# Lattice cryptography: from cryptanalysis to New Foundations

Woerden, W.P.J. van

# Part II

# Short and Close Lattice Vectors

# CHAPTER 3

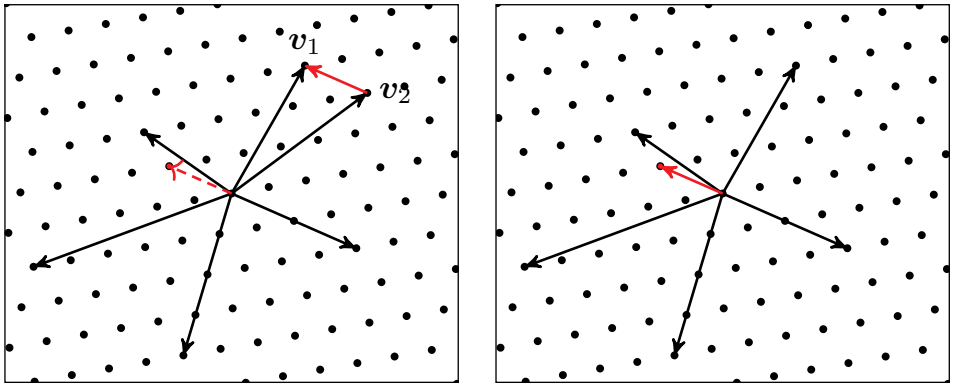
# Theory of Lattice Sieving

*This chapter gives an introduction to the state-of-the-art of lattice sieving.*

## 3.1 Introduction

The hardness of the Shortest Vector Problem (SVP) is fundamental to the security of lattice-based cryptography; an efficient algorithm for SVP would break almost all lattice-based schemes. More precisely, the best asymptotic and concrete attacks on many schemes boil down to solving SVP in some (possibly lower dimensional) lattice, and thus parameters for lattice-based schemes are directly influenced by the complexity of solving SVP.

Lattice sieving algorithms solve SVP in single exponential time, making them asymptotically superior to enumeration techniques running in super-exponential time, at the cost of also using single exponential space. The central idea of sieving algorithms is to start with a large list of (long) lattice vectors, and to find many sums and differences of these vectors that are shorter. These shorter combinations are inserted back into the list, possibly replacing longer vectors, and this




Find close vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in L$. Replace $\boldsymbol{v}_2 \leftarrow \boldsymbol{v}_1 - \boldsymbol{v}_2$.

Figure 3.1: Lattice Sieving

process is repeated until the list contains many short vectors, among which (hopefully) a shortest one of the lattice. See Figure 3.1 for one such iteration.

The first lattice sieving algorithm [AKS01] was proposed in 2001 by Ajtai, Kumar, and Sivakumar (AKS), giving the first provable algorithm that solved SVP in single exponential time $2^{O(n)}$ for a rank $n$ lattice. By choosing the AKS parameters carefully, Nguyen et al. later showed that the AKS sieve can provably solve SVP in time $2^{5.9n+o(n)}$ and space $2^{2.95n+o(n)}$. In a further line of work [PS09; MV10; HPS11a] algorithmic and analytic improvements brought the constants down to time $2^{2.456n+o(n)}$ and space $2^{1.233n+o(n)}$. Provably solving (worst-case) SVP with lattice sieving leads to many technical problems, such as showing that we can actually find enough short combinations and in particular that they are new, i.e., they are not present in our list yet; unfortunately, side-stepping these technicalities leads to the mentioned high time and memory complexities.

In contrast, for cryptanalysis we are interested in the average-case behaviour of lattice sieving algorithms. Instead of provable results these algorithms are based on heuristics, which are often verified experimentally, or which can only be proven (partially) for average-case instances. The first and simplest of these practical sieving algorithms by Nguyen and Vidick uses a list of $N = (4/3)^{n/2+o(n)} = 2^{0.2075n+o(n)}$ vectors and runs in time $N^{2+o(1)} = 2^{0.415n+o(d)}$ by repeatedly checking

all pairs $\mathbf{v} \pm \mathbf{w}$ [NV08]. The list size of $(4/3)^{n/2+o(n)}$ is the minimal number of vectors that is needed in order to keep finding enough shorter pairs.

In a line of works [Laa15; BGJ15; BL16; BDGL16] the time complexity was gradually improved to $2^{0.292n+o(n)}$ by nearest neighbour searching techniques to find close pairs more efficiently [IM98]. Instead of checking all pairs they first apply some bucketing strategy in which close vectors are more likely to fall into the same bucket. By only considering the somewhat-close pairs inside each bucket, the total number of checked pairs can be decreased.

To lower the space cost below $N = 2^{0.2075+o(n)}$ one has to look beyond sums and differences of pairs, and consider triples $\mathbf{u} \pm \mathbf{v} \pm \mathbf{w}$ or more generally $k$-tuples [BLS16; HK17; HKL18]. The current best triple sieve [HKL18] has a space complexity of $2^{0.1887n+o(n)}$, but comes with a higher time complexity of $2^{0.3588n+o(n)}$.

### 3.1.1 Organisation

In Section 3.2 we introduce the basics of lattice sieving algorithms and their heuristic analysis. Section 3.3 introduces some Nearest Neighbour Search techniques to obtain asymptotically faster sieving algorithms. In Section 3.4 we introduce more advanced practical sieving techniques that give significant polynomial and even subexponential speed-ups. In Section 3.5 we give an overview of the General Sieve Kernel (G6K), the current state-of-the-art framework and implementation for (lattice reduction via) lattice sieving.

## 3.2 Heuristic lattice sieving

In this section we introduce a very basic lattice sieving algorithm inspired by the first practical sieving algorithm [NV08], and explain heuristically why it works. Lattice sieving algorithms start with a large list $L_0 \subset \mathcal{L}$ consisting of $N$ long lattice vectors. Given any basis one can always produce long lattice vectors by discrete Gaussian sampling, or simply by taking small random combinations of the basis vectors. Note that due to the additive structure of the lattice, for any two lattice vectors $\mathbf{x}, \mathbf{y} \in L_0$, the difference $\mathbf{x} - \mathbf{y}$ is still a lattice vector. And, in particular the difference might be a *shorter* lattice

vector. The idea behind lattice sieving is to compute the difference $\mathbf{x} - \mathbf{y}$ for all lattice vectors $\mathbf{x}, \mathbf{y} \in L_0$ and store those that are short enough in a new list $L_1$. We call such pairs $(\mathbf{x}, \mathbf{y})$ a *reduction*.

**Definition 65** (Reduction). *Let $R > 0$ be some length bound (clear from the context), and let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ be (lattice) vectors, we call $(\mathbf{x}, \mathbf{y})$ a reduction or a reducing pair if*

$$\|\mathbf{x} - \mathbf{y}\| \leq R.$$

With the appropriate length bound the new list $L_1$ contains shorter vectors than the original list $L_0$, and this process is repeated with more lists $L_2, L_3, \ldots$ and decreasing bounds $R_2 \geq R_3 \geq \ldots$ until some saturation condition is met (which will be discussed later). This basic sieving algorithm is summarized in Algorithm 2. Given the algorithm as stated it might as well happen that such reductions do not exist, or at least not enough of them, such that after some iterations we have $L_i = \{\mathbf{0}\}$. Clearly if we add more vectors to the initial list $L_0$, then more pairs are considered, and thus we expect to find more reductions.

**Algorithm 2:** Lattice sieving algorithm.

**Input :** A basis $\mathbf{B}$ of a lattice $\mathcal{L}$, list size $N$, a saturation radius $R$ and a progress factor $2/3 < \gamma < 1$.
**Output:** A list $L$ of short vectors saturating the ball of radius $R$.

Sample a list $L_0 \subset \mathcal{L}$ of size $N$.
$i \leftarrow 0$.
**while** $L_i$ *does not saturate the ball of radius* $R$ **do**
  $R_{i+1} \leftarrow \gamma \cdot \max_{\mathbf{v} \in L_i} \|\mathbf{v}\|$
  $L_{i+1} = \emptyset$.
  **for** *every pair* $\mathbf{v}, \mathbf{w} \in L_i$ **do**
    **if** $\mathbf{v} - \mathbf{w} \notin L_{i+1}$ ***and*** $\|\mathbf{v} - \mathbf{w}\| \leq R_{i+1}$ **then**
      $L_{i+1} \leftarrow L_{i+1} \cup \{\mathbf{v} - \mathbf{w}\}$
    **end**
  **end**
  $i \leftarrow i + 1$
**end**
**return** $L_i$

So how large does the initial list $N = |L_0|$ need to be to find enough reductions?

### Reduction probability

Let's first dive deeper into sufficient conditions for lattice vectors to give a reduction. Pairs $(\mathbf{x}, \mathbf{y})$ give a reduction if their difference is small enough, i.e., if they are somewhat close to each-other. Alternatively this means that the vectors $\mathbf{x}$ and $\mathbf{y}$ have a small angle to each-other. This can be made rigorous in terms of the inner product.

**Lemma 66.** *Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ be (lattice) vectors, then $(\mathbf{x}, \mathbf{y})$ is a reduction for some length bound $R > 0$ if and only if*

$$\langle \mathbf{x}, \mathbf{y} \rangle \geq \frac{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - R^2}{2}.$$

*in particular if $\|\mathbf{x}\| = \|\mathbf{y}\| = R$ then $(\mathbf{x}, \mathbf{y})$ is a reduction if and only if*

$$\langle \mathbf{x}/\|\mathbf{x}\|, \mathbf{y}/\|\mathbf{y}\| \rangle \geq \frac{1}{2}.$$

A normalized inner product of at least $\frac{1}{2}$ corresponds to the vectors having an angle less than $\pi/3$ between each-other. Independent of the length of the vectors $\mathbf{x}, \mathbf{y}$, if $R \geq \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ then the condition that the angle is less than $\pi/3$ is sufficient. Note that in some sense this is the worst-case scenario, the angle condition is only necessary when the vectors have the same length.

In order to say something about the probability that a pair $(\mathbf{x}, \mathbf{y}) \in L_i \times L_i$ gives a reduction we have to assume that these vectors follow some distribution. This is where the heuristics come in. Recall that the Gaussian Heuristic indicates that given some ball $R \cdot \mathcal{B}^n$ of radius $R$, there are approximately $|R \cdot \mathcal{B}^n \cap \mathcal{L}| = (R/\operatorname{gh}(\mathcal{L}))^n$ lattice vectors in it, and those lattice vectors are uniformly distributed over the ball. In addition, for most use cases, any dependencies between the lattice vectors is ignored. Vectors that are uniformly distributed over a ball are in particular uniformly distributed over the sphere after normalization. Given the sufficient condition on the angle we can thus work with the following weaker heuristic.

**Heuristic 67.** *For nonzero list vectors $\mathbf{v} \in L_i \setminus \{\mathbf{0}\}$, the directions $\mathbf{v}/\|\mathbf{v}\|$ are independently uniformly distributed over the sphere $\mathcal{S}^{n-1}$.*

This heuristic matches what we observe in practice, and one could argue that this is in fact the worst-case distribution when ones goal is to find pairs that have a small angle[1], i.e., if the distribution is less uniform and biased into a certain direction then one would expect to find more pairs with a small angle.

As a technical motivation, ignoring dependency issues, note that if we sample the initial list from a discrete Gaussian with large enough parameter, then their directions are indeed close to uniform. Now if vectors $\mathbf{x}, \mathbf{y}$ have uniform directions, then their difference $\mathbf{x} - \mathbf{y}$ also has a uniform direction. Furthermore heuristically there is no direct relation between the direction of $\mathbf{x} - \mathbf{y}$ and its shortness, so the new list also contains vectors with uniform directions.

Given this heuristic we can compute a lower bound on the probability that any two vectors give a reduction. Note that for this only one of the two vectors has to follow the heuristic.

**Lemma 68.** *For any $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$, let $\mathbf{y} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ be such that $\mathbf{y}/\|\mathbf{y}\|$ is uniform over $\mathcal{S}^{n-1}$, and let $R \geq \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ be a length bound, then the probability that $(\mathbf{x}, \mathbf{y})$ is a reduction is at least*

$$\Pr[\|\mathbf{x} - \mathbf{y}\| \leq R] \geq (3/4)^{n/2+o(n)},$$

*with equality if $R = \|\mathbf{x}\| = \|\mathbf{y}\|$.*

*Proof.* Let $\tilde{\mathbf{x}} = \mathbf{x}/\|\mathbf{x}\|, \tilde{\mathbf{y}} = \mathbf{y}/\|\mathbf{y}\|$, the condition $\langle \tilde{\mathbf{x}}, \tilde{\mathbf{y}} \rangle \geq \frac{1}{2}$ implies that $(\mathbf{x}, \mathbf{y})$ is a reduction with the length bound $R \geq \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$. The condition is necessary only when $R = \|\mathbf{x}\| = \|\mathbf{y}\|$. For any $\tilde{\mathbf{x}} \in \mathcal{S}^{n-1}$ the probability

$$\Pr_{\tilde{\mathbf{y}} \sim \mathcal{S}^{n-1}}\left[\langle \tilde{\mathbf{x}}, \tilde{\mathbf{y}} \rangle \geq \frac{1}{2}\right] = \frac{\mathrm{vol}\left(\mathcal{C}^{n-1}_{\tilde{\mathbf{x}}, \frac{1}{2}}\right)}{\mathrm{vol}(\mathcal{S}^{n-1})} = \mathcal{C}^{n-1}(1/2) = (3/4)^{n/2} \cdot n^{O(1)},$$

is given by the relative volume $\mathcal{C}^{n-1}(1/2)$ of the spherical cap, whose asymptotic formula is given by Lemma 46. □

[1] Technically, the existence of a very good non-lattice kissing number configurations would imply a counter-example to this statement, but it is not clear that such a configuration exists, and in particular such a distribution does not occur naturally in practice.

Following Lemma 68 we have to consider about $(4/3)^{n/2+o(n)}$ pairs to find at least a single reduction. Given a list $L_i$ of length $N$ and a length bound of $R = \max_{\mathbf{v}\in L_i} \|\mathbf{v}\|$ we can consider about $\approx \frac{1}{2}N^2$ pairs, and the total number of reductions will heuristically be at least $N^2 \cdot (3/4)^{n/2+o(n)}$. These reductions form the new list $L_{i+1}$ which again has to be of size close to $N$. Solving the list balancing equation

$$N = N^2 \cdot (3/4)^{n/2+o(n)},$$

we thus need a list of size $N = (4/3)^{n/2+o(n)}$ to make sure that the lists do not decrease in size (ignoring duplicates).

In Algorithm 2 the length bound is set slightly smaller at $R = \gamma \cdot \max_{\mathbf{v}\in L_i} \|\mathbf{v}\|$ for some progress factor $\gamma < 1$. Depending on the specific factor one would have to increase the list size slightly; but by setting e.g., $\gamma = 1 - \frac{1}{n}$ an asymptotic list size of $N = (4/3)^{n/2+o(n)}$ is again sufficient, while still obtaining an exponential decrease in the length bounds $R_i$.

### Duplicates

In the above analysis we are assuming that each of the found reductions $\mathbf{x} - \mathbf{y}$ is distinct. This cannot stay true as otherwise we would eventually find nonzero vectors shorter than $\lambda_1(\mathcal{L})$ under Heuristic 67. Suppose the lattice is normalized to $\mathrm{gh}(\mathcal{L}) = 1$, i.e., such that the Gaussian Heuristic indicates that $\lambda_1(\mathcal{L}) \approx 1$. Asymptotically, for large $n$ almost all of the reductions in the new list have length close to the length bound $R$ (say between $0.99R$ and $R$), and given Heuristic 67 it is natural to assume that these reductions give vectors close to uniform among the lattice vectors in that thin layer. By the Gaussian Heuristic there exist about $R^{n+o(n)}$ such lattice vectors. Given $(4/3)^{n/2+o(n)}$ uniform samples from this set we only expect a significant $\Theta(1)$ fraction of duplicates when $R \leq \sqrt{4/3} + o(1)$. A natural saturation condition is thus to stop whenever $L_i$ contains a significant fraction of the lattice vectors of length at most $\sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L})$, and before this happens the relative number of duplicates is expected to be insignificant.

**Definition 69** (Saturation). *For a rank $n$ lattice $\mathcal{L}$, we say that a list $L \subset \mathcal{L}$ of lattice vectors is saturating (a ball) with radius $R$ if*

$$|\mathcal{L} \cdot R \cdot \mathcal{B}^n| \geq \tfrac{1}{2} \cdot (R/\,\mathrm{gh}(\mathcal{L}))^n$$

The saturation condition is usually applied with radius $R = \sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L})$. The constant $\frac{1}{2}$ is somewhat arbitrary and can be tweaked in practice. A lower constant makes it easier to reach saturation but results in less short vectors in the final list. A larger constant (of at most 1) makes saturation harder to reach and increases the occurrence of duplicates, but results in more short vectors in the final list.

### Heuristic runtime analysis

Again we consider a lattice $\mathcal{L}$ that is normalized such that $\mathrm{gh}(\mathcal{L}) = 1$. Suppose we set $\gamma = 1 - \frac{1}{n}$, then a list size of $N = (4/3)^{n/2+o(n)}$ is enough, and there are $N^2 = (4/3)^{n+o(n)}$ reductions to check every iteration, each at the cost of $O(n)$ arithmetic operations to compute an inner product. Checking for duplicates can be done in $O(\log N)$ or amortized $O(1)$ vector operations per vector by keeping the list sorted or by a hash-set.

By first LLL reducing the basis and then sampling vectors by taking small random combinations (or by Gaussian sampling) we can construct an initial list $L_0$ of vectors such that $R_1 = \max_{\mathbf{v} \in L_0} \|\mathbf{v}\| = 2^{O(n)}$. Given that $\gamma^k = (1 - \frac{1}{n})^k$ decreases like $(1/e)^{k/n}$ we obtain $R_i \approx \sqrt{4/3}$ after at most a polynomial number of $k = O(n^2)$ iterations, and following the previous discussion the saturation condition will (have) trigger(ed).

In total the complexity is thus dominated by the $\mathrm{poly}(n) \cdot N^2 = (4/3)^{n+o(n)}$ time cost of checking all pairs and the $\mathrm{poly}(n) \cdot N = (4/3)^{n/2+o(n)}$ memory cost of storing the lists.

### A single list

Algorithm 2 uses multiple lists $L_1, L_2, \ldots$ to clarify the steps in the heuristic analysis. In practice it is common to use a single list, where new reduced vectors $\mathbf{x} - \mathbf{y}$ either replace the longest of $\mathbf{x}$ or $\mathbf{y}$, or replace just the longest element in the list. The list and length bound are then continuously updated and the vectors are getting shorter and shorter until saturation is achieved. This has the added benefit of explicitly reusing all the short enough vectors found so far (although this was implicitly done in Algorithm 2 by forcing $\mathbf{0}$ to be part of each list).

One could pick random pairs to reduce, which might result in a loss of performance due to rechecking the same pair. The approach of the Gauss Sieve [MV10] is to split the list into two parts: the queue part $Q$ and the list part $L$. At the start of the sieve all vectors are in the queue part, and the list part is empty. If the queue part becomes too small before saturation is achieved we sample more vectors into it.

A queue vector $\mathbf{v} \in Q$ is only inserted in the list part if it does not form a reduction with any of the list vectors (with $R = \max\{\|\mathbf{v}\|, \|\mathbf{w}\|\}$ for $\mathbf{w} \in L$). This is achieved by explicitly checking for reductions between $\mathbf{v}$ and all list vectors, where any reduction $(\mathbf{v}, \mathbf{w})$ replaces the longest of the two vectors, and is inserted back into the queue part (if it was not in the database yet). By following this strategy any pair of vectors is only checked for a reduction once (except if a vector disappears and later reappears in the database), preventing most of the duplicate checks, and giving a significant speed-up in lowish dimensions.

Another improvement is to not store both the vector $\mathbf{x}$ and its negation $-\mathbf{x}$, which saves a factor 2 in memory. Additionally we can check if either $\pm(\mathbf{x}-\mathbf{y})$ or $\pm(\mathbf{x}+\mathbf{y})$ is short by looking at the absolute inner product $|\langle \mathbf{x}, \mathbf{y} \rangle|$, which saves a factor 4 in the number of inner products we have to compute.

### Triple sieve

In Section 3.3 we will explain how to decrease the time complexity below $N^2 = (4/3)^{n+o(n)}$, by making reductions easier to find. However all such methods do no improve the memory usage below $N = (4/3)^{n/2+o(n)}$, as we still require the same number of reductions.

To improve the memory cost we have to go beyond pairs and consider triples $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ such that $\mathbf{x} \pm \mathbf{y} \pm \mathbf{z}$ is short or even $k$-tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ [BLS16; HK17; HKL18]. The probability that a triple uniformly sampled from the sphere gives a reduction is about $(16/27)^{n/2+o(n)}$, which is much smaller than for a pair. But at the same time there are about $N^3$ triples to consider, leading to the list balancing equation

$$N = (16/27)^{n/2+o(n)} \cdot N^3,$$

with solution $N = (27/16)^{n/4+o(n)} = 2^{0.1887n+o(n)}$ compared to a list size of $(4/3)^{n/2+o(n)} = 2^{0.2075n+o(n)}$ for the sieve based on pairs. This comes at the cost of a higher time, as we now need to check $N^3 = 2^{0.5662n+o(n)}$ triples. Considering tuples of $k \geq 4$ vectors further decreases the memory cost, but with a large impact on the runtime.

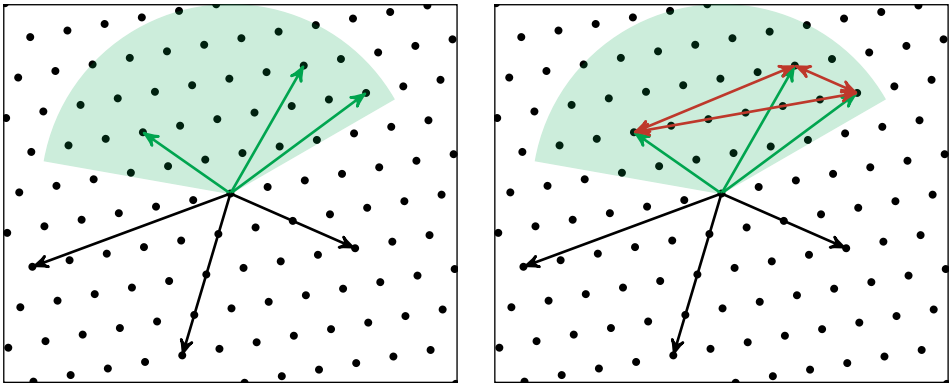
**Provable sieves**

In the literature on lattice sieving there is a large gap between the heuristic (and real-world average-case) performance of lattice sieving algorithms and the provable sieving algorithms. Heuristically we need a list size of about $(4/3)^{n/2+o(n)}$ to keep finding enough reduction, and this seems realistic in practice. However in the worst-case we could construct such a list that has no reduction at all. For a naive provable result on finding enough reductions one would have to at least square the list size to $(4/3)^{n+o(n)} = 2^{0.415n+o(n)}$. The exact number that is needed is directly related to the non-lattice variant of the kissing number, which bound can be improved to $2^{0.401n+o(n)}$.

Still the problem remains that many of the found reductions might collide, and this is where most of the technical difficulties in provable sieving algorithms stem from. One technical solution is to add extra random errors to the lattice point in order to somehow 'hide the lattice structure' from the sieving algorithm, which allows one to prove results about the distribution of the list and the probability of obtaining duplicates, but this comes at the cost of significantly increasing the asymptotic complexity. When allowing for a constant approximation factor, e.g., that the algorithm returns provably only a vector of size $\mu \cdot \lambda_1(\mathcal{L})$ for some constant $\mu > 1$, this increase in complexity can be prevented, leading to a $2^{0.802n+o(n)}$ time and $2^{0.401n+o(n)}$ space algorithm [AUV19; EV22; RV22], surprisingly not only in the standard Euclidean norm, but for any $p$-norm.

## 3.3 Bucketing and sieving variants

Recall from Section 3.2 that for a (pair-)sieve we need a list of $N = (4/3)^{n/2+o(n)}$ lattice vectors, and we need to find all the reductions among the $N^2$ pairs. Naturally the naive algorithm that checks all pairs takes a time complexity on the order of $N^2$.



Create a bucket $B \subset L$. Search reducing pairs in $B$.

Figure 3.2: Lattice sieving with bucketing.

This can be improved by bucketing methods. The idea is to first group vectors together that point somewhat in the same direction. Then within each group each pair is more likely to give a reduction; and this is where we improve the time complexity. To find all reductions we might have to repeat the bucketing process (after randomizing) for multiple iterations. See Algorithm 3.

Suppose that in each iteration we have $m$ buckets $B_1, \ldots, B_m$. Each of the $N$ vectors is added to on average $M$ buckets, depending on some bucket condition, at a cost of $T_b$ per vector. For convenience we will for now assume that $M \approx 1$, such that assuming that the buckets are well balanced, they are of size $|B_i| \approx N/m$.

We assume that any pair of vectors that lies in the same bucket has (at least) some conditional probability $p$ to give a reduction. To find $N$ reductions we thus have to check in total about $N/p$ bucketed pairs, and when $p \gg N^{-1}$ this is less than in the naive algorithm.

This is only helpful if the cost of the bucketing process is less or equal to $\tilde{O}(N/p)$. Per iteration we have $m$ buckets and we find on the order of $p \cdot (N/m)^2$ reductions per bucket. To find $N$ reductions we thus have to repeat $r = \frac{m}{N \cdot p}$ many iterations. At each iteration we have to find an appropriate bucket for each of the $N$ vectors at a cost of $T_b$ per vector, the total cost of bucketing is thus given by

$$r \cdot N \cdot T_b = \frac{n}{N \cdot p} \cdot N \cdot T_b = \frac{m}{p} \cdot T_b.$$

As an alternative explanation of the bucketing cost note that each vector has after bucketing an expected number of $|B_i| \cdot p = \frac{N}{m} \cdot p$ reductions, to achieve a total of $N$ reductions we thus have to bucket $N/(\frac{N}{m} \cdot p) = m/p$ vectors with a total cost of $\frac{m}{p} \cdot T_b$.

The total time complexity of bucketing and checking pairs for reductions is thus given by

$$T = \tilde{O}\left(\frac{m \cdot T_b}{p} + \frac{N}{p}\right).$$

Investing more time in bucketing often allows for more and higher quality buckets, which results in a larger reduction probability $p$ and thus in a lower reduction cost. Investing less time in bucketing in general leads to a higher reduction cost. For a fixed bucketing algorithm the optimal parameters are (often) such that the above terms are balanced, which means that $m \cdot T_b \approx N$.

**Algorithm 3:** Bucketed lattice sieving algorithm.

**Input :** A basis $\mathbf{B}$ of a lattice $\mathcal{L}$, list size $N$, and a saturation radius $R$.
**Output:** A list $L$ of short vectors saturating the ball of radius $R$.

```
Sample a list L ⊂ ℒ of size N.
while L does not saturate the ball of radius R do
    B_1, ..., B_m ← Bucket(L, m).
    for every bucket B_i do
        for every pair v, w ∈ B_i do
            if v − w ∉ L and ‖v − w‖ < max_{u∈L} ‖u‖ then
                Replace longest vector in L by v − w.
            end
        end
    end
end
return L
```

We now discuss two different bucketing algorithms, and their optimal time complexity. The first is relatively simple and very practical, while the second is somewhat more complicated but reaches the best known asymptotic time complexity.

### BJG1 sieve

We start with the simple 1 layer variant of [BGJ15], better known as the BGJ1 sieve. Recall that the idea of bucketing is to subdivide our large list into groups of vectors pointing somewhat in the same direction. We can take this quite literal and pick $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathcal{S}^{n-1}$ bucket directions uniformly at random, and put a vector $\mathbf{v}$ in bucket $i$ if it points mostly in the direction of $\mathbf{b}_i$, i.e., if $\langle \mathbf{b}_i/\|\mathbf{b}_i\|, \mathbf{v}/\|\mathbf{v}\| \rangle \geq \alpha$ for some parameter $\alpha > 0$. For a vector $\mathbf{v}$ we can compute the above inner product for each of the $m$ buckets, and thus we obtain a bucketing cost of $T_b = O(m \cdot n)$ per vector.

Geometrically each bucket presents a spherical cap $\mathcal{C}^{n-1}_{\mathbf{b}_i,\alpha}$ in some random direction. The probability that a (normalized) lattice vector lies in such a cap is given by its relative volume $\mathcal{C}^{n-1}(\alpha) = (1-\alpha^2)^{n/2+o(n)}$, and thus we need $m = (1-\alpha^2)^{-n/2+o(n)}$ such buckets to make sure each vector is on average part of $M \approx 1$ bucket. I.e., geometrically we need about $(1-\alpha^2)^{-n/2+o(n)}$ such spherical caps to cover each point on the sphere (in expectation).

Now what about the improved probability $p$, that a pair inside a bucket gives a reduction. Recall that before bucketing this probability is about $(3/4)^{n/2+o(n)}$. For a pair inside a spherical cap with parameter $\alpha$ the probability is, as expected, better.

**Lemma 70.** *Let $\mathbf{b} \in \mathcal{S}^{n-1}$ be a direction, and let $\mathbf{v}, \mathbf{w} \sim \mathcal{S}^{n-1}$. Then for any parameter $\sqrt{1/2} > \alpha > 0$ we have*

$$\Pr\left[\langle \mathbf{v}, \mathbf{w} \rangle \geq \frac{1}{2} \middle| \langle \mathbf{v}, \mathbf{b} \rangle \geq \alpha \text{ and } \langle \mathbf{w}, \mathbf{b} \rangle \geq \alpha\right] = \left(\frac{\frac{3}{4} - \alpha^2}{(1-\alpha^2)^2}\right)^{n/2+o(n)}.$$

*Proof.* Let $A$ be the event that $\langle \mathbf{v}, \mathbf{w} \rangle \geq \frac{1}{2}$, and let $B$ be the event that $\langle \mathbf{v}, \mathbf{b} \rangle \geq \alpha$ *and* $\langle \mathbf{w}, \mathbf{b} \rangle \geq \alpha$. Note that $\Pr[A] = \mathcal{C}^{n-1}(1/2)$, and $\Pr[B] = \mathcal{C}^{n-1}(\alpha)^2$. For $\gamma \in [\frac{1}{2}, 1]$ the probability density function for the event $\langle \mathbf{v}, \mathbf{w} \rangle = \gamma$ is asymptotically proportional to $\mathcal{C}^{n-1}(\gamma) = (1-\gamma^2)^{n/2}$. Furthermore the probability $\Pr[B|\langle \mathbf{v}, \mathbf{w} \rangle = \gamma]$ can geometrically be interpreted as the wedge $\mathcal{W}^{n-1}_{\mathbf{v},\mathbf{w},\alpha} := \mathcal{C}^{n-1}_{\mathbf{v},\alpha} \cap \mathcal{C}^{n-1}_{\mathbf{w},\alpha}$ with relative volume $\mathcal{W}^{n-1}(\alpha, \gamma)$. We have

$$\mathcal{W}^{n-1}(\alpha, \gamma) \cdot \mathcal{C}^{n-1}(\gamma) = ((1 + \gamma - 2\alpha^2)(1-\gamma))^{n/2+o(n)},$$

and for $0 < \alpha < \sqrt{1/2}$ the constant $(1 + \gamma - 2\alpha^2)(1-\gamma)$ is strictly maximized for $\gamma = \frac{1}{2}$. Asymptotically, we thus have $\Pr[B|A] =$

$\Pr[B|\langle \mathbf{v}, \mathbf{w}\rangle \geq \frac{1}{2}] = 2^{o(n)} \cdot \Pr[B|\langle \mathbf{v}, \mathbf{w}\rangle = \frac{1}{2}] = (1 - \frac{4}{3}\alpha^2)^{n/2+o(n)}$. Now by Bayes' theorem we have

$$\Pr[A|B] = \frac{\Pr[B|A] \cdot \Pr[A]}{\Pr[B]} = \frac{(1 - \frac{4}{3}\alpha^2)^{n/2+o(n)} \cdot \mathcal{C}^{n-1}(1/2)}{\mathcal{C}^{n-1}(\alpha)^2} = \left(\frac{\frac{3}{4} - \alpha^2}{(1-\alpha^2)^2}\right)^{n/2+o(n)}.$$

□

To summarize we have $m = (1-\alpha^2)^{-n/2+o(n)}, T_b = \tilde{O}(m)$, and $p = (\frac{3}{4} - \alpha^2/((1-\alpha^2)^2))^{n/2+o(n)}$. Now let us pick $\alpha$ such that the total time complexity is minimized. From $N = m \cdot T_b = \tilde{O}(m^2)$ we obtain $(1-\alpha^2)^{-n/2+o(n)} = m \approx N^{1/2} = (4/3)^{n/4+o(n)}$ and thus $\alpha = \sqrt{1 - \sqrt{4/3}} + o(1) \approx 0.366$. This gives $p = (\sqrt{4/3} - \frac{1}{3})^{n/2+o(n)} = 2^{-0.142n+o(n)}$, and a total time complexity of $N/p = 2^{0.349n+o(n)}$.

When applying the same bucketing strategy recursively the time complexity can be further reduced to $2^{0.311n+o(n)}$ [BGJ15]. However, the recursion comes with large overheads, and therefore it is unclear how practical this version is.

### BDGL sieve

The asymptotically optimal bucketing method from [BDGL16] is similar to BGJ1 as in that it is based on spherical caps. The difference is that in contrast to BGJ1 the bucket centers are not arbitrary but structured, allowing to find the correct bucket without having to compute the inner product with each individual bucket center. This allows us to reduce the bucketing cost below $T_b = \tilde{O}(m)$ per vector.

Following [BDGL16], such a bucketing strategy looks as follows. First we split the dimension $n$ into $k$ smaller blocks of similar dimensions $n_1, \ldots, n_k$ that sum up to $n$. In order to randomize this splitting over different iterations one first applies a random orthonormal transformation $\mathbf{Q}$ to each input vector. Then the set $C$ of bucket centers is constructed as a direct product of random local bucket centers, i.e., $C = \pm C_1 \times \pm C_2 \cdots \times \pm C_k$ with $C_b \subset \mathbb{R}^{n_b}$. A (normalised) list vector $\mathbf{v}$ is then similarly split into parts $\mathbf{v}_1, \ldots, \mathbf{v}_k$, and we have to find the bucket centers $(\mathbf{c}_1, \ldots, \mathbf{c}_k) \in C$ such that $\sum_i \langle \mathbf{c}_i, \mathbf{v}_i\rangle \geq \alpha$. Note that to

| Bucketing | $T_b$ | $\lvert B_i\rvert$ | $c_{\text{time}}$ |
|---|---|---|---|
| None | 0 | $\tilde{O}(N)$ | 0.415 |
| BGJ1 | $\tilde{O}(m)$ | $\tilde{O}(N^{1/2})$ | 0.349 |
| BDGL ($k=1$) | $\tilde{O}(m)$ | $\tilde{O}(N^{1/2})$ | 0.349 |
| BDGL ($k=2$) | $\tilde{O}(m^{1/2})$ | $\tilde{O}(N^{1/3})$ | 0.329 |
| BDGL ($k=3$) | $\tilde{O}(m^{1/3})$ | $\tilde{O}(N^{1/4})$ | 0.320 |
| BDGL ($k=\Theta(\log(n))$) | $\tilde{O}(m^{1/k})$ | $\tilde{O}(N^{1/(k+1)})$ | 0.292 |

Table 3.1: Bucketing cost $T_b$ per vector, optimal bucket size $\lvert B_i\rvert$ and the resulting time complexity $2^{c_{\text{time}}n+o(n)}$ for the discussed bucketing algorithms with a space complexity of $N = 2^{0.2075n+o(n)}$.

find the closest global bucket center to $\mathbf{v}$, we only have to pick the closest local bucket centers $\mathbf{c}_i$ to $\mathbf{v}_i$, implicitly considering $m = 2^k \prod_b |C_b|$ global bucket centers at the cost of only $T_b = \sum_b |C_b| \approx O(m^{1/k})$ local inner products. By sorting the local inner products we can also efficiently find all bucket centers within a certain angle. For a fixed number of buckets $m$ we can expect some performance loss compared to BGJ1 as the bucket centers are not perfectly random, but this does not influence the asymptotics. I.e., the analysis of [BDGL16, Theorem 5.1] shows this leads to at most a subexponential loss if $k = O(\log(n))$, and the experimental analysis of [Duc22] shows that it is also reasonable small in practice.

To optimize the parameters we again balance the cost of bucketing and reducing. Note that for $k = 1$ we essentially obtain BGJ1 with buckets of size $\tilde{O}(N^{1/2})$ and a time complexity of $2^{0.349n+o(n)}$. For $k = 2$ or $k = 3$ the optimal buckets become smaller of size $\tilde{O}(N^{1/3})$ and $\tilde{O}(N^{1/4})$ respectively and of higher quality, leading to a time complexity of $2^{0.3294n+o(n)}$ and $2^{0.3198n+o(n)}$ respectively. By letting $k$ slowly grow, e.g., $k = \Theta(\log(n))$ there will only be a subexponential $2^{o(n)}$ number of vectors in each bucket, leading to the best known time complexity of $2^{0.292n+o(n)}$. Note however that a lot of subexponential factors might be hidden inside this $o(n)$, and thus for practical dimensions a rather small value of $k = 2, 3, 4$ might give best results. See Table 3.1 for an overview of the time complexity of the discussed bucketing algorithms.

## 3.4 Advanced lattice sieving

In this section we discuss some more advanced techniques used in lattice sieving. The ideas and implementation tricks in this section only give subexponential, polynomial or even only constant speed-ups, and do not improve the asymptotic time complexity, however they do give significant practical speed-ups. These ideas have largely contributed to pushing the cross-over between enumeration and sieving techniques as low as dimension 70.

### 3.4.1 XOR Popcount SimHash

The main operation in lattice sieving algorithms is the computation of many pairwise inner products $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ given a list $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^n$ of vectors (or alternatively pairwise between two lists), with the goal of finding those pairs $(\mathbf{v}_i, \mathbf{v}_j)$ that are relatively close. The cost of such an inner product computation is $2n$ floating point operations, or $n$ so-called Fused multiply-add (FMA) instructions on say 16 or 32-bit floats. Such floating point operations are complex and require many logical gates to function.

The SimHash filter, better known as the 'XOR popcount trick' [Cha02; Fit+14; Duc18; Alb+19; AGPS20], is a cheaper computation, that already throws out most of the candidates that are far away from each-other, so that the full inner product only has to be computed on the more likely candidates. The idea is that given two normalized close pairs $\mathbf{v}, \mathbf{w} \in \mathcal{S}^{n-1}$ (with e.g., $\langle \mathbf{v}, \mathbf{w} \rangle \geq \frac{1}{2}$), and a random direction $\mathbf{y} \sim \mathcal{U}(\mathcal{S}^{n-1})$, the inner products $\langle \mathbf{v}, \mathbf{y} \rangle$ and $\langle \mathbf{w}, \mathbf{y} \rangle$ are correlated, in particular their signs are more likely to coincide. For a list of random (but fixed) directions $\mathbf{y}_1, \ldots, \mathbf{y}_h \in \mathcal{S}^{n-1}$ we can thus define the SimHash $\mathrm{SH} : \mathcal{S}^{n-1} \to \{0,1\}^h$ by

$$\mathrm{SH} : \mathbf{v} \mapsto \left( \begin{cases} 0, & \text{if } \langle \mathbf{v}, \mathbf{y}_i \rangle \leq 0, \\ 1, & \text{if } \langle \mathbf{v}, \mathbf{y}_i \rangle > 0. \end{cases} \right)_i .$$

If $\mathbf{v}, \mathbf{w}$ are uniformly random, then $\mathrm{SH}(\mathbf{v})$ and $\mathrm{SH}(\mathbf{w})$ have on expectation $h/2$ bits in common. Now if $\mathbf{v}$ and $\mathbf{w}$ or $\mathbf{v}$ and $-\mathbf{w}$ are close, then $\mathrm{SH}(\mathbf{v})$ and $\mathrm{SH}(\mathbf{w})$ are expected to have significantly more or significantly less than $h/2$ bits in common respectively. Assuming that the SimHashes are pre-computed (which takes negligible time when

amortized over many pairs), we can compute the number of bits (not) in common by doing an $h$-bit XOR, followed by a population count instruction (which counts the number of ones), which can be much cheaper than the $2n$ floating point operations. The number of directions $h$ and the filter threshold can be balanced for filter strength and computation cost, for theory on this matter look at [AGPS20]. In practice the SimHash with $h = 256$ has been successfully used up to dimension 128, saving about a factor 10 on computation time [Fit+14; Alb+19].

### 3.4.2 Progressive sieving

Due to the exponential time complexity sieving quickly becomes more costly in higher dimensions, especially when the starting list also consists of very long vectors. Relatively, the cost of sieving in slightly lower dimensions becomes almost negligible. The idea of Progressive Sieving is to first sieve in a lower dimensional projected sublattice, and to slowly increment the dimension and list size, while also lifting the list along the way. As a result, the list vectors are already quite short when the highest dimension is reached, and thus we have to run less high dimensional sieving iterations, at the cost of (more) low dimensional sieving iterations.

Recall that for a lattice $\mathcal{L}$ with basis $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{d-1})$ we denote the projected sublattice $\mathcal{L}(\pi_l(\mathbf{b}_l), \ldots, \pi_l(\mathbf{b}_{r-1}))$ by $\mathcal{L}_{[l:r)}$. There are two main directions to implement this idea (although a combination is also possible) [Duc18; LM18; Alb+19]. Either we form a chain of sublattices

$$\mathcal{L}_{[0:r)} \subsetneq \mathcal{L}_{[0:r+1)} \subsetneq \ldots \subsetneq \mathcal{L}_{[0:d)} = \mathcal{L},$$

for some $1 \leq r < d$ or a chain of projections

$$\mathcal{L}_{[l:d)} \xleftarrow{\pi_l} \mathcal{L}_{[l-1:d)} \xleftarrow{\pi_{l-1}} \cdots \xleftarrow{\pi_1} \mathcal{L}_{[0:d)} = \mathcal{L},$$

for some $1 \leq l < d$. Both ways give significant speed-ups compared to directly sieving in the full dimension, but they both come with their own pros and cons.

For the sublattice approach, after extending, we do not have to lift the short vectors we have in any way, they are by definition part of the super-lattice. With a normal descending basis profile we have

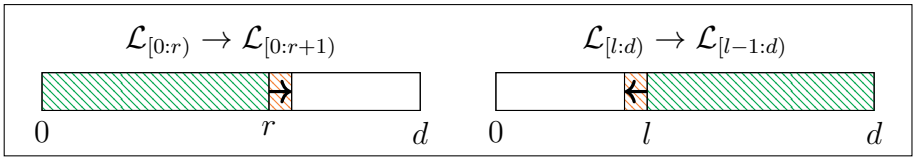


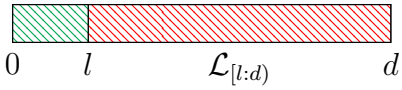

Figure 3.3: Progressive step via a chain of sublattices (on the left) versus a chain of projections (on the right).

$\mathrm{gh}(\mathcal{L}_{[0:r)}) > \mathrm{gh}(\mathcal{L}_{[0:r+1)})$, so after extending the vectors are somewhat longer relative to the local Gaussian Heuristic, but only by a bit. The main disadvantage of the sublattice approach is that after extending, all the short vectors still lie in the same lower rank sublattice, and one has to be careful with sampling new vectors, and in the further sieving process, to also obtain short vectors in $\mathcal{L}_{[0:r+1)} \setminus \mathcal{L}_{[0:r)}$. Without taking explicit care one might implicitly get stuck in a lower rank sublattice, which makes the sublattice approach tricky to work with.

For the projection approach, after extending, the vectors have to be lifted to undo the projection. Undoing e.g., a single projection $\pi_i$ efficiently, while limiting the increase in size, can be done using Babai's nearest plane algorithm. The main disadvantage is that the squared length of a vector can still increase by $\frac{1}{4}\|\tilde{\mathbf{b}}_i\|^2$, which can be significant. At the same time the Gaussian Heuristic predicts that the first minimum of the extended lattice is larger and thus the relative increment in length is somewhat damped by this. The current state-of-the-art sieving implementations use this method, because it is reliable and combines well with the Dimensions for Free technique.

### 3.4.3 Dimensions for free

When a lattice sieving algorithm finishes it does not only recover the shortest vector, but a significant proportion of all lattice vectors up to some radius $R$. For a pair-sieve with a list size of $N = (4/3)^{n/2+o(n)}$ we naturally have this for $R \approx \sqrt{4/3} + o(1)$. The idea of the 'dimensions for free' technique [Duc18], is to sieve in a projected sublattice $\mathcal{L}_{[l:d)}$ for some $l > 0$, and to lift all the short vectors to the full context. If $l$ is not too large, then with high probability one of them lifts to a shortest vector of the full lattice.




To simplify the analysis we simply assume that the sieve in the projected sublattice $\mathcal{L}_{[l:d)}$ computes a list $L$ of all vectors of length up to $\sqrt{4/3}\cdot\mathrm{gh}(\mathcal{L}_{[l:d)})$. The projection $\pi_l(\mathbf{v})$ of a shortest vector $\mathbf{v} \in \mathcal{L}_{[0:d)}$ is thus part of the list $\pi_l(\mathbf{v}) \in L$ if $\|\pi_l(\mathbf{v})\| \leq \sqrt{4/3}\cdot\mathrm{gh}(\mathcal{L}_{[l:d)})$. For not too large $l$, Babai's algorithm then successfully recovers $\mathbf{v}$ back from $\pi_l(\mathbf{v})$.

The maximum value for $l$ for which the above is true depends on the basis profile $(\|\tilde{\mathbf{b}}_0\|, \ldots, \|\tilde{\mathbf{b}}_{d-1|}\|)$. For a well-reduced basis (more on this in Part III) we obtain about $l = O(\log(d)/d)$ 'dimensions for free', and thus we only have to sieve in a lattice of dimension $n = d - O(\log(d)/d)$, leading to a subexponential speed-up.

Note that with the progressive sieve based on a chain of projections we do not have to pick the parameter $l$ a priori. We can increase the chain step by step (decreasing $l$), while at every step we lift the list to the full context. At every step we are increasing the probability that the projection $\pi_l(\mathbf{v})$ is part of the list, and we recover it at the appropriate $l$ without having to specify it beforehand.

Looking from another perspective, every short vector in a projected sublattice $\mathcal{L}_{[l:d)}$ has some probability to lift to a (close to) shortest vector of the full lattice. Therefore to increase the the dimensions for free we could lift any short enough vector that we see on the fly: for example any new vector inserted in the sieving list, or even vectors that are short, but not short enough to be inserted.

We can expect about $l = 20$ (for $d = 100$) to $l = 40$ (for $d = 450$) dimensions for free, and even more when enabling on the fly lifting, making this an extremely important technique that we explicitly have to take into account for concrete hardness estimates.

## 3.5 The General Sieve Kernel

The General Sieve Kernel (G6K) [Alb+19] is a lattice reduction framework based on sieving algorithms that is designed to be 'stateful' instead of treating sieving as a black-box SVP oracle. In short it allows to easily move between different contexts $\mathcal{L}_{[l:r)}$, while maintaining a

database $L$ of short vectors. This encompasses advanced techniques like progressive sieving and dimensions for free. It includes an open-source implementation of the framework that broke several new TU Darmstadt SVP Challenges [SG10] up to dimension 155. This implementation is multi-threaded and low-level optimized, and includes many of the implementation tricks from the advanced lattice sieving literature and some more. In this section we recall the state, instructions, global strategies and some implementation details of G6K.

### 3.5.1 State

Naturally, the state includes a lattice basis $\mathbf{B} \in \mathbb{Z}^{d \times d}$ and its corresponding Gram-Schmidt basis $\tilde{\mathbf{B}}$. The current state keeps track of a *sieving context* $\mathcal{L}_{[l:r)}$ and a *lifting context* $\mathcal{L}_{[\kappa:r)}$ for $0 \leq \kappa \leq l \leq r \leq d$. The sieving dimension will often be denoted by $n := r - l$. There is a database $L$ containing $N$ lattice vectors from the sieving context $\mathcal{L}_{[l:r)}$. To conclude G6K also keeps track of good insertion candidates $\mathbf{i}_\kappa, \ldots, \mathbf{i}_l$ for the corresponding positions in the current lattice basis.

### 3.5.2 Instructions

We begin with several instructions that change the sieving context, and we explain how the database is updated such that it does not get invalidated.

- Extend Left: The extend left operation moves the sieving context from $\mathcal{L}_{[l:r)}$ to $\mathcal{L}_{[l-k:r)}$ for some $0 < k \leq l$. The database vectors are lifted from $\mathcal{L}_{[l:r)}$ to $\mathcal{L}_{[l-k:r)}$ using Babai's nearest plane algorithm in the context $\mathcal{L}_{[l-k:l)}$.

- Shrink Left: The shrink left operation moves the sieving context from $\mathcal{L}_{[l:r)}$ to $\mathcal{L}_{[l+k:r)}$ for some $0 < k \leq r - l$. Each database vector $\mathbf{v} \in L \subset \mathcal{L}_{[l:r)}$ is projected to $\pi_{l+k}(\mathbf{v}) \in \mathcal{L}_{[l+k:r)}$, and duplicates are removed.

- Extend Right: The extend right operation moves the sieving context from $\mathcal{L}_{[l:r)}$ to the super-lattice $\mathcal{L}_{[l:r+k)}$ for some $0 < k \leq d - r$. The database can remain unchanged.

- INSERTION: The insertion operation inserts one of the insertion candidates $\mathbf{i}_k$ (often a short vector) back into the basis, at some position $\kappa \leq k \leq l$. I.e., assuming the vector $\mathbf{i}_k$ is primitive we choose some local (basis) transformation on the context $\mathcal{L}_{[k:r)}$ such that $\mathbf{i}_k$ becomes the new basis vector at position $k$, and the Gram-Schmidt basis $\tilde{\mathbf{B}}$ is updated accordingly. If $k = l$ the database vectors still live in the context $\mathcal{L}_{[l:r)}$ after the basis change, and so we do not have to do anything. Typically however, we will have $k < l$, which makes it less trivial. By carefully choosing the local transformation and by moving to a slightly smaller sieving context $\mathcal{L}_{[l+1:r)}$ we can however still recycle most of the database after an insertion. For prevent confusion we assume that after an insertion operation the sieving context is always moved to $\mathcal{L}_{[l+1:r)}$.

These instruction were focused on moving between different contexts and maintaining the database along the way. We now consider some instructions that act on the database.

- SIEVE: The sieve operation applies a lattice sieving algorithm to the database vectors in order to decrease their size. The end result is an updated database that is saturated in the sieving context $\mathcal{L}_{[l:r)}$ according to some saturation condition. During the sieving process any short enough vector is lifted to the lifting context $\mathcal{L}_{[\kappa:r)}$, and, if short enough, replaces one of the insertion candidates $\mathbf{i}_\kappa, \ldots, \mathbf{i}_l$.

- GROW: The grow operation increases the size of the database by sampling new vectors. This might be needed to have enough vectors to run a sieving algorithm, for example after increasing the dimension of the sieving context. The sampling procedure is not fixed, but preferably generates shortish vectors (by using the known some-what reduced basis, or short vectors in the current database).

- SHRINK: The shrink operation decreases the size of the database by throwing away the longest vectors.

### 3.5.3 Global strategies

The implementation of G6K consists of a high level `Python` layer and a low-level `C++` layer. The earlier mentioned instructions can be called and parametrized from the `Python` layer, while the core implementation consists of highly optimized `C++` code. This allows one to quickly experiment with different global strategies.

For example, the leftwards progressive sieving technique explained in Section 3.4.2, can be described as follows: start in a small context of say $\mathcal{L}_{[d-40:d)}$ and alternate the Extend Left, Grow and Sieve instructions. The authors of G6K call such an alternation of extending and sieving a *pump up*, up to some final context $\mathcal{L}_{[l:d)}$, and recall that it is much more efficient than running a single Grow and Sieve instruction in the final context $\mathcal{L}_{[l:d)}$. A full *pump* consists of a pump up followed by a *pump down*: repeat the Insertion instruction to improve the basis while making the context smaller again, and optionally combine this with the Sieve instruction to find better insertion candidates. To solve SVP-instances among other things G6K combines such pumps in a *workout*, which is a sequence of longer and longer pumps, until a short enough vector is found in the full context by lifting. Each pump improves the quality of the basis, which as a result lowers the expected length increase from lifting, making consequent pumps faster and simultaneously improving the probability to find a short vector in the full context.

### 3.5.4 Sieve implementations

The current open-source implementation of G6K contains multiple sieving algorithms that implement the Sieve instruction. There are single-threaded implementations of the Nguyen–Vidick (`nv`) [NV08] and Gauss sieve (`gauss`) [MV10], mostly for low dimensions and testing purposes. Furthermore G6K includes a fully multi-threaded and low-level optimized version of the Becker–Gama–Joux (BGJ) sieve with a single bucketing layer (`bgj1`) [BGJ15]. The filtering techniques from `bgj1` were also extended and used in a triple sieve implementation (`hk3`) [BLS16; HK17]. This implementation considers both pairs and triples and its behaviour automatically adjusts based on the database size, allowing for a continuous time-memory trade-off between the (pair) sieve `bgj1` and a full triple sieve with minimal

memory. Note that the asymptotically best sieve algorithm, which we will refer to as BDGL [BDGL16], has been implemented before [MLB17], but not inside of G6K. In Chapter 4 we discuss an optimized implementation of the BDGL sieve inside G6K, which has since been merged into the open-source implementation of G6K.

### 3.5.5 Data representation

Given that lattice sieving uses an exponential number of vectors, it is of practical importance how much data is stored per vector in the database. G6K stores for each lattice vector $\mathbf{v} = \mathbf{Bx} \in \mathbb{R}^n$ the (16-bit integer) coordinates $\mathbf{x} \in \mathbb{Z}^n$ as well as the (32-bit floating-point) Gram-Schmidt representation $\mathbf{y} = (\langle \mathbf{v}, \tilde{\mathbf{b}}_i \rangle / \|\tilde{\mathbf{b}}_i\|)_i \in \mathbb{R}^n$ normalized by the Gaussian Heuristic of the current sieving context. The latter representation is used to quickly compute inner products between any two lattice vectors in the database, and to change between contexts. On top of that other preprocessed information is stored for each vector, like the corresponding lift target $\mathbf{t}$ in $\mathrm{span}(\mathcal{L}_{[\kappa:l)})$, the squared length $\|\mathbf{v}\|^2$, a 256-bit SimHash (see Section 3.4.1) and a 64-bit hash as identifier. In order to sort the database on length, without having to move the entries around, there is also a lightweight database that only stores for each vector the length, a SimHash and the corresponding database index. A hash table keeps track of all hash identifiers, which are derived from the $\mathbf{x}$-coordinates, in order to quickly check for duplicates. All of this adds up to a total of $\approx 2^{10}$ bytes per vector in a sieving dimension of $n = 128$.