# Adaptive Precision Floating Point LLL*

Thomas Plantard, Willy Susilo, and Zhenfei Zhang

Centre for Computer and Information Security Research
School of Computer Science & Software Engineering (SCSSE)
University Of Wollongong, Australia
{thomaspl, wsusilo, zz920}@uow.edu.au

**Abstract.** The LLL algorithm is one of the most studied lattice basis reduction algorithms in the literature. Among all of its variants, the floating point version, also known as $\mathrm{L}^2$, is the most popular one, due to its efficiency and its practicality. In its classic setting, the floating point precision is a fixed value, determined by the dimension of the input basis at the initiation of the algorithm. We observe that a fixed precision overkills the problem, since one does not require a huge precision to handle the process at the beginning of the reduction. In this paper, we propose an adaptive way to handle the precision, where the precision is adaptive during the procedure. Although this optimization does not change the worst-case complexity, it reduces the average-case complexity by a constant factor. In practice, we observe an average 20% acceleration in our implementation.

*Keywords: Lattice Theory, Lattice Reduction, LLL, floating point arithmetic*

## 1 Introduction

A lattice $\mathcal{L}$ is a discrete subgroup of $\mathbb{R}^n$. It is usually represented by a set of integer linear combinations of vectors $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$, $\mathbf{b}_i \in \mathbb{R}^n$, $d \leq n$. $\mathbf{B}$ is a basis of $\mathcal{L}$, if $\mathbf{b}_i$-s are linearly independent, and $d$ is known as the dimension of the $\mathcal{L}$. For a given lattice $\mathcal{L}$, there exists an infinite number of bases for $d \geq 2$. Given a "bad" basis (i.e., a basis with large coefficients), to find a short vector of the lattice (a vector with small coefficients), or a "good" basis (i.e., a basis with small coefficients and the vectors are almost orthogonal), is known as the *lattice reduction*.

The LLL algorithm, named after its inventors, Lenstra, Lenstra and Lovász [11], is a polynomial time lattice reduction algorithm. For a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$, the LLL algorithm is proven to terminate within $O(d^6\beta^3)$ time, where $\beta$ is the maximum bit length of all the Euclidean norm of input vectors. This algorithm is of great importance in cryptanalysis, since finding vectors with exponential approximation (in $d$) to the shortest non-zero vector will break some of the cryptosystems, such as the Coppersmith's method [6, 5] against RSA cryptosystem

* This work is supported by ARC Future Fellowship FT0991397.

[21]. For this reason, one of the working direction is to increase the time efficiency of the algorithm.

One of the greatest achievements towards this end was due to [16, 18], which incorporates the floating point arithmetics. It is the first algorithm that achieves quadratic complexity in terms of $\beta$, hence it was named $\mathrm{L}^2$. In practice, there exist two main versions, a standard version (referred to as "L2") that delivers the proven complexity and a heuristic version (referred to as "FP") that adopts some heuristics to boost the reduction.

$\mathrm{L}^2$ uses floating points instead of integers, where the precision is set to $O(d)$ to deliver error free arithmetics. The precision was determined at the beginning of the procedure. However, we notice that this setting indeed is an overkill. LLL deals with vectors progressively. During the process, when $k < d$ vectors are involved, it only requires $O(k)$ bit precision to deliver correct reduction. In fact, the only time that the algorithm requires $O(d)$ precision is when all the vectors are involved. This inspires us to propose the adaptive precision floating point LLL algorithm.

*Our Contribution.* We present an adaptive precision floating point LLL algorithm, the *ap-fplll*. We consider both the proven version, L2, and the most efficient version, FP. We test our *ap-fplll* with random lattices. In practice, we are always faster than the standard version of $\mathrm{L}^2$. When the dimension and/or determinant are sufficiently large, we are also faster than the fastest implementation of $\mathrm{L}^2$. In general, we accelerate the reduction by 20%.

*Roadmap.* In the next section, we will discuss the background knowledge and terminology required throughout the paper and briefly recall the $\mathrm{L}^2$ algorithm. In Section 3, we show our adaptive precision floating point LLL algorithm and analyze its performance. In Section 4, we show the implementation result of our proposed algorithm. Finally, Section 5 concludes the paper.

## 2 Background

It this section, we briefly review the related area. We refer readers to [12, 14] for a more complex account of lattice theory, and [4, 19] for the LLL algorithm.

### 2.1 Lattice Basics

The lattice theory, also known as the geometry of numbers, was introduced by Minkowski in 1896 [15].

**Definition 1 (Lattice).** *A lattice $\mathcal{L}$ is a discrete sub-group of $\mathbb{R}^n$, or equivalently the set of all the integral combinations of $d \leq n$ linearly independent vectors over $\mathbb{R}$.*

$$\mathcal{L} = \mathbb{Z}\mathbf{b}_1 + \mathbb{Z}\mathbf{b}_2 + \cdots + \mathbb{Z}\mathbf{b}_d, \mathbf{b}_i \in \mathbb{R}^n$$

$\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ *is called a basis of $\mathcal{L}$ and $d$ is the dimension of $\mathcal{L}$, denoted as* $\dim(\mathcal{L})$. *$\mathcal{L}$ is a full rank lattice if $d$ equals to $n$.*

**Definition 2 (Successive Minima).** *Let $\mathcal{L}$ be an integer lattice of dimension $d$. Let $i$ be a positive integer. The $i$-th successive minima with respect to $\mathcal{L}$, denoted by $\lambda_i$, is the smallest real number, such that there exist $i$ non-zero linearly independent vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_i \in \mathcal{L}$ with*

$$\|\mathbf{v}_1\|, \|\mathbf{v}_2\|, \ldots, \|\mathbf{v}_i\| \leq \lambda_i.$$

The norm of $i$-th minima of a random lattice is estimated by

$$\lambda_i(\mathcal{L}) \sim \sqrt{\frac{d}{2\pi e}} \det(\mathcal{L})^{\frac{1}{d}}. \tag{1}$$

**Definition 3 (Gram-Schmidt Orthogonalization).** *Let $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a basis of $\mathcal{L}$. The Gram-Schmidt Orthogonalization (GSO) of $\mathbf{B}$ is the following basis $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_d^*)$*

$$\begin{aligned}
\mathbf{b}_1^* &= \mathbf{b}_1, \\
\mathbf{b}_i^* &= \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*, && (2 \leq i \leq d), \\
\mu_{i,j} &= \frac{\mathbf{b}_i \cdot \mathbf{b}_j^*}{\mathbf{b}_j^* \cdot \mathbf{b}_j^*}.
\end{aligned}$$

**Definition 4 (Gram determinants).** *Let $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a basis of $\mathcal{L}$. Let $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_d^*)$ be the corresponding GSO. The Gram determinants of $\mathbf{B}$, noted $\{\Delta_1^*, \ldots, \Delta_d^*\}$ is defined as*

$$\Delta_i^* = \det(\mathbf{b}_1^*, \ldots, \mathbf{b}_i^*).$$

The loop invariant is defined as the product of all Gram determinants as: $D = \prod_{i=1}^{d-1} \Delta_i^*$. For any basis, $D$ is upper-bounded by $2^{\beta d(d-1)}$ [4].

## 2.2 The LLL algorithm

The LLL algorithm was proposed by Lenstra, Lenstra and Lovasz [11] in 1982. We briefly sketch the algorithm as in Algorithm 1 and 2. LLL produces a $(\delta, 0.5)$-reduced basis for a given basis (see definitions below).

**Definition 5 ($\eta$-size reduced[18]).** *Let $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a basis of $\mathcal{L}$. $\mathbf{B}$ is $\eta$-size reduced, if $|\mu_{i,j}| \leq \eta$ for $1 \leq j < i \leq d$. $\eta \geq 0.5$ is the reduction parameter.*

**Definition 6 ($(\delta, \eta)$-reduced basis[18]).** *Let $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a basis of $\mathcal{L}$. $\mathbf{B}$ is $(\delta, \eta)$-reduced, if the basis is $\eta$-size reduced and it satisfies Lovász condition as follows: $\delta \|\mathbf{b}_{i-1}^*\|^2 \leq \|\mathbf{b}_i^* + \mu_{i,i-1}^2 \mathbf{b}_{i-1}^*\|^2$ for $2 \leq i \leq d$. $\frac{1}{4} < \delta \leq 1$ and $\frac{1}{2} \leq \eta < \sqrt{\delta}$ are two reduction parameters.*

For a lattice $\mathcal{L}$ with dimension $d$, and a basis $\mathbf{B}$, where the Euclidean norm of all spanning vectors in $\mathbf{B}$ is $\leq 2^\beta$, the worst-case time complexity is polynomial $O(d^6\beta^3)$.

This complexity comes from the following. Firstly, there exists $O(d^2\beta)$ loop iterations. It has been shown that the loop invariant $D$ is not changed except during the swap procedure, while during the swap, $D$ is decreased by a factor of $\delta$. Hence, the total number of swaps is upper bounded by the absolute value of $\frac{\beta d(d-1)}{\log_2 \delta}$. Therefore there are maximum $O(d^2\beta)$ loop iterations. As a result, the total number of size reduction is $O(d^2\beta)$. Secondly, for each size reduction, there are $O(d^2)$ arithmetic operations. Finally, each operation involves integer multiplications with a cost of $\mathcal{M}(d\beta)$ due to rational arithmetics.[1] Hence, the original LLL algorithm terminates in polynomial time $O(d^6\beta^3)$.

**Algorithm 1** Size Reduction

```
Input: $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_d)$, its GSO and an index $k$.
Output: A new basis $\mathbf{B}$, where $\mathbf{b}_k$ is size reduced, and the updated GSO.
  for $i = (k-1) \to 1$ do
    $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lceil \mu_{k,i} \rfloor \cdot \mathbf{b}_i$.
    Update GSO
  end for
  return $\mathbf{B}$.
```

Note that for the bases of a random lattices that are using in our analysis and implementation, the number of loop iterations is $O(d\beta)$ instead of $O(d^2\beta)$ (see Remark 3, [16]). So the complexity will be reduced by $O(d)$.

## 2.3 Floating point LLL algorithm

The most costly part in an LLL procedure is the size reduction. When one performs a size reduction, the GSO needs to be regularly updated. During the update, the classic LLL needs to operate on integers with length of $O(d\beta)$. As a result, the multiplication of those integers incurs a cost of $O(\mathcal{M}(d\beta))$.

**The first floating point LLL algorithm.** In [22] and [23], Schnorr showed that using floating points instead of integers for LLL can reduce the cost of multiplications from $\mathcal{M}(d\beta)$ to $\mathcal{M}(d+\beta)$. To the best of our knowledge, this is the first time where floating points make a significant difference in the LLL algorithm. However, it is obseved that the hidden constant in the bit complexity remains huge.

[1] $\mathcal{M}(d)$ represents the cost of multiplication between two $O(d)$ length integers. In this paper, we follow the LLL algorithm by using $\mathcal{M}(d)$ to be $O(d^2)$ assuming a naive integer multiplication. One can replace it with $O(d^{1+\varepsilon})$ to obtain the exact bit complexity in practice.

**Algorithm 2** LLL

```
Input: $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_d)$ and a reduction parameter $\eta$
Output: A $(\delta, 0.5)$-reduced basis $\mathbf{B}$.
  Compute GSO.
  $k \leftarrow 2$.
  while $k \leq d$ do
    size reduce $(\mathbf{B}, k)$;
    if $\delta\|\mathbf{b}^*_{k-1}\|^2 \leq \|\mathbf{b}_k\|^2 + \mu^2_{k,k-1}\|\mathbf{b}^*_{k-1}\|^2$ (Lovász condition) then
      $k \leftarrow k + 1$
    else
      swap $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$;
      $k \leftarrow \max(k - 1, 2)$;
      Update GSO;
    end if
  end while
  return $\mathbf{B}$.
```

**The $\mathbf{L^2}$ algorithm.** To further improve the efficiency, the $\mathrm{L}^2$ algorithm was proposed by Nguyen and Stehlé [16] in 2005. It is the first variant whose worst-case time complexity is quadratic with respect to $\beta$. It uses a worst-case time complexity of $O(d^5\beta^2 + d^6\beta)$ to produce a $(\delta, \eta)$-reduced basis for $\frac{1}{4} < \delta \leq 1$ and $\frac{1}{2} < \eta < \sqrt{\delta}$.

The $\mathrm{L}^2$ algorithm incorporates the *lazy reduction* as follows: firstly, the size reduction consists of many floating point reductions (*fp*-reduction). Then, within each *fp*-reduction, one works on floating point whose precision is $O(d)$. The factor within $O(\cdot)$ is influenced by the reduction parameters. A default setting in the *fplll* is approximately $1.6d$. As a consequence, the multiplication cost is reduced to $O(\mathcal{M}(d))$. However, the trade-off is that, each *fp*-reduction may be incomplete, and one is required to perform $O(1 + \frac{\beta}{d})$ *fp*-reductions to ensure that the vectors is size reduced.

**$\mathbf{L^2}$ in practice.** In practice, the *fplll* library [20] and MAGMA [3] are two well known implementations. Within both implementations, there exist two main versions, "L2" and "FP" (it is known as "LM_WRAPPER" in the *fplll*). The L2 is described as above. It is the proved version of $\mathrm{L}^2$. Meanwhile, in practice, one can further improve the average performance with some heuristics. To the best of our knowledge, the most efficient implementation of $\mathrm{L}^2$ is FP. As far as the floating point is concerned, FP is L2 plus some early reductions.

In FP, the basis is early reduced as follows: the algorithm will choose several fixed precisions subject to the following conditions:

- The arithmetics are fast with those precisions, for instance, 53 for C double precision.
- Reductions with those precisions are likely to produce a correct basis, for instance, $d$ rather than $1.6d$ (see Remark 4, [17]).

Reductions with above precisions are cheaper, while they produce somewhat reduced bases. So the algorithm will try all early reductions with different fixed precisions, and finally perform an L2 to ensure the quality of reduction. We note that those early reductions do not change the overall complexity, since in theory the last L2 is still the most costly one. Nevertheless, in practice, the early reductions are very effective to accelerate the whole procedure.

## 3 Our adaptive precision floating point LLL algorithm

### 3.1 The algorithm

The LLL algorithm uses a stepping method. For a basis $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_d)$, it starts with the first 2 vectors, and then adds 1 vector into the procedure during each step. We notice that, one does not require a floating point precision of $O(d)$ to reduce in the first $d-1$ steps. In fact, for any $k$ vectors, one only requires $O(k)$ precisions. Hence, a possible improvement is to adaptively select the precision according to the number of vectors that are involved. This leads to the adaptive precision floating point LLL algorithm (*ap-fplll*) as shown in Algorithm 3.

**Algorithm 3** Adaptive precision floating point LLL algorithm

```
Input: $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_d)$, reduction parameters $(\delta, \eta)$ and a starting index $\gamma$.
Output: An $(\delta, \eta)$-reduced basis $\mathbf{B}$.
  $k \leftarrow 2$, $k_{max} \leftarrow \gamma$.
  SetPrecision($\gamma$) and Compute GSO accordingly.
  while $k \leq d$ do
    Size reduce $(\mathbf{B}, k)$;
    if $\delta\|\mathbf{b}^*_{k-1}\|^2 \leq \|\mathbf{b}_k + \mu_{k,k-1}\|\mathbf{b}^*_{k-1}\|^2$ (Lovász condition) then
      $k \leftarrow k + 1$;
      if $k > k_{max}$ then
        $k_{max} \leftarrow k$;
        if $k_{max} > \gamma$ then
          SetPrecision($k_{max}$) and Compute GSO accordingly.
        end if
      end if
    else
      Swap $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$;
      $k \leftarrow \max(k-1, 2)$;
      Update GSO;
    end if
  end while
  return $\mathbf{B}$.
```

Algorithm 3 describes the L2 version of our *ap-fplll* algorithm. $k$ indicates the current vector the algorithm is working on, while $k_{max}$ indicates the maximum number of the vectors that are involved. When $k_{max}$ changes, one is required to reset the precision. To obtain the FP version of the algorithm, one conducts early reductions as in *fplll* when $k_{max}$ increases.

We also introduce an index parameter $\gamma$ due to implementation issue. For some of the library, there exists a minimum precision for floating point. If the required precision is smaller than this bound, the algorithm will automatically use the bound. In this case, the precision is not $O(d)$, rather it is a fixed value subject to the system. Hence, using an adaptive precision will not reduce the cost of multiplication, rather it will repetitively recompute the GSO. We set $\gamma$ such that when more than $\gamma$ vectors are involved, the algorithm will need to use a precision subject to the dimension.

*Remark 1.* In our algorithm, we follow the L2 by setting the precision to be the exact value that is required, i.e., $1.6d$, to deliver a fair comparison. Nevertheless, it is worth pointing out that the *mpfr* library [1] that the *fplll* depends on operates a floating number as a linked list of blocks of 32 bits (or 64 bits), therefore, it is possible that increasing precisions with respect to the size of the block (the actual size may be smaller than 32 or 64 due to the overhead of storing a floating number) may derive a better performance, since in this case, the GSO will be updated less often.

## 3.2 Worst-case complexity

In this part, we prove that our algorithm uses the same worst-case complexity with $\mathrm{L}^2$.

The reduction part of $\mathrm{L}^2$ algorithm can be seen as our algorithm with a fixed precision of $O(d)$. Therefore, during the reduction part we can never be more costly than $\mathrm{L}^2$. However, our algorithm needs to recompute the GSO for each step, where the GSO is updated partially in $\mathrm{L}^2$. On the worst-case, we can be more costly than $\mathrm{L}^2$ by the cost of computing the GSO.

For each step, the cost of computing GSO is $O(d^2k^2\beta)$. This brings an overall cost of $O(d^5\beta)$, hence it will not affect the worst-case complexity of $O(d^6\beta + d^5\beta^2)$. As a result, the *ap-fplll* uses a same worst-case complexity with $\mathrm{L}^2$.

## 3.3 Average behaviors

We construct the random lattices as in [9]. There exist bases of those lattices that are of the following form:

$$\mathbf{B} = \begin{pmatrix} X_1 & 0 & 0 & \dots & 0 \\ X_2 & 1 & 0 & \dots & 0 \\ X_3 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ X_d & 0 & 0 & \dots & 1 \end{pmatrix},$$

where $X_1$ is a large prime with $\beta$ bits. $X_i$-s ($i \neq 1$) are chosen uniformly between 0 and $p$.

These bases are somewhat standard to analyze lattice reductions. They are believed to leak the least information for the corresponding lattice, since it can be obtained by any lattice basis within polynomial time. They are adopted in [8, 17] where the LLL behavior is widely analyzed. Further, when setting $\beta \sim 10d$, these bases are also used for the shortest vector problem (SVP) challanges in [2].

We analyze the average case complexity of our algorithm with the above bases. Since the lattice is a random one, then its minimas $\lambda_i$ follow Equation 1.

$$\mathbf{B}_k = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,k} & 0 & \dots & 0 \\ x_{2,1} & x_{2,2} & \dots & x_{2,k} & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ x_{k,1} & x_{k,2} & \dots & x_{k,k} & 0 & \dots & 0 \\ X_{k+1} & 0 & \dots & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ X_d & 0 & \dots & 0 & 0 & \dots & 1 \end{pmatrix}$$

For the $k$-th step ($k > 2$), the basis is shown as above, where $\|\mathbf{b}_i\| \lesssim 2^{\frac{k-1}{2}} 2^{\frac{\beta}{k-1}}$ for $i < k$ and $\|\mathbf{b}_k\| \lesssim 2^{\frac{k-2}{2}} 2^{\frac{\beta}{k-2}}$. Hence, the loop invariant for the current step $D_k$ is then bounded by

$$\prod_{i=1}^{k} \|\mathbf{b}_i\|^{2(k-i+1)} = 2^{k(k-1)^2-1} 2^{2\beta k + \frac{\beta}{(k-1)(k-2)}}.$$

When the $k$-th step terminates, $\mathbf{b}_i$ will be reduced to $2^{\frac{k}{2}} 2^{\frac{\beta}{k}}$ for $i \leq k$. Hence, one obtains $O(\beta)$ loop iterations on average cases. We note that this observation is quite natural, since there are $O(d\beta)$ loop iterations in total, hence, on average there are $O(\beta)$ loop iterations for each $k$.

Let $l$ be the precision to be used in the algorithms. Then for each loop iteration, one needs to perform $O(1+\frac{\beta}{l})$ floating point reductions, each at a cost of $O(d^2\mathcal{M}(l))$. Since $l = O(k)$, so it will cost $O(d^2\beta \sum_{i=\gamma}^{d}(1+\frac{\beta}{i})\mathcal{M}(i)))$ that is $\frac{1}{6}c_1 d^5\beta + \frac{1}{2}c_2 d^4\beta^2$ for some constants $c_1$ and $c_2$, if we assume $\mathcal{M}(d) = O(d^2)$.

For comparison, we also show the complexity of L$^2$: $O(d^2\beta \sum_{i=1}^{d}(1+\frac{\beta}{d})\mathcal{M}(d))$ which is $c_1 d^5\beta + c_2 d^4\beta^2$ for the same constants.

It is straightforward to see that our algorithm uses the same bit complexity with L$^2$. Further, our algorithm wins on both terms. However, the factor $\frac{1}{6}$ on the first term does not make a difference, which is due to the following. Firstly, in this case, $\beta < d$ which indicates that for each lazy reduction, only requires $O(1)$ *fp*-reductions, while our advantage is in fact a faster *fp*-reduction. Hence, our advantage diminishes. Secondly, the cost of recompute the GSO is also $O(d^5\beta)$ on worst cases as well.

Nevertheless, when $\beta > d$, we anticipate a lot of reductions. In this case, we should be able to accelerate the reduction by a factor between 0 and 50% for L2 (due to the fact that in practice $\mathcal{M}(d) \leq O(d^{1+\varepsilon})$).

As for FP, in practice, it is possible that the early reduction already produces a good basis. It happens a lot when the dimension is small and $\frac{\beta}{d}$ is small. In this case, the adaptive precision will not boost the reduction, since our advantage works on the final procedure, while in the final procedure, the basis is already in a good shape. Nevertheless, when one increases the dimension and/or the $\frac{\beta}{d}$, the adaptive precision will still accelerate the reduction.

## 4 Implementation

In this section we show the implementation results of our algorithm. The tests were conducted with *fplll* library version 4.0 on Xeon E5640 CPUs @ 2.66GHz. The memory was always sufficient since the algorithm only requires a polynomial space. We used the random lattice basis as shown in the last section. We set the dimension to 64 and increase it by 32 each time. For each dimension, we set $\beta = 10d$, $20d$, $\cdots$, and generated the bases accordingly. For each dimension/determinant, we tested 10 different bases where the random numbers are generated from different seeds $0 \sim 9$ using the pseudo-random generator of the NTL library [24].

We set the index $\gamma = 40$ so that the required precision is strictly greater than 53. In deed, one can change $\gamma$ to improve *ap-fplll*. However, to show a more fair comparison, we use a same value for all the tests. The reduction parameter pair is set to $(0.99, 0.51)$ as the default value of *fplll*. This results in a very strongly reduced basis which is in general most useful for cryptanalysis.

We show the implementation results as follows. As one can see from Table 1, one can merely observe any difference between two algorithms at the beginning of the tests, although *ap-fplll*-L2 is slightly faster than *fplll*-L2. We believe the reason is that the cost of recompute the GSO is more or less the same of the advantage of using smaller precisions. However, when the dimension grows, the reductions influence the total complexity more importantly compared with the GSO computation, and as a result, the *ap-fplll* starts to be a lot faster. Figure 1 illustrated the winning percentage of *ap-fplll*-L2 versus *fplll*-L2. When $d = 64$, we accelerate the reduction by 10%, since it is closer to the starting index $\gamma = 40$. When $d \geq 96$, the influence of $\gamma$ diminishes, and we start to see the phenomenon where the dimension and/or determinant grow, the advantage increases as well. When the dimension and the determinant are sufficiently large, we can expect an advantage up to 40%. Overall, our algorithm is always faster in all cases, and we anticipate a boost of over 20% in general.

The results for the FP version are shown in Table 2. The results are not as stable as L2 due to the early reductions. As we anticipated, with small determinant/dimension, i.e., $\beta = 10d$, our algorithm does not accelerate the reduction. The early reduction technique works extremely efficient in those cases. Nevertheless, the disadvantage is still acceptable considering that even in dimension 448, the disadvantage is less than several of minutes.

Meanwhile, for the other cases, when the dimension grows, we start to observe advantages. Further, the advantage rises with the increase of dimension

| | | $\beta$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 10$d$ | | 20$d$ | | 30$d$ | | 40$d$ | |
| | | *ap-fplll*-L2 | *fplll*-L2 | *ap-fplll*-L2 | *fplll*-L2 | *ap-fplll*-L2 | *fplll*-L2 | *ap-fplll*-L2 | *fplll*-L2 |
| | 64 | 5.298 | 5.742 | 11.015 | 12.319 | 17.719 | 20.069 | 23.941 | 27.212 |
| | 96 | 29.488 | 30.607 | 64.643 | 69.336 | 104.553 | 113.513 | 144.9 | 158.222 |
| | 128 | 95.445 | 99.326 | 221.722 | 237.136 | 368.633 | 409.15 | 516.012 | 577.822 |
| | 160 | 234.241 | 253.711 | 575.6 | 636.703 | 971.459 | 1149.22 | 1417.34 | 1718.84 |
| | 192 | 470.986 | 522.537 | 1241.9 | 1416.41 | 2278.79 | 2876.82 | 3248.4 | 4184.2 |
| | 224 | 838.059 | 1003.08 | 2385.81 | 2944.07 | 4386.86 | 6062.06 | 6604.18 | 9238.31 |
| $d$ | 256 | 1349 | 1702.95 | 4221.74 | 5452.48 | 8235.8 | 11684.3 | 11942.6 | 17102.9 |
| | 288 | 2033.29 | 2561.16 | 6979.11 | 9080.97 | 13481.2 | 19231.7 | | |
| | 320 | 2772.15 | 3702.22 | 11007.3 | 15048.3 | | | | |
| | 352 | 3686.8 | 5181.06 | | | | | | |
| | 384 | 4772.02 | 7175.78 | | | | | | |
| | 416 | 6087.31 | 9476.52 | | | | | | |
| | 448 | 7641.25 | 12563.9 | | | | | | |

Table 1: Test Results: *ap-fplll*-L2 vs *fplll*-L2

| | | $\beta$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $10d$ | | $20d$ | | $30d$ | | $40d$ | | $50d$ | |
| | | *ap-fplll*-FP | *fplll*-FP | *ap-fplll*-FP | *fplll*-FP | *ap-fplll*-FP | *fplll*-FP | *ap-fplll*-FP | *fplll*-FP | *ap-fplll*-FP | *fplll*-FP |
| $d$ | 64 | 1.251 | 0.968 | 2.19 | 1.941 | 3.302 | 3.1 | 4.484 | 4.275 | 5.707 | 5.536 |
| | 96 | 6.195 | 4.833 | 12.206 | 11.488 | 19.283 | 18.919 | 26.769 | 27.424 | 34.075 | 35.028 |
| | 128 | 19.426 | 16.934 | 40.471 | 40.672 | 66.898 | 70.512 | 93.99 | 102.083 | 123.823 | 134.324 |
| | 160 | 48.422 | 42.133 | 109.613 | 113.167 | 185.293 | 201.169 | 268.095 | 298.629 | 357.412 | 418.395 |
| | 192 | 108.432 | 95.222 | 264.037 | 266.864 | 476.426 | 533.024 | 699.073 | 777.995 | 982.345 | 1076.97 |
| | 224 | 220.045 | 201.6 | 543.712 | 626.538 | 1160.49 | 1719.1 | 1922.35 | 2589.31 | 2537.65 | 3705.78 |
| | 256 | 564.462 | 605.925 | 1862.08 | 2829.1 | 3979.16 | 5243.41 | 6013.42 | 8036.89 | 8105.54 | 10185.1 |
| | 288 | 1127.6 | 1175.76 | 4557.49 | 5576.77 | 8451.64 | 10784.1 | 12722.3 | 16226 | 16787.2 | 21073.3 |
| | 320 | 1879.03 | 1868.19 | 7826.52 | 9341.18 | 15175.8 | 18896 | | | | |
| | 352 | 2800.87 | 2896.21 | 12445.3 | 14664.5 | | | | | | |
| | 384 | 4136.41 | 4123.88 | | | | | | | | |
| | 416 | 6223.48 | 6150.35 | | | | | | | | |
| | 448 | 8821.49 | 8641.27 | | | | | | | | |

Table 2: Test Results: *ap-fplll*-FP vs *fplll*-FP

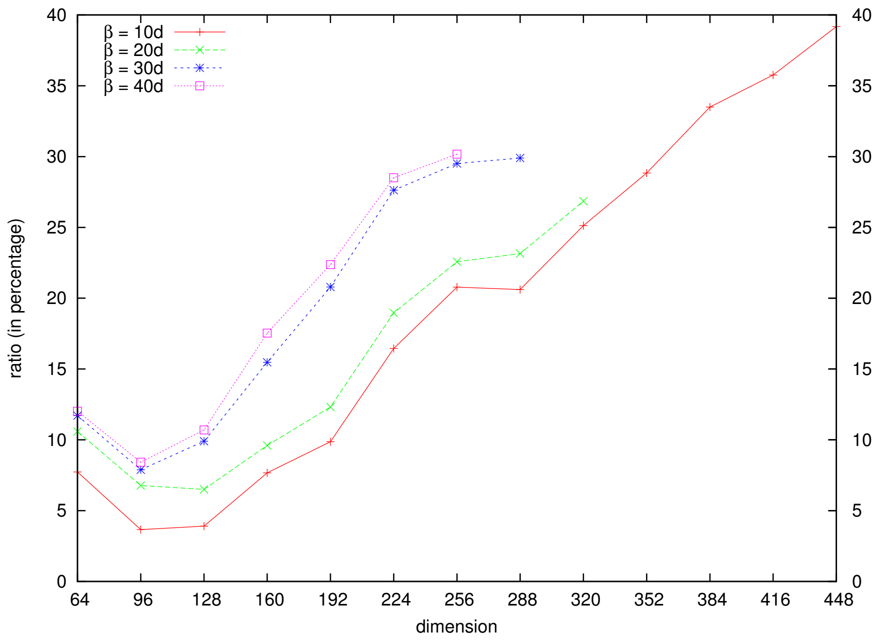

Fig. 1: Test Results: winning percentage of *ap-fplll* vs *fplll* using L2

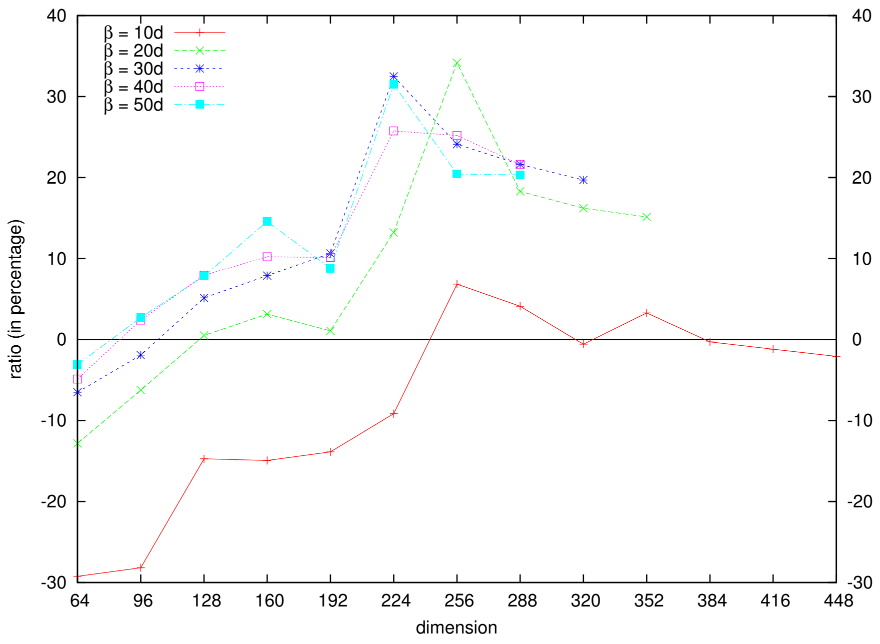

Fig. 2: Test Results: winning percentage of *ap-fplll* vs *fplll* using FP

and determinant, just like L2. However, we notice the advantage is not stable. This is because the early reduction affects differently for different dimensions. Overall, as shown in Figure 2, with dimension grows, we accelerate the reduction by approximately 20% for $\beta \geq 20d$. In cryptanalysis, one usually needs to deal with lattice with massive dimension and/or determinants, for instance, the Coppersmith-Shamir's technique [7] against an NTRU cryptosystem [10], so it is still helpful to use adaptive precisions when $d \geq 128$ and $\beta \geq 20d$.

## 5 Conclusion

In this paper, we presented an adaptive floating point precision LLL algorithm. The cost of reduction relies heavily on the precision of the floating point, and the precision used in $\mathrm{L}^2$ in fact overkills the problem. Therefore, we presented an approach that adaptively handles the precision. In practice, our algorithm is always faster than the proved version of $\mathrm{L}^2$. It also out-performs the fastest implementation so far in most cases.

## References

1. mpfr library. online. `http://www.mpfr.org/`.
2. SVP CHALLENGE. online. `http://www.latticechallenge.org/svp-challenge/index.php`.
3. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. the user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.
4. M. Bremner. *Lattice Basis Reduction: An Introduction to the LLL Algorithm and Its Applications.* Pure and Applied Mathematics. CRC PressINC, 2012.
5. D. Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Maurer [13], pages 178–189.
6. D. Coppersmith. Finding a small root of a univariate modular equation. In Maurer [13], pages 155–165.
7. D. Coppersmith and A. Shamir. Lattice attacks on ntru. In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 52–61. Springer, 1997.
8. N. Gama and P. Q. Nguyen. Predicting lattice reduction. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, pages 31–51, Berlin, Heidelberg, 2008. Springer-Verlag.
9. D. Goldstein and A. Mayer. On the equidistribution of hecke points. In *Forum Mathematicum.*, volume 15, pages 165–189, 2006.
10. J. Hoffstein, J. Pipher, and J. H. Silverman. Ntru: A ring-based public key cryptosystem. In J. Buhler, editor, *ANTS*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.
11. A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen, Springer-Verlag*, 261:513–534, 1982.
12. L. Lovász. *An Algorithmic Theory of Numbers, Graphs and Convexity*, volume 50 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM Publications, 1986.

13. U. M. Maurer, editor. *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*. Springer, 1996.
14. D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems, A Cryptographic Perspective*. Kluwer Academic Publishers, 2002.
15. H. Minkowski. *Geometrie der Zahlen*. B. G. Teubner, Leipzig, 1896.
16. P. Q. Nguyen and D. Stehle. Floating-point LLL revisited. In *Advances in Cryptology - Eurocrypt 2005, Lecture Notes in Computer Science 3494, Springer-Verlag*, pages 215–233, 2005.
17. P. Q. Nguyen and D. Stehle. LLL on the average. In *7th International Symposium on Algorithmic Number Theory (ANTS 2006)*, pages 238–256, 2006.
18. P. Q. Nguyen and D. Stehlé. An lll algorithm with quadratic complexity. *SIAM J. Comput.*, 39(3):874–903, 2009.
19. P. Q. Nguyen and B. Valle. *The LLL Algorithm: Survey and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009.
20. X. Pujol, D. Stehle, and D. Cade. fplll library. online. `http://perso.ens-lyon.fr/xavier.pujol/fplll/`.
21. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
22. C.-P. Schnorr. A more efficient algorithm for lattice basis reduction. *J. Algorithms*, 9(1):47–62, 1988.
23. C.-P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.
24. V. Shoup. NTL - A Library for Doing Number Theory. online. `http://www.shoup.net/ntl/index.html`.