/01/

ALGORITMO DE BOOTH EN ARITMÉTICA MODULAR PARA OPERACIONES DE MULTIPLICACIÓN ESCALAR

BOOTH ALGORITHM MODULAR ARITHMETIC FOR SCALAR MULTIPLICATION OPERATIONS

Jesús Ayuso Pérez
Compositor musical y desarrollador.
Licenciado en Ingeniería Informática por la Universidad Carlos III de Madrid (UC3M).
ayusoperez@gmx.es

Recepción: 29/12/2017. **Aceptación:** 11/05/2018. **Publicación:** 29/06/2018

Citación sugerida:
Ayuso Pérez, J. (2018). Algoritmo de Booth en aritmética modular para operandos de multiplicación escalar. *3C TIC: Cuadernos de desarrollo aplicados a las TIC,* 7(2), 10-35. DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

RESUMEN

El algoritmo dado por Andrew Donald Booth en 1950 (Booth, 1951) para la multiplicación no es únicamente aplicable a dicha operación cuando actúan números enteros, se puede emplear a la misma también al multiplicar un entero por un punto de una curva elíptica (Ayuso 2015, pp. 255-221), dando otra dimensión al citado método, ya que dicho cálculo no queda descrito como el producto de varias componentes, sino que se define de manera completamente distinta: basada en su naturaleza geométrica. De ahí que en el presente documento, propongamos varios algoritmos de multiplicación escalar sobre nuevas relaciones aditivas (Cassels, 1966, pp. 193–29) basados en el concepto ideado por Booth. Viendo distintas aportaciones a la hora de realizar ese cómputo y con el añadido de apoyarnos en operaciones que igualmente ya explotan el concepto introducido por Booth.

ABSTRACT

The algorithm given by Andrew Donald Booth in 1950 (Booth, 1951) for multiplication is not only applicable to this operation when integers are used, but can also be used at the same scale by multiplying a whole number by a point on a curve elliptical (Ayuso 2015, pp. 255-221), giving another dimension to the method mentioned above, since this calculation does not remain as the product of several components, but is defined in a completely different way: based on its geometric nature. Therefore, in the current document, we propose several algorithms of scalar multiplication on new additive relationships (Cassels, 1966, pp. 193–29) based on the concept devised by Booth. See the contributions at the time of making this calculation and with the addition of supporting us in operations that also exploit the concept presented by Booth.

PALABRAS CLAVE

Booth, Algoritmo, Multiplicación Escalar, Modular, Curva Elíptica.

KEY WORDS

Booth, Algorithm, Scalar Multiplication, Modular, Elliptic Curve.

Ed. 25. Vol.7 Nº 2. Junio-Septiembre 2018
DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

1. INTRODUCCIÓN

Partiendo de que una curva elíptica viene definida por un polinomio homogeneo de grado 3 (Cassels, 1966, pp. 193–29), y que sin embargo es posible hacer cambios de variable, sin alterar la estructura de grupo, y que nos permiten hacer 0 muchos de los coeficientes. Se llega así a la ecuación simplificada de Weierstrass para cuerpos de característica distinta de 2 y 3; tirando de bibliografía, queda resumido el pseudocódigo para el cálculo de la suma de 2 puntos, P y Q, módulo m, de una curva elíptica, descrita por $y\text{^}2 = x\text{^}3 + a\,x + b$, de manera que en dichos cuerpos puede expresarse como:

```
if P = Q
     lambda = ( (3 * P[x] * P[x]) + a ) * ( 2 * P[y] )^-1
else
     lambda = ( Q[y] - P[y] ) * ( Q[x] - P[x] )^-1


 result[x] = ( lambda * lambda ) - P[x] - Q[x]


 result[y] = ( lambda * (P[x] - result[x]) ) - P[y]
```

Figura 1. Algoritmo de suma de puntos de una Curva Elíptica.

La operación de multiplicación escalar es planteada como una sucesión de sumas y/o restas entre puntos de la curva elíptica, entendidas éstas últimas, las restas, al estar en un contexto modular, como sumas por el inverso aditivo.

Con la definición anterior, en el presente desarrollo se verá cómo el concepto de Booth aplicado dentro de un contexto modular a la operación de grupo que se define entre los puntos que conforman una curva elíptica soporta muchos cálculos cancelativos, los cuales nos permitirán ahorrar operaciones, con

el consiguiente aumento en rendimiento. Además de la ya consabida capacidad de reducción de las secuencias de 1s, para el caso de construir una operación contra un escalar, partiendo de esa primera operación algebraica. Este tipo de cálculos son muy utilizados en distintos campos y especialmente en el mundo de la Criptografía (Diffie-Hellman, pp. 644–654).

Dicho esto, se entenderá que la operación de multiplicación escalar es planteada como una sucesión de sumas y/o restas entre puntos de la curva elíptica, entendidas éstas últimas, las restas, al estar en un contexto modular, como sumas por el inverso aditivo. Y además, en todas ellas la modularidad nos será transparente, por apoyarnos en las implementaciones de primitivas modulares referenciadas (Ayuso, pp. 222-229, 255-221, 28-41, 1-12): multiplicación, inverso, adición, sustracción… Todas ellas a su vez inspiradas en el algoritmo de Booth.

2. MÉTODOS

En este apartado, lo primero que haremos será dar la implementación de la operación de grupo dentro de una curva elíptica: suma y resta de puntos (Cassels, 1966, pp. 193–29). Más concretamente, como decíamos, se dará una versión usando las implementaciones que se describen en la referencia bibliográfica titulada: '*Booth algorithm in modular exponentiation operations*' (Ayuso, 2017, pp. 1-12), tanto de las suma y resta modular entre enteros (Ayuso, 2015, pp. 222-229), como las de multiplicación (Ayuso, 2015, pp. 255-221) e inverso modular (Ayuso, 2016, pp. 28-41). De hecho, se hará referencia a ellas exactamente con el mismo nombre con el que son usadas en dichas publicaciones.

Aun así, previamente se tendrá que exponer algo de código para alcanzar el algoritmo de la operación de grupo, Figura 1, de una forma limpia y entendible, de cara a construir la operación de multiplicación escalar, que es la que realmente interesa y se está tratando en el presente documento.

La primera operación descrita para facilitar el algoritmo final se trata de una que calcularía el resultado de multiplicar un entero por él mismo, y luego triplica el resultado; es decir, t por t por 3, módulo m, sería:

DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

```
 result = t;

 opBooth = mMultiplication1(result, t, m);

 aux = result << 1; // MULTIPLICA t^2 por 2

 if(aux >= m)

aux = subtraction1(m, aux);

 switch(mAddition1(result, aux, m)) {

          case ( 1 0 ):

               opBooth = opBooth == ( 1 0 ) ? ( 0 1 )

                                              : ( 1 0 );
               break;

          default:

               break;

 }

 return opBooth;
```

Figura 2. Algoritmo de multiplicación y triplicado (multiplyTriple).

La siguiente operación que describimos será una que calcularía el resultado de duplicar un entero, y luego calcula el inverso modular al resultado; es decir, inverso de t más t, módulo m, sería:

```
result = t << 1; // MULTIPLICA t por 2


 if(result >= m)

result = subtraction1(m, result);


 opBooth = extendedEuclideanBinary(result, result, m)


 return opBooth;
```

Figura 3. Algoritmo de duplicado e inversión (multiplyInverse).

La última operación que predefinimos será una que obtiene el resultado de una resta entre enteros, y luego calcula el inverso modular al resultado; es decir, inverso de *t* menos *s*, módulo *m*, sería:

```
result = t;

 opBooth = mSubtraction1(result, s);


 switch(extendedEuclideanBinary(result, result, m)) {

          case ( 1 0 ):

                opBooth = opBooth == ( 1 0 ) ? ( 0 1 )

                                              : ( 1 0 );


                break;

          default:

                break;

 }
 return opBooth;
```

Figura 4. Algoritmo de resta e inversión (subtractInverse).

Llegados a este punto, para no perdernos, recapitulamos un poco. Este apartado abre de manera que se está describiendo el código para calcular la operación aditiva expresada en **Figura 1**. Para

definir la misma de una manera más inteligible, y también más compacta, nos apoyamos en 3 operaciones, las que figuran en **Figura 2**, **3** y **4**. Como se ha ido detallando, dichas suboperaciones simplemente absorben unos cálculos parciales, valiéndose de los algoritmos referenciados en la bibliografía.

Apostillado lo anterior, con las tres operaciones previas, ya nos es más sencillo definir el cálculo de la variable nombrada lambda en la **Figura 1 (Algoritmo de suma de puntos de una curva elíptica)** utilizada en la operación de grupo para el cálculo de la pendiente (Cassels, 1966, pp. 193–29). Puede observarse en la citada imagen, que tenemos 2 casos, que se traducen en 2 códigos distintos para calcular dicha variable. El primero, la suma de 2 puntos, *P* y *Q*, módulo m, de una curva elíptica, descrita por *y^2 = x ^3 + a x + b*, cuando *P* es igual a *Q*, quedaría:

```
aux = P[x];

opBooth1 = multiplyTriple(aux, m);

switch(opBooth1) {

          case ( 1 0 ):

                opBooth1 = mSubtraction1(aux, a) == ( 1 0 ) ?

                                    ( 0 1 ) : ( 1 0 );

                break;

          default:

                opBooth1 = mAddition1(aux, a, m));
                break;
}
```

DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

```
lambda = P[y];


 opBooth2 = multiplyInverse(lambda, m);

 switch(mMultiplication1(lambda, aux, m)) {

    case ( 1 0 ):


          if(opBooth2 == ( 1 0 ))

               opBooth1 = opBooth1 == ( 1 0 ) ?

                    ( 1 0 ) : ( 0 1 );

          else

               opBooth1 = opBooth1 == ( 1 0 ) ?

                    ( 0 1 ) : ( 1 0 );

          break;
```

```
aux = P[x];

    default:

          if(opBooth2 == ( 1 0 ))

                opBooth1 = opBooth1 == ( 1 0 ) ?

                      ( 0 1 ) : ( 1 0 );

          break;
}

return opBooth1;
```

Figura 5. Algoritmo de cálculo de la pendinte cuando P es igual a Q (calcEquLambda).

El segundo caso para la pendiente denotada como lambda es con la suma de 2 puntos, P y Q, módulo m, de una curva elíptica, descrita por y^2 = x ^3 + a x + b, cuando P es distinto a Q, quedando:

```
lambda = Q[y];

 aux = Q[x];

opBooth1 = mSubtraction1(lambda, P[y]);
```

Figura 6. Algoritmo de cálculo de la pendiente cuando P es distinto a Q (calcNeqLambda).

Bien, ya se puede escribir el código de la operación de grupo. En la introducción se ha dado un pseudocódigo en la **Figura 1 (Algoritmo de suma de puntos de una curva elíptica)** entendiendo que la coordenada y del resultado se obtenía como:

```
result[y] = ( lambda * (P[x] - result[x]) ) - P[y]
```

Figura 7. Algoritmo para coordenada Y en suma de puntos de una curva elíptica.

Esto es así, porque al calcular el tercer punto, *R*, donde la recta dada por los puntos *P* y *Q* intersecta la curva elíptica, lo que en verdad se obtiene de forma directa es -*R*, por la naturaleza geométrica de la operación, en lugar de *R*; y por ello se requiere proyectarse contra el elemento negativo (Cassels, 1966, pp. 193–29). En nuestro caso, veremos que el aplicar los conceptos de Booth a los presentes cálculos, nos permite siempre trabajar indiferentemente con el valor de un elemento o de su inverso algebraico, haciendo insignificante lo anterior. De ahí que nuestra implementación para operación se suma de puntos, para la coordenada y queda realmente implementada como:

```
result[y] = ( lambda * (result[x] - P[x]) ) + P[y]
```

Figura 8. Algoritmo para coordenada inversa Y en suma de puntos de una curva elíptica.

Remarcado lo anterior, se presenta la suma de 2 puntos, *P* y *Q* de una curva elíptica, descrita por *y^2 = x ^3 + a x + b*, módulo m, como:

```
 if(P == Q)
opBooth1 = calcEquLambda(P, m, a, lambda);
 else
opBooth1 = calcNeqLambda(P, Q, m, a, lambda);


 result[x] = lambda;
switch(mMultiplication1(result[x], lambda, m)) {
          case ( 1 0 ):
               opBooth2 = mAddition1(result[x], P[x], m))
                     == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
               break;
          default:
               opBooth2 = mSubtraction1(result[x], P[x]);
               break;
 }
 switch(opBooth2) {
```

```
          case ( 1 0 ):
                opBooth2 = mAddition1(result[x], Q[x], m))
                      == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
                break;
          default:
opBooth2 = mSubtraction1(result[x], Q[x]);
                break;
 }
 result[y] = result[x];
 switch(opBooth2) {
          case ( 1 0 ):
                tmpOp = mAddition1(result[y], P[x], m))
                      == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
                break;
          default:
```

```
                tmpOp = mSubtraction1(result[y], P[x]);
                break;
 }
switch(mMultiplication1(result[y], lambda, m)) {
          case ( 1 0 ):
if(tmpOp == ( 1 0 ))
                opBooth1 = opBooth1 == ( 1 0 ) ?
                      ( 1 0 ) : ( 0 1 );
          else
                opBooth1 = opBooth1 == ( 1 0 ) ?
                      ( 0 1 ) : ( 1 0 );
          break;
     default:
          if(tmpOp == ( 1 0 ))
```

```
                opBooth1 = opBooth1 == ( 1 0 ) ?
                      ( 0 1 ) : ( 1 0 );
          break;
 }
 switch(opBooth1) {
      case ( 1 0 ):
                opBooth1 = mSubtraction1(result[y], P[y]);
                       == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
                break;
          default:
                opBooth1 = mAddition1(result[y], P[y], m))
                break;
 }
 if(opBooth2 = ( 1 0 ))
result[x] = subtraction1(m, result[x]);

 return opBooth1 == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
```

Figura 9. Algoritmo de suma de puntos de una curva elíptica (mAdditionEC).

En primer lugar, se comprueba que lo primero que realiza el algoritmo de la **Figura 9**, sobre estas líneas, es como habíamos adelantado hacer uso de las funciones definidas previamente:

- *calcEquLambda* (algoritmo de la **Figura 5**)

- *calcNeqLambda* (algoritmo de la **Figura 6**)

De manera que se invoca a una u otra dependiendo de si los elementos que vamos a operar son distintos o no, tal y como queda reflejado en la primera condición de la implementación que figura en la **Figura 1** de la introducción del artículo. El resto del pseudocodigo, como puede constatarse, simplemente realiza la implementación de las operaciones de las que consta la operación aditiva, tal y como está definida engendrando un grupo abeliano. Es decir, el algoritmo dado en **Figura 1** sólo que haciendo uso de las implementaciones documentadas en las referencias bibliográficas que hacen uso del algoritmo de Booth.

Por otra parte, y para terminar de construir todo lo necesario para lo que se busca exponer en el presente documento, se requerirá de la capacidad de restar un punto de la curva, pero como hemos adelantado en la introducción, esto se realizará como sumas por el inverso aditivo del valor del elemento que se desea sustraer, es decir, operando por el elemento algebraico inverso en el contexto en el que estamos trabajando: un contexto aditivo. Tenemos pues que la resta de 2 puntos, P y Q de una curva elíptica, descrita por $y^2 = x^3 + a\,x + b$, módulo m, sería:

```
result = P;
aux[x] = Q[x];
aux[y] = subtraction1(m, Q[y]);

return mAdditionEC(result, aux, m, a);
```

Figura 10. Algoritmo de resta de puntos de una curva elíptica (mSubtractionEC).

Una vez en este punto, destacar que, como hemos podido deducir, todas las versiones de los algoritmos utilizados, salvo *subtraction1*, retornan un valor que hace referencia a una acción de

DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

Booth, por ello entenderemos que dejan el resultado del cómputo en la primera variable que se le pase como parámetro en la llamada. Por simplicidad. Salvo en el caso de las nuevas operaciones *calcEquLambda* y *calcNeqLambda*, las cuales entendemos que dejan el resultado del cálculo en el último parámetro de la llamada, en *lambda*.

Ahora repasemos la tabla dada por Booth para reducir el número de operaciones necesarias, apoyándonos en la capacidad de hacer y deshacer las proyecciones que realizamos de unos elementos contra otros, que posee la operación algebraica con la que se construye nuestro cálculo:

Tabla 1. Tabla de acciones de Booth.

bit menos significativo	bit extra	Interpretación	Acción
0	0	intermedio cadena de 0s	ninguna
0	1	final cadena de 1s	operación
1	0	comienzo cadena de 1s	operación inversa / inverso misma operación
1	1	intermedio cadena de 1s	ninguna

Por último, simplemente para poder dar un código algo más compacto, definimos la siguiente pequeña función, la cual recibe 2 valores de la tabla de acciones de Booth, *opBooth1* y *opBooth2*, y genera 1 valor salida acorde al concepto. Entendemos una función que:

```
if(opBooth2 = ( 1 0 ))
    return opBooth1 == ( 1 0 ) ? ( 0 1 ) : ( 1 0 );
return opBooth1;
```

Figura 11. Algoritmo de cálculos parciales de acciones de Booth (calcOpBooth).

Por fin, partiendo de la tabla (**Tabla de acciones de Booth**) vamos a entrar en la operación que nos ocupa: la multiplicación escalar. Nuestro algoritmo de Booth aplicado a la multiplicación de un entero *t* por el punto *P* de una curva elíptica, descrita por *y^2 = x ^3 + a x + b*, módulo *m*, y todos de longitud *n*, sería:

```
opBooth1 = (0 0);     // NINGUNA ACCION AL INICIO
```

```
opBooth2 = (0 0);    // NINGUNA ACCION AL INICIO
bitExtra = 0;


weight[x] = P[x];
weight[y] = P[y];
opBooth2 = (0 1);


for(int i = 0; i < n; i++)  { // TRATA ELEMENTO NEUTRO EC
     if(t[i] == 1) {
          k = i;
          break;
     }
     tmpOp = mAdditionEC(weight, weight, m, a);
opBooth2 = calcOpBooth(opBooth2, tmpOp);
}
result[x] = weight[x];          // OP BOOTH INICIAL (1 0)
```

DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

```
result[y] = subtraction1(m, weight[y]);

opBooth1 = opBooth2;

bitExtra = 1;


tmpOp = mAdditionEC(weight, weight, m, a);

opBooth2 = calcOpBooth(opBooth2, tmpOp);


for(int i = k + 1; i < n; i++)  {

    switch(actionBooth(t[i], bitExtra) {

  case ( 0 1 ):

    if((opBooth1 == ( 1 0 ) && opBooth2 == ( 1 0 ))

          || (opBooth1 == ( 0 1 ) && opBooth2 == ( 0 1 )))

          tmpOp = mAdditionEC(result, weight, m, a);

    else

          tmpOp = mSubtractionEC(result, weight, m, a);

     opBooth1 = calcOpBooth(opBooth1, tmpOp);

     break;
```

```
   case ( 1 0 ):
     if((opBooth1 == ( 1 0 ) && opBooth2 == ( 1 0 ))
          || (opBooth1 == ( 0 1 ) && opBooth2 == ( 0 1 )))
          tmpOp = mSubtractionEC(result, weight, m, a);
     else
          tmpOp = mAdditionEC(result, weight, m, a);

     opBooth1 = calcOpBooth(opBooth1, tmpOp);
     break;
  default:
     break;
}
tmpOp = mAdditionEC(weight, weight, m, a);
opBooth2 = calcOpBooth(opBooth2, tmpOp);

bitExtra = t[i];
}
```

DOI: http://dx.doi.org/10.17993/3ctic.2018.60.10-35/

```
if(opBooth1 = ( 1 0 ))

   result[y] = subtraction1(m, result[y]);



return result;
```

Figura 12. Algoritmo de multiplicación escalar en Curvas Elípticas.

Hacer hincapié en que damos por sentado que las operaciones retornan una acción de Booth, tal y como se especifica en su implementación, y el resultado del cálculo es dejado en la primera de las variables que le llega por parámetro en la llamada.

También apostillar que el concepto de Booth exige la invertibilidad de la operación, luego, en el algoritmo mostrado, se da por supuesto que trabaja siempre sobre una estructura algebraica donde todo elemento es invertible, es decir, para el caso que nos ocupa, entendemos que estamos ante un grupo abeliano.

Y sobre todo profundizar en la forma en la que es aplicado el concepto de Booth en cada fragmento del algoritmo anterior, para una compresión completa. Si nos fijamos, el primer *switch*, utiliza el concepto de Booth en el sentido más clásico: reduce el número de operaciones a realizar apoyándose en la operación inversa; para el caso que nos ocupa, si entendemos la multiplicación como una sucesión de sumas, conseguimos reducir el número de sumas a realizar entre puntos, ayudándonos de operaciones de resta. Por otro lado, los *ifs* anidados nos sirven para apoyarnos en elementos inversos, y nuevamente reducir el número de operaciones explotando aún más el concepto de Booth. Por ello, cabe destacar esa última comprobación al final del código, en caso de que al salir del bucle, hayamos terminado obteniendo en elemento inverso, un punto negativo, en lugar del elemento resultado: obtenemos su valor real con una simple resta contra el módulo sobre el que estamos trabajando (por tratarse de un inverso aditivo); es decir, el valor positivo.

3. CONCLUSIONES

Aplicar el algoritmo de Booth en métodos que requieren de sucesivas operaciones relacionadas algebraicamente nos ofrece distintas herramientas cancelativas (Ayuso 2017, pp. 1-9) para ahorrar operaciones debido a que contrarrestan su acción ayudándonos en su inversa, en elementos inversos que evitan actuaciones o eliminando operaciones cruzadas (Ayuso 2017, pp. 19-26). Además, el hecho de que todas las operaciones primitivas que componen los cálculos, igualmente hagan uso del concepto de Booth, nos permite trabajar siempre con elementos pertenecientes a la estructura algebraica sobre la que operamos; en lugar de tener que sobrecargar los algoritmos con constantes procesos de reducción o encuadrado dentro del módulo. Con el coste añadido que eso supondría (Ayuso 2018, pp. 13-20).

El hecho de que todas las operaciones primitivas que componen los cálculos, igualmente hagan uso del concepto de Booth, nos permite trabajar siempre con elementos pertenecientes a la estructura algebraica sobre la que operamos.

En conclusión, los conceptos propuestos por Booth, son extensibles a distintos cuerpos y dimensiones algebraicas mientras se acabe trabajando sobre una aritmética construida con proyecciones invertibles, ofreciendo siempre la posibilidad de cancelar operaciones y reducir el número de cálculos necesarios (Ayuso 2017, pp. 1-12). Además de resultar una solución mucho más elegante, proporcionando algoritmos de una naturaleza más acorde conceptualmente y más rápidos. Ocurre también, gracias a las variables de apoyo descritas en la técnica dada originalmente por Booth, que la posibilidad de tener control sobre si se está en un lado u otro de esa dualidad algebraica que se produce por ese efecto de proyectabilidad de la relación de equivalencia, binaria, para el caso, nos ofrece la posibilidad de trabajar con elementos proyectados o no, indiferentemente, (Ayuso 2017, pp. 28-41), sin afectar al resultado final y evitándonos los posibles costes añadidos de deshacer la proyección.

Por otro lado, proporciona una técnica que es portable a operaciones de naturaleza bien diferente, o las cuales están definidas de manera completamente distinta, siempre y cuando acaben describiendo una relación bien formada entre los elementos de la estructura algebraica (Ayuso 2017, pp. 33-43). Llegando a convertirse en todo un paradigma para abordar cómputos que cumplas con los ya mencionados requisitos.

Como posible futura línea de investigación, se pueden tratar de aplicar los conceptos descritos en la aritmética de Montgomery (Montgomery 1985, pp. 519–521), ya que ésta permite realizar la operación de multiplicación de una manera más eficiente. Por lo que sería una técnica complementaria a la presente.

4. REFERENCIAS BIBLIOGRÁFICAS

Ayuso, J. (2015). Booth algorithm modular arithmetic operations of addition and subtraction. *3C TIC*, 4(3), 222-229.

Ayuso, J. (2015). Booth algorithm modular arithmetic operations of multiplication, *3C TIC*, 4(4) 255-221.

Ayuso, J. (2015). Booth algorithm operations addition and subtraction. *3C TIC*, 4(2) 113-119.

Ayuso, J. (2016). Booth algorithm in signed-digit representation. *3C TIC*, 5(3), 33-43.

Ayuso, J. (2016). Booth algorithm operations modular inverse. *3C TIC*, 5(2), 28-41.

Ayuso, J. (2017). Booth algorithm in arity with multiple operands. *3C TIC*, 6(4), 19-26.

Ayuso, J. (2017). Booth algorithm in modular exponentiation operations. *3C TIC*, 6(2), 1-12.

Ayuso, J. (2017). Booth algorithm hardware operations addition and subtraction. *3C TIC*, 6(3), 1-9.

Ayuso, J. (2018). Karatsuba algorithm operations exponentiation. *3C TIC*, 7(1), 13-20.

Booth, A. D. (1945). A method of calculating reciprocal spacings for X-ray reflections from a monoclinic crystal. *J. Sci. Instr,* 22, p. 74.

Booth, A. D. and Britten, K. H. V. (1947). *General Considerations in the Design of an Electronic Computer.*

Booth, A. D. (1951). A signed binary multiplication technique. *Q.J. Mech. and Appl. Math.*, 4(2), 236-240.

Cassels, J.W.S. (1966). Diophantine equations with special reference to elliptic curves. *J. London Math. Soc.*, 41, 193–29.

Diffie, W., & Hellman, E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 644–654.

Euclid of Alexandria. (1557). *Elements*. T.L. Heath's.

Montgomery, P. (1985.). Modular Multiplication Without Trial Division. *Math. Computation*, 44, 519–521.

W. Reitwiesner, G. (1960). Binary Arithmetic, 231-308.