Double-and-Add with Relative Jacobian Coordinates

Björn Fay
mail@bfay.de

December 20, 2014

Abstract

One of the most efficient ways to implement a scalar multiplication on elliptic curves with precomputed points is to use mixed coordinates (affine and Jacobian). We show how to relax these preconditions by introducing relative Jacobian coordinates and give an algorithm to compute a scalar multiplication where the precomputed points can be given in Jacobian coordinates. We also show that this new approach is compatible with Meloni's trick, which was already used in other papers to reduce the number of multiplications needed for a double-and-add step to 18 field multiplications.

Keywords: elliptic curve, relative Jacobian coordinates, co-Z coordinates, scalar multiplication, double-and-add, precomputed points

1 Introduction

There are many possible ways to compute doubling and addition on elliptic curves. A good overview is given in the Explicit-Formulas Database [BL14]. For a generic approach normally the short Weierstrass form is used, because every curve can be written in such a form and most of the standards use it. For double-and-add algorithms there are already quite some optimizations done. Based on the trick shown by Meloni in [Mel07] to reuse some intermediate values for the next computation Longa and Miri have given in [LM08b] and [LM08a] a fast formula to compute a double-and-add step with only 18 field multiplications. Goundar et al. showed in [GJM$^+$11] how to use Meloni's trick to implement e.g.

a Montgomery ladder with only 14 field multiplications per step. Rivain also showed in [Riv11] how to implement regular signed window algorithms with this trick.

The drawback of these signed window algorithms is still that for maximum efficiency you need the precomputed points in affine coordinates, which needs an additional inversion for the precomputation. We introduce a new variant of (modified) Jacobian coordinates which circumvents this shortcoming. We call these coordinates relative (modified) Jacobian coordinates, because the Z-coordinate is given relative to a (common) Z-coordinate.

The rest of the paper is structured as follows. In section 2 we provide the basic formulas for (modified) Jacobian coordinates from which we start to introduce our new coordinate system. In section 3 we introduce the new relative coordinates, apply Meloni's trick and give a full double-and-add algorithm for scalar multiplication. And finally in section 4 we summarize what we achieved in this paper.

2 Basic Formulas

We just start by looking at the normal formulas for an elliptic curve E in short Weierstrass form. So let E be an elliptic curve defined by the equation $y^2 = x^3+ax+b$ over $K = \mathrm{GF}(p^n)$ with $p > 3$, $n \in \mathbb{N}$ and $4a^3+27b^2 \neq 0$. To add two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \neq \pm P_1$ you have to compute $P_3 = (x_3, y_3)$ with $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{y_1-y_2}{x_1-x_2}$. To double a point $P_1 = (x_1, y_1)$ you have to compute $P_3 = (x_3, y_3)$ with $x_3 = \lambda^2 - 2x_1$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{3x_1^2+a}{2y_1}$. If we transform these equations into modified Jacobian coordinates ($P = (X : Y : Z : aZ^4)$, with $X = xZ^2$, $Y = yZ^3$ and $Z \in K^*$) and assume common Z-coordinates ($Z_1 = Z_2$), we get the following equations for an addition:

$$\begin{aligned}
L &= Y_1 - Y_2 && (1)\\
Z_3 &= (X_1 - X_2)Z_1 && (2)\\
X_3 &= L^2 - (X_1 + X_2)(X_1 - X_2)^2 && (3)\\
Y_3 &= L(X_1(X_1 - X_2)^2 - X_3) - Y_1(X_1 - X_2)^3 && (4)\\
aZ_3^4 &= (X_1 - X_2)^4 aZ_1^4 && (5)
\end{aligned}$$

For normal Jacobian coordinates you can just drop the last equation. For a doubling we get the following equations:

$$\begin{aligned}
L &= 3X_1^2 + aZ_1^4 & (6)\\
Z_3 &= 2Y_1Z_1 & (7)\\
X_3 &= L^2 - 8X_1Y_1^2 & (8)\\
Y_3 &= L(4X_1Y_1^2 - X_3) - 8Y_1^4 & (9)\\
aZ_3^4 &= 16Y_1^4aZ_1^4 & (10)
\end{aligned}$$

And again you can drop the last equation if you just want to compute normal Jacobian coordinates.

3 Double-and-Add

Let us now have a look at different (left-to-right) double-and-add algorithms. There are several flavors (e.g. sliding window method), but they all have in common that they use some number of precomputed points (for simple double-and-add this is just the base point), from which they chose one per double-and-add step to add to the accumulated point. Let us denote this chosen precomputed point per step with P_0 and further define that all these precomputed points shall have a common Z-coordinate ($Z_0 = 1$ for affine, but works also for Jacobian coordinates), which is easy to achieve by some field multiplications (no inversion needed). Further we also need the fourth component of the modified Jacobian coordinates aZ_0^4 for doublings, but which is of course also the same for all precomputed points. The accumulated point we denote with P_1.

To be able to use the addition formulas from the previous section we have to ensure that P_0 and P_1 have a common Z-coordinate. To achieve this we store the point P_1 not in normal (modified) Jacobian coordinates but in relative (modified) Jacobian coordinates. This means that we store P_1 as $(X_1 : Y_1 : Z'_1 : aZ_1^4)$, so that $Z_1 = Z_0Z'_1$ (in the very first step we have $Z'_1 = 1$) and the fourth component is not always computed (only for doublings, see further down for more details). With this we can easily compute $P_2 = (X_0Z_1'^2 : Y_0Z_1'^3 : Z_0Z'_1 : aZ_1^4)$, which has the same Z-coordinate as P_1, but where $Z_0Z'_1 = Z_1$ and aZ_1^4 are not needed. This computation of P_2 needs 4 field multiplications (for X and Y coordinates). For the computation in equation 2 and 7 you now just have to replace all Zs by Z's. For the doubling you also have to compute aZ_1^4 slightly differently now as $aZ_1^4 = Z_1'^4aZ_0^4$.

Now we also use Meloni's trick. So we see that in equations 2 and 4 we already have the coordinates $(X_1(X_1 - X_2)^2 : Y_1(X_1 - X_2)^3 : (X_1 - X_2)Z'_1)$ of P_1 having a common (relative) Z-coordinate with the result P_3. Now instead of computing $2P_1 + P_2$ we compute $(P_1 + P_2) + P_1$ for a double-and-add step. This means that the first addition is a normal one (where we first have to compute P_2 from P_0 as described above), but the second addition can make full use of the common (relative) Z-coordinate of this new representation of P_1 (so we can start directly with the result P_3 as new P_2). This means that we can replace a doubling by an addition (with only 7 field multiplications).

If we now look at the whole double-and-add algorithm we see some further facts. If you are making an irregular algorithm (e.g. for signature verification optimized for speed) you have to mix these double-and-add steps with some normal double steps. For these you start for the first of the consecutive doublings with Jacobian coordinates and want to compute modified Jacobian coordinates to speed up further doublings. For the last doubling in such a row you do not want to compute aZ_3^4 anymore, because it will not be needed for the next double-and-add step. In the case of $a = -3$ you can optimize the computation of equation 6 and save one field multiplication by computing

$$L = 3X_1^2 - 3Z_1^4 = 3(X_1 + Z_1^2)(X_1 - Z_1^2) = 3(X_1 + Z_0^2 Z_1'^2)(X_1 - Z_0^2 Z_1'^2)$$

which means that you have to store Z_0^2 in addition to aZ_0^4 as (common) fifth component of the precomputed points. But please note that you can only use this optimization if you have to compute a single double step, because you cannot compute aZ_3^4 anymore. For the ease of the algorithm, we do not care further about this optimization (algorithm 4 shows the steps for this computation).

If you want to make a regular double-and-(always)-add algorithm, you have the drawback that you cannot use dummy additions anymore if you use Meloni's trick. But you can work around that by doing some scalar recoding as shown e.g. in [Riv11], which gets rid of all zero entries in the scalar.

Putting it all together, we give here a complete double-and-add algorithm, where the individual steps can be optimized for the given platform (trade-off between multiplications, squarings and additions). We start with the overall algorithm 1 and afterwards present the building blocks (algorithms 2 and 3).

Please note that step 7 of algorithm 1 might not always work with algorithm 2 because R_{k_i} might be $\pm P_1$. In this case you have to do an ordinary double-and-add using algorithm 3 and algorithm 2 skipping step 19. In case $R_{k_i} = 2P_1$ you can take algorithm 2 again and for $R_{k_i} = -2P_1$ you can restart by setting $P_1 = R_{k_{i-1}}$ and skipping the next step. The cases that can occur depend on the used double-and-add variant and recoding of the scalar. A careful selection can avoid these cases at least after the first few steps and also reduce the possible cases in the first few steps so that the ordinary double-and-add can be used without any extra cases for the first few steps and then switch to algorithm 2. If different timing is not a problem then you can also check in steps 6 and 7 of algorithm 2 for zero to recognize in which case you are and react accordingly.

Algorithm 1 can in principle be used for regular scalar multiplication, where all $k_i \neq 0$, or for performance optimized implementation, where some or most of the $k_i = 0$, e.g. for

Algorithm 1: Scalar Multiplication

```
   Input: precomputed points $R_i$ (in Jacobian coordinates)
          (recoded) scalar $k = (k_n, \ldots, k_1), k_n \neq 0$
   Output: $kP$
1  align $R_i$ to have a common $Z$-coordinate $Z_0$
2  compute $aZ_0^4$
3  $P_1 = R_{k_n}$
4  $Z_1' = 1$                    // $P_1$ in relative Jacobian coordinates
5  for $i = n-1$ to 1 do
6      if $k_i \neq 0$ then
7          $P_1 = 2P_1 + R_{k_i}$  // using algorithm 2 (might not always be possible)
8      else
9          $P_1 = 2P_1$            // using algorithm 3
10     end if
11 end for
12 return $P_1$
```

sliding window NAF. It only depends on the recoding of the scalar and the precomputed points. Of course if you only have $k_i \neq 0$ you can drop the computation of aZ_0^4, which is only needed for algorithm 3. And if on top you only have two precomputed points, e.g. R_1 and $R_{-1} = -R_1$, then you should better use e.g. algorithm 8 in [Riv11], which needs only 14 field multiplications per scalar bit. For the building blocks (algorithms 2 and 3) you need 4 auxiliary field registers (called S, T, U, V).

Algorithms 2, 3 and 4 are not optimized e.g. to use squarings instead of multiplications (the trade off there depends on the used platform; also field additions are not for free). The only optimization that was done is to enable implementation of operations that are not in-place and otherwise get a nice structure. So depending on the target platform other optimizations may be needed. The given algorithms need (M, S, A are standing for multiplications, squarings and additions respectively):

- $13M + 5S + 14A$ for a double-and-add step,
- $d(4M + 4S + 12A) + 2S - 1A$ for d consecutive doublings,
- $5M + 4S + 12A$ for a single doubling with $a = -3$.

The scalar multiplication needs 2 field-registers per precomputed point plus 1 extra register to store Z_0 and for irregular implementations (some $k_i = 0$) another register to store aZ_0^4 (plus another one for Z_0^2 if algorithm 4 is used). Further you need 3 registers to store P_1 and another 4 as auxiliary registers for the point operations.

Algorithm 2: Double-and-Add

```
   Input: $P_0 = (X_0 : Y_0 : Z_0)$ (in Jacobian coordinates)
          $P_1 = (X_1 : Y_1 : Z'_1)$ (in relative Jacobian coordinates, $Z_1 = Z_0 Z'_1$)
   Output: $2P_1 + P_0$
1  $S = Z_1'^2$              // $S = Z_1'^2$
2  $T = Z'_1 S$              // $T = Z_1'^3$
3  $U = X_0 S$               // $U = X_2$
4  $V = Y_0 T$               // $V = Y_2$
5  $T = Y_1 - V$             // $T = L = Y_1 - Y_2$
6  $V = X_1 - U$             // $V = X_1 - X_2$
7  $Z'_1 = V Z'_1$           // $Z'_1 = (X_1 - X_2) Z'_1 = Z'_3$ (can use $S$ as temp)
8  $S = V^2$                 // $S = (X_1 - X_2)^2$
9  $V = US$                  // $V = X_2(X_1 - X_2)^2$
10 $U = X_1 S$               // $U = X_1(X_1 - X_2)^2$
11 $X_1 = T^2$               // $X_1 = L^2$
12 $S = X_1 - V$             // $S = L^2 - X_2(X_1 - X_2)^2$
13 $X_1 = S - U$             // $X_1 = L^2 - (X_1 + X_2)(X_1 - X_2)^2 = X_3$
14 $S = U - V$               // $S = (X_1 - X_2)^3$
15 $V = Y_1 S$               // $V = Y_1(X_1 - X_2)^3$
16 $Y_1 = U - X_1$           // $Y_1 = X_1(X_1 - X_2)^2 - X_3$
17 $S = T Y_1$               // $S = L(X_1(X_1 - X_2)^2 - X_3)$
18 $Y_1 = S - V$             // $Y_1 = L(X_1(X_1 - X_2)^2 - X_3) - Y_1(X_1 - X_2)^3 = Y_3$
19 repeat steps 5 to 18 once (computing $P_3 + P_1$)
20 return $P_1$
```

Algorithm 3: Double

```
Input: $P_1 = (X_1 : Y_1 : Z'_1)$ (in relative Jacobian coordinates, $Z_1 = Z_0 Z'_1$)
       $aZ_0^4$
       $S = aZ_1^4$ if preceded by another double
Output: $2P_1$
        $S = aZ_3^4$ if followed by another double
1  if first double then
2      $S = Z_1'^2$                // $S = Z_1'^2$
3      $T = S^2$                   // $T = Z_1'^4$
4      $S = TaZ_0^4$               // $S = Z_1'^4 aZ_0^4 = aZ_1^4$
5  end if
6  $U = X_1^2$                     // $U = X_1^2$
7  $T = U + U$                     // $T = 2X_1^2$
8  $V = T + U$                     // $V = 3X_1^2$
9  $T = V + S$                     // $T = 3X_1^2 + aZ_1^4 = L$
10 $V = Y_1 Z'_1$                  // $V = Y_1 Z'_1$
11 $Z'_1 = V + V$                  // $Z'_1 = 2Y_1 Z'_1 = Z'_3$
12 $V = Y_1^2$                     // $V = Y_1^2$
13 $Y_1 = V + V$                   // $Y_1 = 2Y_1^2$
14 $U = X_1 Y_1$                   // $U = 2X_1 Y_1^2$
15 $V = U + U$                     // $V = 4X_1 Y_1^2$
16 $X_1 = T^2$                     // $X_1 = L^2$
17 $U = X_1 - V$                   // $U = L^2 - 4X_1 Y_1^2$
18 $X_1 = U - V$                   // $X_1 = L^2 - 8X_1 Y_1^2 = X_3$
19 $U = V - X_1$                   // $U = 4X_1 Y_1^2 - X_3$
20 $V = TU$                        // $V = L(4X_1 Y_1^2 - X_3)$
21 $U = Y_1^2$                     // $U = 4Y_1^4$
22 $T = U + U$                     // $T = 8Y_1^4$
23 $Y_1 = V - T$                   // $Y_1 = L(4X_1 Y_1^2 - X_3) - 8Y_1^4 = Y_3$
24 if not last double then
25     $U = TS$                    // $U = 8Y_1^4 aZ_1^4$
26     $S = U + U$                 // $S = 16Y_1^4 aZ_1^4 = aZ_3^4$
27 end if
28 return $P_1$
```

Algorithm 4: Single Double for $a = -3$

```
Input: $P_1 = (X_1 : Y_1 : Z'_1)$ (in relative Jacobian coordinates, $Z_1 = Z_0 Z'_1$)
       $Z_0^2$
Output: $2P_1$
1  $S = Z_1'^2$                // $S = Z_1'^2$
2  $T = SZ_0^2$                // $T = Z_1'^2 Z_0^2 = Z_1^2$
3  $U = X_1 + T$               // $U = X_1 + Z_1^2$
4  $V = X_1 - T$               // $V = X_1 - Z_1^2$
5  $S = UV$                    // $S = X_1^2 - Z_1^4$
6  $U = S + S$                 // $U = 2(X_1^2 - Z_1^4)$
7  $T = U + S$                 // $T = 3(X_1^2 - Z_1^4) = L$
8  $V = Y_1 Z'_1$              // $V = Y_1 Z'_1$
9  $Z'_1 = V + V$              // $Z'_1 = 2Y_1 Z'_1 = Z'_3$
10 $V = Y_1^2$                 // $V = Y_1^2$
11 $Y_1 = V + V$               // $Y_1 = 2Y_1^2$
12 $U = X_1 Y_1$               // $U = 2X_1 Y_1^2$
13 $V = U + U$                 // $V = 4X_1 Y_1^2$
14 $X_1 = T^2$                 // $X_1 = L^2$
15 $U = X_1 - V$               // $U = L^2 - 4X_1 Y_1^2$
16 $X_1 = U - V$               // $X_1 = L^2 - 8X_1 Y_1^2 = X_3$
17 $U = V - X_1$               // $U = 4X_1 Y_1^2 - X_3$
18 $V = TU$                    // $V = L(4X_1 Y_1^2 - X_3)$
19 $U = Y_1^2$                 // $U = 4Y_1^4$
20 $T = U + U$                 // $T = 8Y_1^4$
21 $Y_1 = V - T$               // $Y_1 = L(4X_1 Y_1^2 - X_3) - 8Y_1^4 = Y_3$
22 return $P_1$
```

4 Conclusion

We have shown how to modify normal formulas for Jacobian coordinates to get the same efficiency as formulas with mixed coordinates by introducing relative Jacobian coordinates. We also have shown that these new coordinates can be used together with Meloni's trick in [Mel07] by giving a complete algorithm for a scalar multiplication, which needs as input only precomputed points in Jacobian coordinates (not affine) and an accordingly recoded scalar. The double-and-add step can be done with 18 field multiplications and the doublings with 8 field multiplications per doubling plus 2 extra multiplications for the first doubling. For a regular implementation (all $k_i \neq 0$) with r precomputed points you need $2r + 8$ field registers. For an irregular implementation (some $k_i = 0$) you need one extra register, in total $2r + 9$ field registers.

References

[BL14] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. `http://hyperelliptic.org/EFD`, 2014.

[GJM+11] Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on weierstraß elliptic curves from co-z arithmetic. *J. Cryptographic Engineering*, 1(2):161–176, 2011.

[LM08a] Patrick Longa and Ali Miri. New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields (full version). *IACR Cryptology ePrint Archive*, 2008:51, 2008.

[LM08b] Patrick Longa and Ali Miri. New multibase non-adjacent form scalar multiplication and its application to elliptic curve cryptosystems (extended version). *IACR Cryptology ePrint Archive*, 2008:52, 2008.

[Mel07] Nicolas Meloni. New point addition formulae for ecc applications. In Claude Carlet and Berk Sunar, editors, *WAIFI*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2007.

[Riv11] Matthieu Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptology ePrint Archive*, 2011:338, 2011.