université PARIS-SACLAY

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

GLV method

Simon Masson

Contents

February 22, 2017

Introduction

This document is a study of [3]. The goal of this project is to implement an algorithm used in elliptic curves, using the GMP library. Almost all of the GMP functions used in this project are explained in [2].

The first section explains the general method, the second one gives some details about the implementation, and the last one shows experimental results and comparaisons.

1 GLV method

The main operation in elliptic curves used in cryptography is the scalar multiplication, i.e computing kP where k is an integer and P is a point of the curve. A first way to compute kP is to use a square–and–multiply, but in our case, in additive notations, we tell it a double–and–add. It is clearly better than the naive method, but we can find a better way in some cases with the algorithm presented in this article.

Let E be an elliptic curve in a finite prime field, and P a point of E of order n. Note that in cryptography, the number of points in the curve is often a prime number, so every $P \neq \mathcal{O}$ is of order n. We can separate the GLV method in three steps :

1. Finding an endomorphism φ of the curve that satisfies $\varphi(Q) = [\lambda]Q$ for all $Q \in \langle P \rangle$, with λ a root of the characteristic polynomial χ_φ.
2. Decomposing $k = k_1 + \lambda k_2 \mod n$ with k_1, k_2 the shortest possible.
3. Computing kP using the couple (k_1, k_2) and the endomorphism φ.

1.1 Endomorphism of a curve

Definition (endomorphism)**.** An *endomorphism* of E is a rational map $\varphi : E \longrightarrow E$ satisfying $\varphi(\mathcal{O}) = \mathcal{O}$. If the rational map is defined over $\mathbb{F}_q$, φ is also said to be defined over $\mathbb{F}_q$, and is a group homomorphism of $E(\mathbb{F}_q)$.

In order to apply the GLV method, we have to find an endomorphism φ such that his characteristic polynomial χ_φ has roots mod n.

Remark. As $\deg(\chi_\varphi) = 2$, finding a root mod n is not more difficult than finding elements that are square mod n. Half of the elements are square, so this is possible.

Let λ be a root of χ_φ such that $\varphi(P) = \lambda P$. Then, as $\langle P \rangle$ is a cyclic group (isomorph to $\mathbb{Z}/n\mathbb{Z}$), the value of $\varphi(Q)$ for $Q \in \langle P \rangle$ is entirely defined by his value in P :

$$\varphi(Q) = \varphi(mP) = m\varphi(P) = m\lambda P = \lambda mP = \lambda Q$$

Example 1. Let $p \equiv 1 \mod 4$ be a prime number, and consider the elliptic curve

$$E_1 : y^2 = x^3 + ax$$

defined overe $\mathbb{F}_p$.
As $p-1$ is a multiple of 4, $(\mathbb{F}_p)^* \simeq \mathbb{Z}/(p-1)\mathbb{Z}$ has a subgroup $\mathbb{Z}/4\mathbb{Z}$, and so we can find $\alpha \in \mathbb{F}_p$ of

order 4. Then, we define φ by

$$\begin{array}{rccc} \varphi: & E_1 & \longrightarrow & E_1 \\ & (x,y) & \longmapsto & (-x, \alpha y) \\ & \mathcal{O} & \longmapsto & \mathcal{O} \end{array}$$

Remark that $\varphi^2(x,y) = (x, \alpha^2 y)$, and $\alpha^2 = -1$ because α is of order 4. That means that $\varphi^2(P) = -P$ for all $P \neq \mathcal{O}$. We found the characteristic polynomial X^2+1. Let $\lambda, \tilde{\lambda}$ be the roots of X^2+1 in $\mathbb{Z}/n\mathbb{Z}$. φ is an endomorphism of $\langle P \rangle$ that is a cyclic group, so it is entirely defined by his value in a generator (P), and so we find that

$$\varphi(Q) = [\lambda]Q \qquad \forall Q \in \langle P \rangle \qquad \lambda^2 = -1 \mod n$$

To compute $\varphi(x,y)$, we have to compute only $\alpha \times y$, that is very fast !

Example 2. Let $p \equiv 1 \mod 3$ be a prime number, and consider the elliptic curve

$$E_2 : y^2 = x^3 + b$$

defined overe $\mathbb{F}_p$.
As in the previous example, we can find $\beta \in \mathbb{F}_p$ of order 3. We define then φ by

$$\begin{array}{rccc} \varphi: & E_2 & \longrightarrow & E_2 \\ & (x,y) & \longmapsto & (\beta x, y) \\ & \mathcal{O} & \longmapsto & \mathcal{O} \end{array}$$

We find here $\varphi^2(P) + \varphi(P) + P = \mathcal{O}$. Indeed, if $P = (x,y)$, the addition formula gives

$$\varphi^2(P) + \varphi(P) + P = (\beta^2 x, y) + (\beta x, y) + (x,y) = (-\beta^2 x - \beta x, -y) + (x,y) = (x,-y) + (x,y) = \mathcal{O}$$

(we use the fact that $1 + \beta + \beta^2 = 0$ and then $P + (-P) = \mathcal{O}$)
If λ is such that $\lambda^2 + \lambda + 1 \equiv 0 \mod n$, then the same argument as in the previous example gives that

$$\varphi(Q) = [\lambda]Q \qquad \forall Q \in \langle P \rangle$$

Again, computing $\varphi(x,y)$ is not more cumbersome than computing $\beta \times x$.

Remark. This second example is computed in the subsection 2.6. In particular, we explain how to find an element $\beta \in \mathbb{F}_p$ of order 3.

We found φ such that

$$\varphi(Q) = [\lambda]Q \qquad \forall Q \in \langle P \rangle \qquad \chi_\varphi(\lambda) = 0$$

Then, we use this λ to decompose the integer $k \bmod n$, in order to speed up the algorithm of kP.

1.2 Decomposing k

In this section, we want to decompose $k = k_1 + \lambda k_2 \mod n$, with k_1 and k_2 the shortest possible, i.e the vector $(k_1, k_2) \in \mathbb{Z} \times \mathbb{Z}$ has "small" Euclidian norm.

Let $G = \mathbb{Z} \times \mathbb{Z}$ and consider the homomorphism

$$\begin{array}{rccl} f: & G & \longrightarrow & \mathbb{Z}/n\mathbb{Z} \\ & (i,j) & \longmapsto & (i+\lambda j) \mod n \end{array}$$

We wish to find a short vector $u \in G$ such that $f(u) = k$. Then, the components of u can be used for k_1 and k_2. We can easily find a vector $v \in G$ such that $f(v) = k$ because $(k, 0)$ works, but the problem is to find a short one.

We are going to use the extented Euclidian algorithm to find u. This is divided in three steps :

1. Find $v_1, v_2 \in G$ (short and independant vectors) such that $f(v_1) = f(v_2) = 0$.
2. Find v in the lattice generated by v_1 and v_2 that is close to $(k, 0)$. Define $u := (k, 0) - v$.
3. u is a short vector that satisfies $f(u) = k$.

$\gcd(n, \lambda) = 1$ because n is prime. Using the extented Euclidian algorithm on n and λ (see 2.2 for details), we produce sequences (s_i, t_i, r_i) such that

$$s_i n + t_i \lambda = r_i \qquad i = 0, 1, 2, \ldots$$

where $s_0 = 1, t_0 = 0, r_0 = n, s_1 = 0, t_1 = 1, r_1 = \lambda$, and $r_i \geq 0$ for all i.

Lemma. *Let s_i, t_i, r_i be the sequence of variables produced by the extended Euclidian algorithm on n and λ. Them,*

1. *$r_i > r_{i+1} \geq 0$ for all $i \geq 0$.*
2. *$|s_i| < |s_{i+1}|$ for $i \geq 1$.*
   $|t_i| < |t_{i+1}|$ for $i \geq 2$ [1].
3. *$t_i t_{i+1} \leq 0$ for all $i \geq 0$.*
4. *$r_i |t_{i+1}| + r_{i+1} |t_i| = n$ for all $i \geq 0$.*

Proof. We proove the four properties by induction. We note q_ℓ the quotient in the euclidian division of r_ℓ by $r_{\ell+1}$. $(r_i)_{i \in \mathbb{N}}$ are the successive remainders in the euclidian algorithm, and n and λ are positive integers, so $(q_i)_{i \in \mathbb{N}}$ are also positive.

1. $n = r_0 > r_1 = \lambda \geq 0$.
   Suppose that there exists $k \in \mathbb{N}$ such that $r_k > r_{k+1} \geq 0$. Then, r_{k+2} is the remainder of the euclidian division of r_k by r_{k+1}, so $0 \leq r_{k+2} < r_{k+1}$.
2. $0 = |s_1| < |s_2| = |s_0 - q_0 s_1| = |s_0| = 1$.
   $0 = |t_0| < |t_1| = 1$.
   Suppose that there exists $k \in \mathbb{N}^*$ such that $|s_k| < |s_{k+1}|$. Then,

$$|s_{k+2}| = |s_k - q_k s_{k+1}| \geq ||s_k| - q_k |s_{k+1}|| = q_k |s_{k+1}| - |s_k| > q_k |s_{k+1}| \geq |s_{k+1}|$$

[1] The article [3] gives $i \geq 0$ for t_i but it looks to be a mistake for t_0 : the inequality is not strict. See the proof for further precisions.

because $|s_k| > 0$ and the quotients in the euclidian divisions are all ≥ 1 because the divisions are done from an integer, by a smaller one (even at first step with n and λ because we use for λ the representant in $[\![0, n[\![$).
The proof is exactly the same for $(t_i)_{i\geq 2}$. We begin with $i \geq 2$ because $|t_0| = 0$ and so the strict inequality does not take place.

3. $t_0 t_1 = 0 \times 1 = 0$.
Suppose that there exists $k \in \mathbb{N}$ such that $t_k t_{k+1} \leq 0$. Then,

$$t_{k+1}t_{k+2} = t_{k+1}(t_k - q_k t_{k+1}) = t_{k+1}t_k - \underbrace{q_k t_{k+1}^2}_{\geq 0} \leq t_k t_{k+1} \overset{\text{induc.}}{\leq} 0$$

4. $r_0|t_1| + r_1|t_0| = n \times 1 + \lambda \times 0 = n$.
Suppose that there exists $k \in \mathbb{N}$ such that $r_k|t_{k+1}| + r_{k+1}|t_k| = n$. We can write

$$r_{k+1}|t_{k+2}| + r_{k+2}|t_{k+1}| = r_{k+1}|t_k - q_k t_{k+1}| + (r_k - q_k r_{k+1})|t_{k+1}|$$

Then, it is easy to proove that $|t_k - q_k t_{k+1}| = |t_k| + q_k|t_{k+1}|$. Indeed,

- If $t_k > 0$, $t_{k+1} \leq 0$ (by 3.) and $|t_k| + q_k|t_{k+1}| = t_k - q_k t_{k+1}$.
We also have $t_k > 0 \geq q_k t_{k+1}$, and so $|t_k - q_k t_{k+1}| = t_k - q_k t_{k+1}$.
- If $t_k \leq 0$, $t_{k+1} \geq 0$ (by 3.) and $|t_k| + q_k|t_{k+1}| = -t_k + q_k t_{k+1}$.
We also have $t_k < 0 \leq q_k t_{k+1}$ and so $|t_{k-1} - q_{k-1}t_{k+1}| = -t_k + q_k t_{k+1}$.

Now we can compute :

$$\begin{aligned} r_{k+1}|t_{k+2}| + r_{k+2}|t_{k+1}| &= r_{k+1}|t_k - q_k t_{k+1}| + (r_k - q_k r_{k+1})|t_{k+1}| \\ &= r_{k+1}(|t_k| + q_k|t_{k+1}|) + (r_k - q_k r_{k+1})|t_{k+1}| \\ &= r_{k+1}|t_k| + r_{k+1}q_k|t_{k+1}| + r_k|t_{k+1}| - q_k r_{k+1}|t_{k+1}| \\ &= r_k|t_{k+1}| + r_{k+1}|t_k| = n \end{aligned}$$

□

1.2.1 Find v_1 and v_2

Let m be the greatest index for which $r_m \geq \sqrt{n}$. Then, by the previous lemma,

$$r_m|t_{m+1}| + r_{m+1}|t_m| = n$$

and so $r_m|t_{m+1}| < n$ and it follows that $|t_{m+1}| < \frac{n}{\sqrt{n}} = \sqrt{n}$.
We take $v_1 = (r_{m+1}, -t_{m+1})$. We verify that $f(v_1) = 0$:

$$f(v_1) = r_{m+1} - \lambda t_{m+1} = s_{m+1}n \equiv 0 \mod n$$

Also, since $|t_{m+1}| < \sqrt{n}$ and $|r_{m+1}| < \sqrt{n}$, we have $\|v_1\| \leq \sqrt{2n}$.
We also take v_2 to be the shorter of $(r_m, -t_m)$ and $(r_{m+2}, -t_{m+2})$. we proove similarly that $f(v_2) = 0$. Heuristically, we expect that v_2 is also short, but we can't proove it without further

restictions. Experiments with various values of λ validate this assumption.
v_1 and v_2 are linearly independant : *if they did not* (suppose $v_2 = (r_m, -t_m)$), then

$$\frac{r_{m+1}}{r_m} = \frac{-t_{m+1}}{-t_m} = \frac{t_{m+1}}{t_m}$$

but $r_{m+1}/r_m < 1$ and $|t_{m+1}/t_m| > 1$ by 3. by the previous lemma. *Impossible.*

Remark. Since v_1 and v_2 only depend on n and λ (and not on k), they can be precomputed if n and λ are shared domain parameters.

1.2.2 Find v

We set v in the integer lattice generated by v_1 and v_2, that is closed to $(k, 0)$. First, write $(k, 0)$ in the basis (v_1, v_2) :

$$(k, 0) = \beta_1 v_1 + \beta_2 v_2 \qquad \beta_1, \beta_2 \in \mathbb{Q}$$

Then, set $b_1 := \lfloor \beta_1 \rceil$, $b_2 := \lfloor \beta_2 \rceil$ (the nearest integers to β_1, β_2).
Finally, set $v = b_1 v_1 + b_2 v_2$ and $u := (k, 0) - v$.
The implementation of this part is described in 2.3.

1.2.3 u is a solution of the problem

The vector u constructed just before has norm at most $\max(\|v_1\|, \|v_2\|)$. Indeed,

$$u = (k, 0) - v = (\beta_1 v_1 + \beta_2 v_2) - (b_1 v_1 + b_2 v_2) = (\beta_1 - b_1)v_1 + (\beta_2 - b_2)v_2$$

and since $|\beta_1 - b_1| \leq \frac{1}{2}$ and $|\beta_2 - b_2| \leq \frac{1}{2}$, by the Triangle Inequality we have

$$\|u\| \leq \frac{1}{2}\|v_1\| + \frac{1}{2}\|v_2\| \leq \max(\|v_1\|, \|v_2\|)$$

u satisfies $f(u) = k$ because $f(u) = f((k, 0)) - f(v) = k - 0 = k$. The coordinates of u are the integers k_1 and k_2 wanted.

1.3 Computing kP

We found a couple $(k_1, k_2) \in \mathbb{Z}^2$ such that $k \equiv k_1 + \lambda k_2 \mod n$. As P is a point of order n, we have (if $k = k_1 + \lambda k_2 + An$)

$$kP = (k_1 + \lambda k_2 + An)P = k_1 P + k_2 \lambda P + AnP = k_1 P + k_2 \varphi(P) + \mathcal{O} = k_1 P + k_2 \varphi(P)$$

As k_1 and k_2 have been found the shortest possible, computing $k_1 P$ and $k_2 \varphi(P)$ is less expensive than kP. Also, we use a simultaneous multiplication algorithm to speed up this step.
All of this is described in 2.5.

2 Algorithms

2.1 Curve addition in jacobian coordinates

Implementing the addition in the curve is an important part. There is a lot of templates to compute $P+Q$, using many multiplications and additions in $\mathbb{F}_p$. We implement the Jacobian projective coordinates, recommended by [1]. The algorithm is explained in [1]. Let E an elliptic curve over $\mathbb{F}_p$ be

$$E : y^2 = x^3 + ax + b \qquad (a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0)$$

The jacobian coordinates sets $x = X/Z^2$ and $y = Y/Z^3$, i.e

$$E : Y^2 = X^3 + aXZ^4 + bZ^6$$

The addition formulas in the jacobian coordinates are the following. Let $P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2)$ and $P + Q = R = (X_3, Y_3, Z_3)$.

- Curve addition formula in jacobian coordinates ($P \neq \pm Q$)

  $$X_3 = -H^3 - 2U_1H^2 + r^2, Y_3 = -S_1H^3 + r(U_1H^2 - X_3), Z_3 = Z_1Z_2H$$

  where $U_1 = X_1Z_2^2, U_2 = X_2Z_1^2, S_1 = Y_1Z_2^3, S_2 = Y_2Z_1^3, H = U_2 - U_1, r = S_2 - S_1$.

- Curve doubling formula in jacobian coordinates ($R = 2P$)

  $$X_3 = T, Y_3 = -8Y_1^4 + M(S - T), Z_3 = 2Y_1Z_1$$

  where $S = 4X_1Y_1^2, M = 3X_1^2 + aZ_1^4, T = -2S + M^2$.

The addition formula in jacobian coordinates is similar to projective coordinate : it does not require any division modulo p in either addition or doubling and does require a division only once in the final stage of the computation of elliptic curve exponentiation (because $x = X/Z^2$ and $y = Y/Z^3$).

However, jacobian coordinates offer a doubling with less computation amount but an addition with more computation amount than projective coordinates. This feature should be suitable for elliptic curve exponentiation since the number of additions required in elliptic curve exponentiation can be reduced by algorithms explained in this paper : during a double–and–add algorithm, the number of add correspond to the number of 1 in the binary representation. We present in 2.5 a signed binary representation (with 1, 0, and -1) that has lots of zeros, so the number of add can be reduced with this algorithm.

2.2 Extended Euclidian algorithm

We use the extended Euclidian algorithm to find a couple (k_1, k_2) such that $k = k_1 + \lambda k_2 \mod n$. We compute $\gcd(n, \lambda)$ with this algorithm : First, we write

$$n = 1 \times n + 0 \times \lambda$$

$$\lambda = 0 \times n + 1 \times \lambda$$

and then, we compute the rest of the euclidian division of n by λ, keeping the coefficients of n and λ on the right part :

$$n - q\lambda = (1 - q \times 0)n + (0 - q \times 1)\lambda$$

In the extended Euclidian algorithm, we do it untill we find a zero remainder. The second step is the euclidian division of λ by $n - q\lambda$, etc. This algorithm gives us a sequence $(r_i, s_i, t_i)_{i \geq 0}$ such that

$$r_i = s_i n + t_i \lambda$$

An equation $r_k = s_k n + t_k \lambda$ is determined by the two previous. Suppose that we have variables satisfying

```
rem = u * n + v * lambda
rrem = uu * n + vv * lambda
```

We have to replace the variables of the first line by the ones of the second line, and modify the second line by the algorithm described before. We must use temporary variables in order to save `rrem`, `uu`, and `vv`.

1. Compute `q` that is the quotient of `rem` by `rrem` : $\texttt{q} \leftarrow \texttt{rem}/\texttt{rrem}$.
2. Store the value of `rrem` in a temporary variable `t` : $\texttt{t} \leftarrow \texttt{rrem}$.
3. `rrem` is the rest of the Euclidian division of `rem` by `rrem` : $\texttt{rrem} \leftarrow \texttt{rem} - \texttt{q} \times \texttt{rrem}$.
4. Modify `rem` by the old value of `rrem` : $\texttt{rem} \leftarrow \texttt{t}$.
5. Repeat 2., 3., 4. for `u`,`uu` and then for `v`,`vv`.

In our case, we use a partial extended Euclidian algorithm because we stop the algorithm when the remainder is the last one bigger than $\sqrt{n}$. That is why we use a loop `while`$(\texttt{rrem} > \sqrt{n})$. After the loop, $\texttt{rrem} = \texttt{uu} \times n + \texttt{vv} \times \lambda$ is the first step where $\texttt{rrem} \leq \sqrt{n}$.
Then, we set $v_1 = (r_{m+1}, -t_{m+1}) = (\texttt{rrem}, -\texttt{vv})$ and v_2 be the shorter of $(r_m, -t_m) = (\texttt{rem}, -\texttt{v})$ and $(r_{m+2}, -t_{m+2})$, i.e we do another step on the algorithm :

$$\texttt{q} \leftarrow \texttt{rem}/\texttt{rrem} \qquad (r_{m+2}, -t_{m+2}) = (\texttt{rem} - \texttt{q} \times \texttt{rrem}, -(\texttt{v} - \texttt{q} \times \texttt{vv}))$$

To find the shorter, we compare the square of their norms : $r_m^2 + t_m^2$ and $r_{m+2}^2 + t_{m+2}^2$.

Remark. Either $v_{2,y}$ is set to $-\texttt{vv}$, either it is set to $-(\texttt{v} - \texttt{q} \times \texttt{vv}) = -t_{m+2}$. To compare the norms, we have to compute $-t_{m+2}$. That is why we first set $v_{2,y} \leftarrow -(\texttt{v} - \texttt{q} \times \texttt{vv})$, and use this variable to compute the (square of the) norm. In one case, we change the value of $v_{2,y}$, in another one, we keep the value set before the comparaison.

2.3 Finding a vector in a lattice

When v_1 and v_2 are set, we have to compute v in the lattice $\langle v_1, v_2 \rangle$, close to $(k, 0)$. First, we write the vector $(k, 0)$ in the basis $\{v_1, v_2\}$, i.e we find β_1, β_2 such that

$$\beta_1 v_1 + \beta_2 v_2 = (k, 0)$$

That means that (β_1, β_2) satisfies

$$\underbrace{\begin{pmatrix} v_{1x} & v_{2x} \\ v_{1y} & v_{2y} \end{pmatrix}}_{M} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} k \\ 0 \end{pmatrix}$$

As it is prooved before, v_1 and v_2 are linearly independant so $\det(M) \neq 0$ and we can compute his inverse :

$$M^{-1} = \frac{1}{\det(M)} \begin{pmatrix} v_{2y} & -v_{2x} \\ -v_{1y} & v_{1x} \end{pmatrix}$$

and we find (β_1, β_2) :

$$\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \frac{1}{\det(M)} \begin{pmatrix} v_{2y} & -v_{2x} \\ -v_{1y} & v_{1x} \end{pmatrix} \begin{pmatrix} k \\ 0 \end{pmatrix} = \frac{1}{v_{1x}v_{2y} - v_{1y}v_{2x}} \begin{pmatrix} kv_{2y} \\ -kv_{1y} \end{pmatrix}$$

Then, we have to find b_1, b_2 the nearest integers to β_1, β_2.
If $r = \frac{p}{q}$ is a rational number, the nearest integer to r is $\lfloor \frac{2n+d}{2d} \rfloor$.
Indeed, if $\lfloor r \rfloor \leq r < \lfloor r \rfloor + \frac{1}{2}$, the nearest integer is $\lfloor r \rfloor$, and

$$\left\lfloor \frac{2n+d}{2d} \right\rfloor = \left\lfloor \frac{n}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor = \lfloor r \rfloor$$

If $\lfloor r \rfloor + \frac{1}{2} \leq r < \lfloor r \rfloor + 1$, the nearest integer is $\lfloor r + 1 \rfloor$, and

$$\left\lfloor \frac{2n+d}{2d} \right\rfloor = \left\lfloor \frac{n}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{n}{d} + 1 \right\rfloor = \lfloor r + 1 \rfloor$$

This explains how we find b_1, b_2 using `mpq_numref` and `mpq_denref`.
Finally, computing u is just the substraction $(k, 0) - v$.

2.4 Using a window

We are going to use an algorithm with a window in order to compute faster kP. In this section, we explain why a window can save calculations, in the simple `double-and-add` algorithm.
First, we recall the "left to right" algorithm :
`Set` $R = P$`.`
`For` b `bit of` k `from left to right (begin at second bit) :`
· $R \leftarrow 2R$
· `If` $b = 1$ `:`
· $R \leftarrow R + P$
At the end of the algorithm, $R = kP$.

Supposing that the binary decomposition of k has approximately as many 1 as 0, the algorithm costs n doubles, and $\simeq n/2$ adds. Using a window is a way to decrease the number of adds (and leave the number of doubles).

We explain this fact with an example with a window of size 2.
We browse the bits two by two.

We use the following notation : $k = (a_{n-1}, \ldots, a_1, a_0)_2 = (b_{\lfloor (n-1)/2 \rfloor}, \ldots, b_0)_w$, where a_i is the i^{th} bit of a, and $b_i = (a_{2i+1}, a_{2i})_2 = 2a_{2i+1} + a_{2i}$.
We do a double-and-add browsing i from $\lfloor (n-1)/2 \rfloor$ to 0 :

Double-and-add.
$R \leftarrow 2R$ (one double)
$R \leftarrow R + b_{i,0}P$
– $R + P$ with probability 1/2
– $R + \mathcal{O}$ with probability 1/2
$R \leftarrow 2R$ (one double)
$R \leftarrow R + b_{i,0}P$
– $R + P$ with probability 1/2
– $R + \mathcal{O}$ with probability 1/2
Total. 2 **doubles** and $\frac{1}{2} + \frac{1}{2} = 1$ **add**.

Double-and-add with window.
($\mathcal{O}, P, 2P, 3P$ *are precomputed*)
$R \leftarrow 2^2 R$ (two doubles)
$R \leftarrow R + 2b_{i,1}P + b_{i,0}P$
$2b_{i,1}P + b_{i,0}P \in \{\mathcal{O}, P, 2P, 3P\}$ is precomputed
so it is only one add, and it is :
– $R + \mathcal{O}$ with probability 1/4
– $R + P$ with probability 2/4
– $R + 2P$ with probability 1/4
Total. 2 **doubles** and $\frac{1}{4} \cdot 0 + \frac{2}{4} \cdot 1 + \frac{1}{4} \cdot 1 = \frac{3}{4}$ **add**.

We conclude that using a window, it decreases the number of adds ($\frac{3}{4} < 1$), and keep the same number of doubles (the same count can be done with a bigger window).

2.5 Algorithm of simultaneous multiplication

This algorithm computes $uP + vQ$ using a window w for the reason explained in the previous subsection. We first find a signed binary decomposition of k that has more zeros.

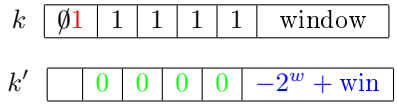
2.5.1 Using a signed binary representation

We implement the algorithm explained in [1]. This algorithm computes (with a window w) a binary decomposition of an integer k with some 1 and 0, but also -1. The goal is to get more 0, in order to speed up algorithm during the double–and–add. We browse the bits of k from right to left (stoping when we are close to the end).

1. If $b = 0$, we copy it in our decomposition, and we look at the following bit.
2. Else, we compute the block corresponding to the window. Then, there is two cases.
   (a) The first bit of the following window is a 0. Then, we copy the same bits of the window loading.
   (b) The first bit of the following window is a 1. We find the first 0 after this 1. We change the bits 1 browsed just before by some 0. We modify the bit 0 found in a 1 in k (not in the resulting chain !). To compensate, we modify the window loading :

$$\text{win} = 2^w - \text{win}$$

and then fill in the resulting chain with the opposite of the bits of win.



Indeed, if t is the rank of the first 0 found, we change this 0 in 1 in k (in red) so $k \leftarrow k + 2^w$. The compensation is done by the zeros replacing the ones (in green) :

$$-2^{i+w} - \ldots - 2^{t-1} = -\sum_{m=0}^{t-1} 2^m + \sum_{m=0}^{i+w-1} 2^m = -\frac{2^t - 1}{2-1} + \frac{2^{i+w} - 1}{2-1} = 2^{i+w} - 2^t$$

and by the modification of the window (in blue) : a term 2^{i+w} compensates.

To store the new decomposition, we use the correspondance :

$$1 \leftrightarrow 10 \qquad 0 \leftrightarrow 00 \qquad -1 \leftrightarrow 11$$

Our signed binary representation uses twice the number of bits, but we get more zeros for the double–and–add.

2.5.2 Implementation of $uP + vQ$ using the signed binary representation

A simple algorithm is to precompute all $iP + jQ$ for $i, j \in [\![0, 2^w - 1]\!]$, and then set $R \leftarrow \mathcal{O}$ and compute

$$R \leftarrow 2^w R + (u^i P + v^j Q)$$

As $u^i P + v^j Q$ is precomputed, this step in less expensive than computing uP and vQ separately by Double–and–add.

With our signed decomposition, the precomputed part can be $\pm iP \pm jQ$, so we store four precomputed tables (where $i, j \in \{0, \ldots, 2^w - 1\}$)

$$\{iP + jQ\} \qquad \{-iP - jQ\} \qquad \{-iP + jQ\} \qquad \{iP - jQ\}$$

Note that if w is big, the precomputation can be cumbersome. In reality, we use a small window. This algorithm is suitable for a signature algorithm (ECDSA for example), where we can precompute the tables upstream because the point P is known.

2.6 Test in an example of curve

We want to apply our method in the following curve, included in the WAP specification of the WTLS protocol :

$$E : y^2 = x^3 + 3$$

over the prime field $\mathbb{F}_p$ where

$$p = 1461501637330902918203684832716283019655932313743$$

is a 160-bit prime, and

$$\#E(\mathbb{F}_p) = 1461501637330902918203687013445034429194588307251$$

We are in the context of the example 2 with $b = 3$. We want to find $\beta \in \mathbb{F}_p$ of order 3 to construct the endomorphism. A such β is a zero of the polynomial

$$X^3 - 1 = (X - 1)(X^2 + X + 1)$$

and so, because $\beta \neq 1$, β is a zero of $X^2 + X + 1$.
We calculate the discriminant $\Delta = 1 - 4 = -3$. The Legendre symbol shows that -3 is a square mod p :

$$\left(\frac{-3}{p}\right) = -\left(\frac{3}{p}\right) = -\left(\frac{p}{3}\right)(-1)^{\frac{3-1}{2}\frac{p-1}{2}} = -\left(\frac{1}{3}\right)(-1)^{\frac{p-1}{2}} = 1$$

because $\frac{p-1}{2}$ is odd.
-3 is a square mod p, and we find $\beta = \frac{-1\pm\sqrt{-3}}{2}$. The problem is now to find explicitly $\sqrt{-3}$. Since $p \equiv 3 \mod 4$, a possible root is $x = (-3)^{\frac{p+1}{4}}$. Indeed,

$$x^2 = (-3)^{\frac{p+1}{2}} = (-3)^{\frac{p-1}{2}} \times (-3) = \underbrace{(\sqrt{-3})^{p-1}}_{=1} \times (-3) = -3$$

($p-1$ is the order of the multiplicative group $\mathbb{F}_p^*$)
Finally, the two elements of order 3 are

$$\beta = \frac{-1 \pm (-3)^{\frac{p+1}{4}}}{2}$$

Then, we can compute φ with β, and we prooved that $\varphi = [\lambda]$, where $\lambda^2 + \lambda + 1 \equiv 0 \mod n$! The same argument gives us (working mod n) :

$$\lambda = \frac{-1 \pm (-3)^{\frac{n+1}{4}}}{2}$$

3 Experimental results

We compare our results in the GLV method with the Double–and–add first algorithm.
All these experimental results are done in the curve $E : y^2 = x^3 + 3$ over the prime field $\mathbb{F}_p$ where

$$p = 146150163733090291820368483271628301965593231374 3$$

As

$$\#E(\mathbb{F}_p) = 146150163733090291820368701344503442919458830725 1$$

is prime, we chose $P = [1 : 2 : 1]$ ($\text{ord}(P) = n$ because $P \neq \mathcal{O}$).
Then, we compute kP for random values of k. In order to analyse the differences, we compute kP a thousand times for each algorithm. We also change the window to compare the results.

Here is the results where the times are average for five random k in seconds (with 1000 loops).

Window	**Double–and–add**	**GLV**
2	0.600952	0.329202
3	0.597540	0.292368
4	0.603717	0.270426
5	0.593262	0.253423
6	0.591852	0.243728
7	0.591472	0.240665

Average times in seconds for 1000 loops algorithms of kP
Comparaisons between Double–and–add and GLV ***with*** *precomputation*

We can conclude that in average, the GLV method with precomputation is 50% better than the double–and–add. Moreover, using a bigger window makes the computation faster. The problem is that when the window is greater than 7, the precomputation is too important.

The tests with a precomputation during GLV (and not before) give obviously less good results :

Window	**Double–and–add**	**GLV**
2	0.604084	0.452246
3	0.601965	0.771484
4	0.593465	2.162002

Average times in seconds for 1000 loops algorithms of kP
Comparaisons between Double–and–add and GLV ***without*** *precomputation*

Our implementation of GLV can be suitable for a ECDSA algorithm : the point P is known, and so we can do precomputation and sign 50% faster than a square–and–multiply. Without precomputation, we obtain a gain for a small window ($w = 2$) only.

A last analysis is done using the profiler to distinguish which functions are cumbersome :

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
52.02	0.13	0.13	3108932	0.04	0.04	add_points
24.01	0.19	0.06	30000	2.00	7.94	multiple_multiplication_without_precompute
20.01	0.24	0.05	60000	0.83	0.83	signedBinary
4.00	0.25	0.01	30000	0.33	0.33	find_v1_v2
0.00	0.25	0.00	3344202	0.00	0.00	point_copy
0.00	0.25	0.00	3226299	0.00	0.00	point_clear
0.00	0.25	0.00	3226299	0.00	0.00	point_init
0.00	0.25	0.00	640000	0.00	0.16	double_and_add
0.00	0.25	0.00	43932	0.00	0.00	point_neg
0.00	0.25	0.00	30000	0.00	0.33	decompose
0.00	0.25	0.00	30000	0.00	0.00	find_u
0.00	0.25	0.00	30000	0.00	0.00	find_v
0.00	0.25	0.00	30	0.00	0.00	convert_jac

*GLV profile **with** precomputing*

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
46.99	0.31	0.31	5129000	0.06	0.07	add_points
36.38	0.55	0.24	15000	16.01	43.68	multiple_multiplication2
6.06	0.59	0.04	30000	1.33	1.33	signedBinary
4.55	0.62	0.03	11939007	0.00	0.00	point_clear
3.03	0.64	0.02	11939007	0.00	0.00	point_init
1.52	0.65	0.01	432000	0.02	0.20	double_and_add
0.76	0.66	0.01	5489000	0.00	0.00	point_copy
0.76	0.66	0.01				point_printf
0.00	0.66	0.00	3500000	0.00	0.00	point_neg
0.00	0.66	0.00	15000	0.00	0.00	decompose
0.00	0.66	0.00	15000	0.00	0.00	find_u
0.00	0.66	0.00	15000	0.00	0.00	find_v
0.00	0.66	0.00	15000	0.00	0.00	find_v1_v2

*GLV profile **without** precomputing*

In both algorithms, the `add_points` function is very cumbersome. Also, this function is called intensely. Using suitable points addition is an important way to speed up the exponentiation.

References

[1] Takatoshi Ono Atsuko Miyaji and Henri Cohen. Efficient elliptic curve exponentiation.

[2] Torbjörn Granlund and the GMP development team. Gnu mp, the gnu multiple precision arithmetic library.

[3] Robert J. Lambert Robert P. Gallant and Scott A. Vanston. Faster point multiplication on elliptic curves with efficient endomorphisms.