Algorithms for Elliptic Nets

Graeme Taylor

January 2008

1 Tate Pairing via Elliptic Nets

For the Tate pairing of two distinct points P,Q with P of order m, we need to compute the terms $(m+1,0)$ and $(m+1,1)$ of the elliptic net corresponding to P,Q (See [Stange]). For this we can employ a double-and-add approach, chaining up from the first term (initial data) to the nth in $\log_2(n)$ steps. However, due to the structure of the recurrence relation, this requires keeping track of additional terms. To achieve this, Stange introduces blocks of terms.

2 Blocks

Define a *block centred on* k to be:

		$(k-1,1)$	$(k,1)$	$(k+1,1)$		
$(k-3,0)$	$(k-2,0)$	$(k-1,0)$	$(k,0)$	$(k+1,0)$	$(k+2,0)$	$(k+3,0)$	$(k+4,0)$

Computing the Tate pairing can thus be achieved by computing the block centred at any of $m-1$, m or $m+1$.

2.1 Double-and-Add Algorithm

Double-and-add Algorithm
INPUT: Integer n and block centred at 1.
OUTPUT: Block centred at n.

1. Compute binary digits d_i of n such that $n = (d_k d_{k-1} \dots d_1)_2$ with $d_k = 1$.
2. Set $c = 1$ (centre)
3. For i=k-1 down to 1 do:
   - If d_i=0 Compute block with centre $2c$; Set c to $2c$
   - Else, Compute block with centre $2c + 1$; Set c to $2c + 1$
4. Return final block.

Clearly, this requires procedures to generate the new blocks from the old- [Stange] gives formulae for these ((12)-(17)). A speed-up (*Id.* S5.1) can be achieved by computing commonly used squares/products at each step, denoted by $A(i) = (i, 0)^2$ and $B(i) = (i - 1, 0) \times (i + 1, 0)$ below.

2.2 Double

Given a block centred on k, we can compute the block centred on $2k$:

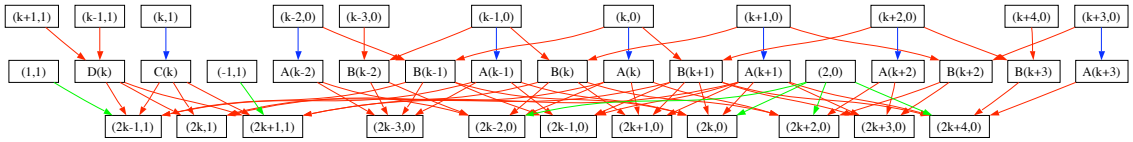

Figure 1: Dependencies for block doubling.

The inversions in the recurrence formulae are independent of k, so are precomputed and made available to **Double** and **DoubleAdd** as multipliers. Including multiplication by these values, the total cost of a block double is 35 multiplications, 7 squarings and no inversions.

2.3 DoubleAdd

Given a block centred on k, we can compute the block centred on $2k + 1$:

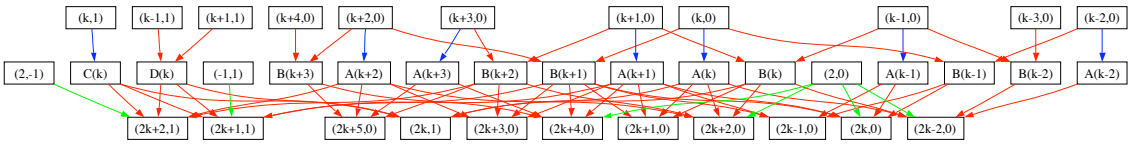

Figure 2: Dependencies for block double-and-add.

The inversions in the recurrence formulae are independent of k, so are precomputed and made available to **Double** and **DoubleAdd** as multipliers. Including multiplication by these values, the total cost of a block double-and-add is 35 multiplications, 7 squarings and no inversions.

3 Short Blocks

Define a *short block centred on* k to be:

		$(k-1,1)$	$(k,1)$	$(k+1,1)$	
$(k-3,0)$	$(k-2,0)$	$(k-1,0)$	$(k,0)$	$(k+1,0)$	$(k+2,0)$	$(k+3,0)$

Notice from 1 that the $(k+4,0)$ term is necessary only to generate $(2k+4,0)$ in the **Double** procedure. Thus given a short block centred at k, we can double it to obtain the short block centred at $2k$, as follows.

3.1 DoubleShort

Given a small block centred on k, we can compute the small block centred on $2k$:

The inversions in the recurrence formulae are independent of k, so are precomputed and made available to **DoubleShort** as multipliers. Including multiplication by these values, the total cost of a **DoubleShort** is 31 multiplications, 6 squarings and no inversions.

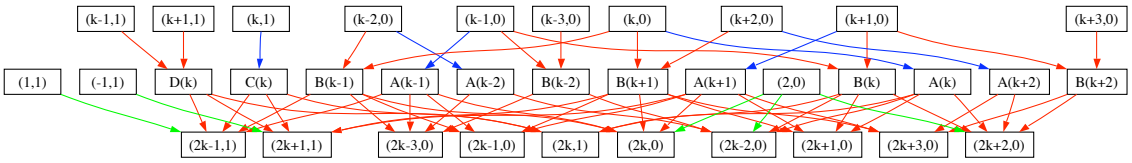

Figure 3: Dependencies for short block doubling.

3.2 DoubleAddShort

From 2 it can be seen that the missing $(k+4, 0)$ is required for both $(2k+4, 0)$ and $(2k + 5, 0)$- although the latter can be dropped for a small block, the former cannot. However, the terms $(k - 2, 0), \ldots, (k + 3, 0)$ already present in the short block are sufficient to restore it to a full block by computing $(k + 4, 0)$:

$$(k + 4, 0) = \frac{(k + 3, 0) \times (k + 1, 0) \times (2, 0)^2 - (3, 0) \times (k + 2, 0)^2}{(k, 0)}$$

Further, by storing $(2, 0)^2$, this patch requires only 2 multiplications and 1 inversion since some of the terms feature in the precomputation:

$$(k + 4, 0) = \frac{(2, 0)^2 B(k + 2) - (3, 0) A(k + 2)}{(k, 0)}$$

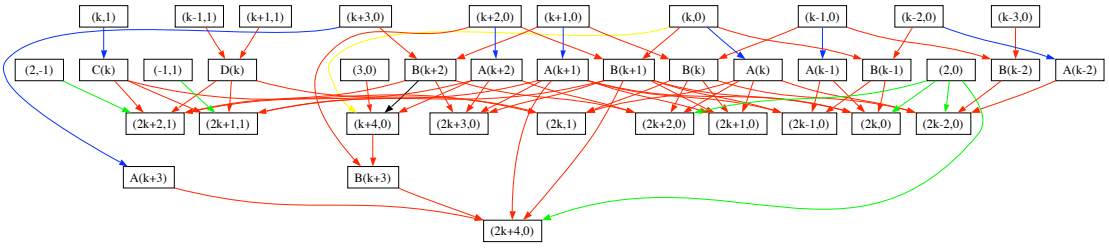

Figure 4: Dependencies for short block double-and-add.

Omitting the calculation of the $(2i + 5, 0)$ term to produce a short block (as illustrated) saves us 2 multiplications, and thus the total cost of **DoubleAddShort** is 35 multiplications (including precomputed inversions independent of k), 7 squarings and 1 inversion that depends on k.

3.3 Relative Performance

DoubleAddShort is more expensive than **DoubleAdd** yet generates less terms! Despite the savings of **DoubleShort** compared to **Double**, a purely short-block algorithm would perform worse than the standard algorithm for binary strings with a high Hamming weight, since for each Double-and-add an inversion is introduced in place of a multiplication. However, when the Hamming weight is low, then the occasional cost of an inversion is balanced by the savings accrued during short doublings. To exploit this, whilst guarding against too many inversions, we introduce an algorithm that uses a mixture of standard ('long') and short blocks.

4 Mixed block-length

4.1 Cost of Procedures

We consider the generation of long or short blocks with centre $2k$ (double) or $2k + 1$ (double-and-add) from long or short blocks of centre k. The cheapest such operation is the generation of the short block with centre $2k$ from a short or long block with centre k, at a cost of 31 multiplications, 6 squarings and no inversions. Using this as a base line, each procedure introduces the following additional operations:

Procedure	M	S	I
DoubleShortFromShort	0	0	0
DoubleLongFromShort	6	1	1
DoubleAddShortFromShort	4	1	1
DoubleAddLongFromShort	6	1	1
DoubleShortFromLong	0	0	0
DoubleLongFromLong	4	1	0
DoubleAddShortFromLong	2	1	0
DoubleAddLongFromLong	4	1	0

4.2 Heuristics

We adopt a windowing approach with two-bit windows: that is, bit b_i informs whether we are to double or double-and-add, but b_{i-1} is also examined to determine whether we should generate a long or short block.
For $b_i b_{i-1} = 00$, the short block approach clearly minimises the cost through these two bits.
For $b_i b_{i-1} = 11$, one should stay with long blocks if these are already in use, to avoid inversion. If short, adopting the long block immediately will mean only a single inversion is required for the following run of 1s.
For $b_i b_{i-1} = 10$, if short, then an inversion is required whether you go long or not: since being long is not necessary for the following double, we keep the multiplication count down by 2 by staying short. Similarly for long: no inversion is required to perform the Double-and-add for either length, but as the next operation will be a double, we go short to avoid the unnecessary 2 multiplications.
For $b_i b_{i-1} = 01$, then it is always worth staying short if you already are, defering the inversion until it is strictly required for $b_{i-1} = 1$ (possibly choosing to go long then based on b_{i-2}). If currently long, going short will save 4 multiplications and a squaring (approximately 5 multiplications). Even if it proves necessary to upgrade to long for the very next bit, that will only cost around 3.6 multiplications (based on $1I = 1.6M$, see Section 4.4). Thus even a single zero bit is worth going short for.

Hence the approach is to always go to (or stay with, if already the case) short blocks, unless $b_i b_{i-1} = 11$ in which case one should go to (or stay with) long blocks. Thus a 2-bit window is sufficient to determine appropriate block length, leading to the following algorithm.

4.3 2-bit Window Algorithm

Double-and-add Mixed-Blocks Algorithm
INPUT: Integer n and long block centred at 1.
OUTPUT: Block centred at n.

1. Compute binary digits d_i of n such that $n = (d_k d_{k-1} \dots d_1)_2$ with $d_k = 1$.

2. Set $c = 1$ (centre); Set *status* ='long'.

3. For $i = k - 1$ down to 2 do:

   - If *status* ='long'
     - If $d_i = 1$
       - If $d_{i-1} = 1$ Compute block with centre $2c + 1$ via **DoubleAddLongFromLong**; Set c to $2c + 1$.
       - Else Compute short block with centre $2c + 1$ via **DoubleAddShortFromLong**; Set c to $2c+1$; Set *status* ='short'.
     - Else
       Compute short block with centre $2c$ via **DoubleShortFromLong**; Set c to $2c$; Set *status* ='short'.
   - Else
     - If $d_i = 1$
       - If $d_{i-1} = 1$ Compute block with centre $2c + 1$ via **DoubleAddLongFromShort**; Set c to $2c+1$; Set *status* ='long'.
       - Else Compute block with centre $2c+1$ via **DoubleAddShortFromShort**; Set c to $2c + 1$.
     - Else
       Compute short block with centre $2c$ via **DoubleShortFromShort**; Set c to $2c$.

4. - If $d_1 = 1$
     - If *status* ='short': Compute block with centre $2c + 1$ via **DoubleAddShortFromShort**.
     - Else Compute block with centre $2c+1$ via **DoubleAddShortFromLong**.
   - Else Compute short block with centre $2c$ via **DoubleShortFromShort**.

5. Return final block.

4.4 Performance and notes

- The maximum possible gain is when n is a power of two, in which case the algorithm proceeds entirely by short doubles. In this case, there is a 12 percent reduction in the number of multiplications/squarings performed, with no inversions required.

- Brute-force analysis of all possible 16-bit strings gives an average reduction of around 9 percent in the number of multiplications/squarings performed. Costing each inversion at 1.6 multiplications (based on average performance in SAGE for a 256-bit prime field), this leads to an average reduction of around 5 percent in the number of multiplications required for such strings. Testing several hundred 256-bit strings gives a similar figure.

- Clearly, inversion is not viable if it will lead to a division-by-zero error. However, since the first zero along $(i, 0)$ will arise at $(m, 0)$, no such error will occur when performing Tate pairing computations.

References

[Stange] *The Tate Pairing via Elliptic Nets* K.Stange
`http://www.math.brown.edu/~stange/tatepairing/`