The modular exponentiation algorithm is used in cryptography; for example, the RSA algorithm [7], the ElGamal signature scheme [8], and the proposed digital signature standard (DSS) of the National Institute for Standards and Technology [9] require the computation of \( a^e \pmod{n} \) for large values of \( n \) (usually \( \log_2 n \geq 512 \)).

2 Case of even modulus

Since the existence of \( r^{-1} \) and \( n' \) requires that \( n \) and \( r \) be relatively prime, one cannot use the Montgomery product algorithm when this rule is not satisfied. Take \( r = 2^w \) since arithmetic operations are based on binary arithmetic modulo \( 2^w \), where \( w \) is the wordsize of the computer. In the case of single-precision integers, you could choose \( k \) to be an integer multiple \( w \). Since \( r = 2^w \), the modular exponentiation algorithm requires that

\[
gcd(r, n) = \gcd(2^w, n) = 1
\]

which is satisfied if and only if \( n \) is odd. This is not a restriction for the RSA algorithm since the modulus, being a product of two primes, is always an odd number.

In the following, a simple technique is described which can be used whenever one needs to compute a modular exponentiation operation with respect to an even modulus. Note that the proposed technique is similar to Quisquater and Couvreur algorithm given in Reference 10, which partitions an RSA decryption operation into two modular exponentiation operations with respect to the prime factors of the user's modulus. Let \( n \) be factored such that

\[
n = q \cdot 2^l
\]

where \( q \) is an odd integer. This easily be accomplished by shifting the even number \( n \) to the right until its least-significant bit becomes one. Then, by application of the Chinese remainder theorem, the computation of

\[
x = a^e \pmod{n}
\]

is broken into two independent parts such that

\[
x_1 = a^e \pmod{q}
\]

\[
x_2 = a^e \pmod{2^l}
\]

The final result \( x \) has the property

\[
x \pmod{q} = x_1 \pmod{q}
\]

\[
x \pmod{2^l} = x_2 \pmod{2^l}
\]

and can be found using one of the Chinese remainder algorithms: The single-radix conversion algorithm or the mixed-radix conversion algorithm [2, 11]. The computation of \( x_1 \) in eqn. 4 can be performed using the \text{ModExp} algorithm since \( q \) is odd. Meanwhile, the computation of \( x_2 \) in eqn. 5 can be performed even more easily, since it involves arithmetic modulo \( 2^l \). There is, however, some overhead involved due to the introduction of the Chinese remainder theorem. According to the mixed-radix conversion algorithm, the number whose residues are \( x_1 \) and \( x_2 \) modulo \( q \) and \( 2^l \), respectively, is equal to

\[
x = x_1 + q \cdot y
\]

where

\[
y = (x_2 - x_1) \cdot q^{-1} \pmod{2^l}
\]

The inverse \( q^{-1} \pmod{2^l} \) exists since \( q \) is odd. It can be computed using the simple algorithm given in Reference 3. One thus has the following algorithm

function \( \text{NewModExp}(a, e, n) \) \{ \( n \) is even \}
\begin{align*}
\text{Step 1} & \quad \text{Shift } n \text{ to the right obtain the factorisation } n = q \cdot 2^l \\
\text{Step 2} & \quad \text{Compute } x_1 := a^e \pmod{q} \text{ using ModExp routine} \\
\text{Step 3} & \quad \text{Compute } x_2 := a^e \pmod{2^l} \text{ using binary method and modulo } 2^w \text{ arithmetic} \\
\text{Step 4} & \quad \text{Compute } q^{-1} \pmod{2^l} \text{ and } y := (x_2 - x_1) \cdot q^{-1} \pmod{2^l} \\
\text{Step 5} & \quad x := x_1 + q \cdot y \text{ and return } x
\end{align*}

In Step 2, one can use Euler's theorem, and prior to calling the \text{ModExp} routine, one can reduce \( e \) modulo \( \phi(q) \), where \( \phi(q) \) represents Euler's totient function of \( q \). This reduction is possible only if the factorisation of \( q \) is known. In Step 3, one can reduce \( e \) modulo \( \phi(2^l) \), since \( \phi(2^l) = 2^l - 1 \). These reductions allow a further decrease in the computation time of the \text{NewModExp} routine.

The new modular exponentiation algorithm breaks the exponentiation operation into two exponentiation problems of smaller size. Let \( n \) be an \( i \)-bit number. Since \( n = q \cdot 2^l \), the odd factor \( q \) is an \((i-l)\)-bit number. It is
known that computation of $a^r \mod n$ requires approximately $1.5k$ multiplications, assuming the number of bits in $e$ is equal to $k$. Therefore ignoring the preprocessing costs, the ModExp routine requires approximately $1.5k^2$ bit operations. The NewModExp routine, on the other hand, requires $1.5k(i-j)^2$ bit operations for computing $a^r \mod q_i$, and $1.5k^2$ bit operations for computing $a^r \mod 2^j$, and thus a total of $1.5k(i^2 - 2j + j^2)$ bit operations.

The speedup is calculated as

$$i^2 - 2j + j^2 = \frac{1}{1 - 2(j/i) + 2(j/i)^2}$$

for $0 \leq j < i$. As $j$ ranges from 0 to $i$, the speedup takes values from 1 to 2. For example, when $j = i/10$, the speedup is equal to 1.22. The optimal value of $j$ which maximises the speedup is found as $j = i/2$; in this case the speedup becomes two.

3 Example

The computation of $x = a^e \mod n$ for $a = 375$, $e = 249$, and $n = 388$ is illustrated.

Step 1: $n = 388 = (110000100) = (11000001) \times 2^2 = 97 \times 2^2$. Thus, $q = 97$ and $j = 2$.

Step 2: Compute $x_1 = a^e \mod q$ by calling ModExp with parameters $a = 375$, $e = 249$, and $q = 97$. Before calling the ModExp routine, reduce $a$ modulo $q$, and $e$ modulo $\phi(q)$. The reduction of $e$ modulo $\phi(q)$ is possible only if the factorisation of $q$ is known. Assuming the factorisation of $q$ is not known, one only reduces $a$ to obtain

$$a \mod q = 375 \mod 97 = 84$$

and call the ModExp routine with parameters (84, 249, 97). Since $q$ is odd, the ModExp routine successfully computes the result as $x_1 = 78$.

Step 3: Compute $x_2 = a^e \mod 2^j$ by calling an exponentiation routine based on the binary method and modulo $2^j$ arithmetic. Before calling such a routine reduce the parameters to

$$a \mod 2^j = 375 \mod 4 = 3$$

$$e \mod \phi(2^j) = 249 \mod 2 = 1$$

In this case, one is able to reduce the exponent since $\phi(2^j) = 2^{j-1}$. Thus, we call the exponentiation routine with the parameters (3, 1, 4). The routine computes the result as $x_2 = 3$.

Step 4: Using the extended Euclid algorithm, compute

$$q^{-1} \mod 2^j = 97^{-1} \mod 4 = 1$$

Then compute

$$y = (x_2 - x_1) \cdot q^{-1} \mod 2^j = (3 - 78) \cdot 1 \mod 4 = 1$$

Step 5: Finally, compute and return the result

$$x = x_1 + q \cdot y = 78 + 97 \cdot 1 = 175$$

4 References