

Sieving Algorithms for Exact SVP

Instructor: *Daniele Micciancio*Scribe: *Amy Kanne*

Definition 1 (Shortest Vector Problem (SVP)). *An input to SVP is a lattice basis $\mathbf{B}$, and the goal is to find a lattice vector of length precisely $\lambda(\mathbf{B})$.*

1 Background

Previously in class we investigated approximation algorithms for solving SVP in polynomial time. However, practical algorithms to solve exact SVP are important in many algorithmic and cryptographic applications. For example, earlier in the course we saw how solving SVP in lower dimensions is critical for performing BKZ lattice reduction quickly.

This lecture focuses on *sieving algorithms* as a method of solving SVP with good theoretical guarantees and strong practical performance. In particular because SVP is known to be NP-Hard, we expect that all of these algorithms will take super-polynomial time (unless $P=NP$). In Section 2 we will explore the *List Sieve* algorithm, introduced in [MV10], which guarantees runtime $2^{3.199n}$ and space $2^{1.325n}$, both exponential in the dimension of the input lattice, n . This is in contrast to other methods of solving the exact SVP problem, such as enumeration methods, which take $O(n^n)$ time instead.

In Section 3, we will look at a practical variant of the List Sieve algorithm, *Gauss Sieve*, which will have simplified analysis and guarantee $2^{0.41n}$ space complexity (and at most $2^{0.21n}$ in practice). However, this algorithm is part of a class of sieving algorithms called *heuristic* sieving algorithms, which have no provable guarantees on the runtime of the algorithm. However, these algorithms tend to be rather quick in practice despite their lack of theoretical guarantees, making them a popular topic for practical algorithm development. We will comment on recent developments in heuristic sieving algorithms in Section 3.2.

2 List Sieve

2.1 Overview

The **ListSieve** algorithm is a randomized algorithm which takes as input a lattice basis $\mathbf{B}$ and a parameter μ corresponding to the length of the vector we want to find. If there exists a vector $\mathbf{s}$ with $\|\mathbf{s}\| \leq \mu$, **ListSieve** finds this $\mathbf{s}$ with high probability, and if no such $\mathbf{s}$ exists, the algorithm always outputs $\perp$. For simplicity, throughout this analysis we assume the input lattice is full-rank, though one can generalize these techniques to lower-dimensional lattices.

The algorithm consists of three primary components: the **Sample** step, the **ListReduce** step, and finally the **ListSieve** protocol which combines these subroutines. The algorithm

centers around a list L of lattice points which we will slowly expand over the runtime of the algorithm. In the **Sample** step, we will pick some random lattice vector $\mathbf{v}$ that is approximately as long as the basis vectors. Then, in the **ListReduce** step, we will “reduce” $\mathbf{v}$ using every $\mathbf{u}$ in the list L to make $\mathbf{v}$ as short as possible. This reduction will be of the form $\mathbf{v} \leftarrow \mathbf{v} - \mathbf{u}$ and will critically use the fact that addition and subtraction of lattice vectors are still lattice vectors. Finally, after $\mathbf{v}$ has been sufficiently reduced by all of the elements of L , we add it to our list L so that future vectors will be reduced accordingly.

Over time, L will grow to become very large, and every vector $\mathbf{v}$ add to L will be very short, as every element of L is used to reduce it. We will prove later that the number of vectors in L is upper bounded unless it includes a very short vector $\mathbf{s}$ with $\|\mathbf{s}\| \leq \mu$, at which point **ListSieve** will return this vector as the solution to the shortest vector problem.

In order to show that list L grows often enough to exceed this upper bound, we “perturb” our lattice vectors using short vectors $\mathbf{e}$ before performing reduction through **ListReduce**. We will see later that these perturbations reduce the performance and complicate the analysis of **ListSieve**, but provide our only provable way of bounding the runtime of the algorithm.

2.2 The Algorithm

2.2.1 Sample

The first subroutine **Sample** will select a random error vector $\mathbf{e} \in \mathcal{B}_n(d)$, where $\mathcal{B}_n(d)$ is the n -dimensional ball of radius d , which we will tune later. This $\mathbf{e}$ is the “perturbation” mentioned earlier, and it might be helpful to think of $\mathbf{e}$ as being very short.

Then, using $\mathbf{e}$, it finds a corresponding lattice vector $\mathbf{v}$. In order to find this vector $\mathbf{v}$, we take $\mathbf{e}$ modulo the lattice basis $\mathbf{B}$. To define what $\mathbf{e} \bmod \mathbf{B}$ means, we first introduce the fact that any vector in $\mathbb{R}^n$ can be broken down into the sum of two vectors: one vector in the lattice, and one vector within the parallelepiped of the lattice.

Fact 2. *Let $\mathbf{B}$ be a lattice basis for a lattice Λ and $\mathcal{P} = \{\sum_{i=1}^n x_i \mathbf{b}_i \mid x_i \in [0, 1)\}$ be the parallelepiped defined by $\mathbf{B}$. Then, for any vector $\mathbf{e} \in \mathbb{R}^n$, there exists a unique lattice vector $\mathbf{v} \in \Lambda$ and vector $\mathbf{p} \in \mathcal{P}$ in the parallelepiped such that $\mathbf{v} = \mathbf{p} - \mathbf{e}$. Such a $\mathbf{v}$ can be found in polynomial time.*

Definition 3. *Let $\mathbf{e}, \mathbf{v}, \mathbf{p}$ be as in Fact 2. Then, $\mathbf{e} \bmod \mathbf{B} := \mathbf{p}$.*

Note that we can recover $\mathbf{v}$ from $\mathbf{e}$ and $\mathbf{p}$ by computing $\mathbf{v} = \mathbf{p} - \mathbf{e}$. We use this peculiar sampling method for lattice vectors because it will simplify our performance analysis of the algorithm later. If we let $\|\mathbf{B}\|$ be the length of the longest column vector of $\mathbf{B}$, this sampling method also guarantees that $\|\mathbf{p}\| \leq n\|\mathbf{B}\|$, because $\mathbf{p}$ lies in the parallelepiped of the basis. The pseudocode of the **Sample** subroutine can be found in Figure 1.

The random sampling of the perturbation $\mathbf{e}$ is the only probabilistic part of the algorithm, but it is essential to prove good runtime guarantees for **ListSieve**. As we will see later, we use the **Sample** subroutine to create the vectors $\mathbf{p}$ that we will reduce with **ListReduce**. It may seem odd that we reduce the vectors $\mathbf{p}$, which differ from the lattice by some small

```

function SAMPLE(B,  $d$ )
   $\mathbf{e} \sim \mathcal{B}_n(d)$ 
   $\mathbf{p} \leftarrow \mathbf{e} \bmod \mathbf{B}$ 
  return ( $\mathbf{p}, \mathbf{e}$ )
end function

```

Figure 1: The `Sample` subroutine of `ListSieve`.

perturbation $\mathbf{e}$, instead of using the actual lattice vectors $\mathbf{v} = \mathbf{p} - \mathbf{e}$. We do this primarily because reducing $\mathbf{p}$ guarantees the following useful property: the sphere $\mathcal{B}_n(\mathbf{p}, d)$ includes at least one lattice point $\mathbf{v}$, and the conditional distribution of $\mathbf{v}$ given $\mathbf{p}$ is uniform over all $\mathbf{v}$ in the sphere. That is to say, if $\mathbf{p}$ is fixed, it is equally likely that $\mathbf{p} - \mathbf{e}$ is any one of the various lattice points inside of the sphere $\mathcal{B}_n(\mathbf{p}, d)$.

Why this property is useful may still be unclear at this point (why not just reduce the actual lattice vectors $\mathbf{v} = \mathbf{p} - \mathbf{e}$ instead?), but we will see later that this fact allows us to prove good runtime bounds on `ListSieve`. We will look at a variant of `ListSieve`, called `GaussSieve`, which doesn't use these small perturbations, but although it is rather fast in practice, we do not know any provable time bounds for it.

2.2.2 ListReduce

Next, we will explore the `ListReduce` subroutine which will take our randomly sampled vectors $\mathbf{p}$ and “reduce” them against some list L , creating smaller and smaller lattice vectors. `ListReduce` will take as input the vector $\mathbf{p}$ generated by `Sample`, a list of lattice vectors L , and a parameter δ . The pseudocode for `ListReduce` can be found in Figure 2.

```

function LISTREDUCE( $\mathbf{p}, L, \delta$ )
  while ( $\exists \mathbf{v}_i \in L : \|\mathbf{p} - \mathbf{v}_i\| \leq \delta \|\mathbf{p}\|$ ) do
     $\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$ 
  end while
  return  $\mathbf{p}$ 
end function

```

Figure 2: The `ListReduce` subroutine of `ListSieve`.

The motivating idea for `ListReduce` is to use the vectors in L to make the vectors $\mathbf{p}$ shorter. In particular, we subtract the vectors $\mathbf{v}$ from $\mathbf{p}$ only if $\|\mathbf{p} - \mathbf{v}\| \leq \|\mathbf{p}\|$. One way to visualize this behavior is to let $\mathbf{v}$ be any vector in our list L and consider the “half-plane” of points that are closer to the origin than to $\mathbf{v}$. Figure 3 demonstrates `ListReduce` pushes the vectors $\mathbf{p}$ closer to the origin each time a vector $\mathbf{v} \in L$ is subtracted from it.

One potential problem with this algorithm as written is that reducing $\mathbf{p}$ by some $\mathbf{v}_1 \in L$ may make it no longer fully reduced relative to some other $\mathbf{v}_2 \in L$. So, it is not clear that `ListReduce` will terminate quickly. In order to avoid this issue, we require that $\|\mathbf{p} - \mathbf{v}\| < \delta \|\mathbf{p}\|$ for some fixed δ , just as in the LLL algorithm. This ensures that not only is $\mathbf{p}$

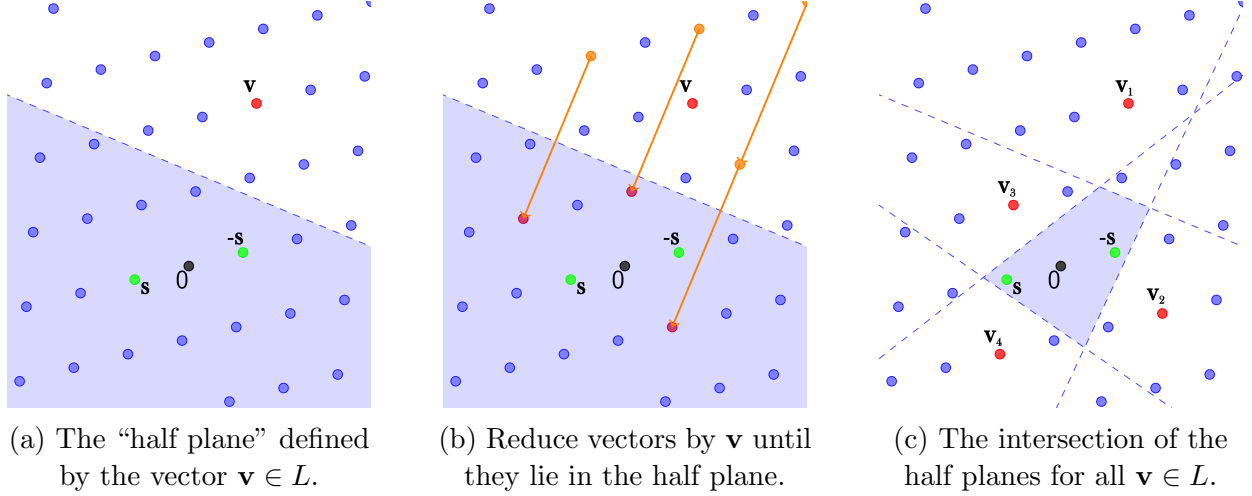


Figure 3: A visualization of the **ListReduce** subroutine. The green points are the vectors $\mathbf{s}$ and $-\mathbf{s}$ with $\|\mathbf{s}\| = \mu$. Each $\mathbf{v} \in L$ defines a half plane, and **ListReduce** will subtract $\mathbf{v}$ from the input $\mathbf{p}$ until it lies in this half plane. Doing this repeatedly for all $\mathbf{v} \in L$ forces $\mathbf{p}$ to lie in the intersection of all of the half planes.

shortening each time we replace $\mathbf{p}$ with $\mathbf{p} - \mathbf{v}$, but that $\mathbf{p}$ is shortening by this constant factor δ . This means that the inside of the loop of **ListReduce** could only execute $\log_{\delta^{-1}} \frac{\|\mathbf{p}\|}{\mu}$ times until in the worst case. Picking $\delta = 1 - \frac{1}{n+1}$ ensures that reduction only happens polynomially many times.

To show this, we will need to assume that the length of the input is polynomial in n , a small loss of generality. Nevertheless, this means that $\log \|\mathbf{B}\|$, the logarithm of the length of the longest column vector in $\mathbf{B}$, is polynomial in n , because it is the same as the number of bits needed to represent this object. Finally, we can see that $\|\mathbf{p}\| \leq n\|\mathbf{B}\|$ because $\mathbf{p}$ lies in the parallelepiped defined by $\mathbf{B}$. Using these two observations and our choice of δ , we can show that **ListReduce** can only loop over L polynomially many times.

$$\log_{\delta^{-1}} \frac{\|\mathbf{p}\|}{\mu} \leq \frac{\log n \|\mathbf{B}\|}{\log \delta^{-1}} \leq \frac{n^{O(1)}}{\log 1 + \frac{1}{n}} = \frac{\log n \|\mathbf{B}\|}{\log \delta^{-1}} \leq \frac{n^{O(1)}}{n^{-1} \log \left(1 + \frac{1}{n}\right)^n} \leq n^{O(1)}$$

Of course, adding this δ coefficient means that our vectors $\mathbf{p}$ may not lie within the half planes depicted in Figure 3 after reduction. Instead, reduction only gets $\mathbf{p}$ “reasonably close” to these half planes. For our purposes, this is enough because $\delta \rightarrow 1$ as $n \rightarrow \infty$. This will be explored in closer detail later during analysis of the algorithm.

2.2.3 ListSieve

Finally, the **ListSieve** algorithm combines both **Sample** and **ListReduce**. The algorithm samples many points using **Sample**, reduces them using **ListReduce**, and then adds them to L if possible. We will show later because L only contains **ListReduced** vectors, L cannot

become too large unless we add a short vector of length $\leq \mu$ to it, at which point we have solved the shortest vector problem. The pseudocode of the algorithm can be found in Algorithm 1. Note that the constants ξ, c will be explained later in our analysis.

Algorithm 1 ListSieve($\mathbf{B}, \mu$)	<i>Output:</i> $\mathbf{v} : \mathbf{v} \in \mathcal{L}(\mathbf{B}) \wedge \ \mathbf{v}\ \leq \mu$ OR: $\perp$
function ListSieve($\mathbf{B}, \mu$) $L \leftarrow \{\mathbf{0}\}, \delta \leftarrow 1 - 1/n, i \leftarrow 0$ $\xi \leftarrow 0.685$ $K \leftarrow 2^{cn}$ while $i < K$ do $i \leftarrow i + 1$ $(\mathbf{p}_i, \mathbf{e}_i) \leftarrow \text{Sample}(\mathbf{B}, \xi\mu)$ $\mathbf{p}_i \leftarrow \text{ListReduce}(\mathbf{p}_i, L, \delta)$ $\mathbf{v}_i \leftarrow \mathbf{p}_i - \mathbf{e}_i$ if $(\mathbf{v}_i \notin L)$ then $\text{if } \exists \mathbf{v}_j \in L : \ \mathbf{v}_i - \mathbf{v}_j\ \leq \mu$ then $\text{return } \mathbf{v}_i - \mathbf{v}_j$ end if $L \leftarrow L \cup \{\mathbf{v}_i\}$ end if end while return $\perp$ end function	function Sample($\mathbf{B}, d$) $\mathbf{e} \sim \mathcal{B}_n(d)$ $\mathbf{p} \leftarrow \mathbf{e} \bmod \mathbf{B}$ return $(\mathbf{p}, \mathbf{e})$ end function function ListReduce($\mathbf{p}, L, \delta$) while $(\exists \mathbf{v}_i \in L : \ \mathbf{p} - \mathbf{v}_i\ \leq \delta \ \mathbf{p}\)$ do $\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$ end while return $\mathbf{p}$ end function

In short, the ListSieve algorithm creates $K = 2^{O(n)}$ sample vectors $\mathbf{v}_i$, perturbs them slightly by some small error $\mathbf{e}_i$ with magnitude at most $\xi\mu$ to create the vectors $\mathbf{p}_i$, reduces them with ListReduce, and then adds the reduced vectors $\mathbf{v}_i$ to L if they are unique. If $\mathbf{v}_i$ can be made into a vector of magnitude $\leq \mu$, return it as the solution. Otherwise, if after enumerating the $2^{O(n)}$ vectors, no such small vector is found, declare that no such vector exists. This algorithm always terminates correctly if $\mathbf{B}$ admits no short vectors with $\|\mathbf{s}\| \leq \mu$, so the only one-sided error is possible. This occurs when $\mathbf{B}$ does have short vectors, but the algorithm fails to find them after enumerating all K samples. Now, we can state the core result of the ListSieve algorithm.

Theorem 4. *Let ξ be a real number greater than 0.5 and $c_1(\xi) = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401$, $c_2 = 0.5 \cdot \log(\xi^2/(\xi^2 - 0.25))$. Then, given any lattice basis $\mathbf{B}$ with target norm μ such that $\lambda(\mathbf{B}) \leq \mu$, ListSieve outputs a lattice vector with norm $\leq \mu$ with high probability in $K = \tilde{O}(2^{(c_1+c_2)n})$ samples, taking $N = \tilde{O}(2^{c_1n})$ space and $\tilde{O}(2^{(2c_1+c_2)n})$ time.*

Note that $\tilde{O}(f(n))$ is used to denote polylogarithmic complexity in $f(n)$. Furthermore, it is a simple optimization exercise to verify that $\xi = 0.685$ minimizes $2c_1 + c_2$, therefore achieving optimal time complexity $\tilde{O}(2^{3.199n})$ and space complexity $\tilde{O}(2^{1.325n})$. Now, all that remains to be shown is that ListSieve will find such a small vector with high probability with this many samples.

2.3 Performance Analysis

It remains to prove runtime and space complexity upper bounds on **ListSieve**. In order to do so, we proceed in three steps.

1. Demonstrate an upper bound on $|L|$ in terms of ξ and n which is only exceeded if L contains a vector with magnitude $\leq \mu$. This will serve as an upper bound on the space complexity of **ListSieve**.
2. Compute a constant c in terms of ξ such that, after $K = 2^{cn}$ iterations, **ListSieve** will exceed the upper bound on the size of L with high probability. Together with the upper bound on $|L|$, this will suffice to prove an upper bound on the time complexity of **ListSieve** of the form $\tilde{O}(K|L|)$, as we perform K iterations of **ListReduce**, which loops over L a polynomial number of times.
3. We will see that $|L|$ will increase as ξ increases, but that higher values of ξ means that we need to do fewer iterations, decreasing K . To resolve this, we pick the optimal value of ξ that minimizes the total running time.

2.3.1 Space Complexity of ListSieve

We will prove the following upper bound N on the number of points in L , in terms of our parameter ξ .

Theorem 5. *The number of points in L is bounded from above by $N = \text{poly}(n) \cdot 2^{c_1 n}$ where*

$$c_1 = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401$$

Before we do so, we need to prove a short lemma that will allow us to rearrange the inequality ensured by **ListReduce**: that $\|\mathbf{p} - \mathbf{v}_i\| > \delta\|\mathbf{p}\|$ for all vectors $\mathbf{v}_i \in L$.

Lemma 6. *For any two vectors $\mathbf{p}, \mathbf{v}$ and real $0 < \delta < 1$, $\|\mathbf{p} - \mathbf{v}\| > \delta\|\mathbf{p}\|$ if and only if $\|(1 - \delta^2)\mathbf{p} - \mathbf{v}\| > \delta\|\mathbf{v}\|$.*

Proof. If we square the first inequality, we get $\|\mathbf{p} - \mathbf{v}\|^2 > \delta^2\|\mathbf{p}\|^2$. Then, using law of cosines, we can expand $\|\mathbf{v} - \mathbf{p}\|^2$ into $\|\mathbf{p}\|^2 + \|\mathbf{v}\|^2 - 2\langle \mathbf{p}, \mathbf{v} \rangle$. Rearranging the inequality gives

$$(1 - \delta^2)\|\mathbf{p}\|^2 + \|\mathbf{v}\|^2 - 2\langle \mathbf{p}, \mathbf{v} \rangle > 0$$

Now, we can multiply both sides by $(1 - \delta^2) > 0$ and rearrange, getting

$$(1 - \delta^2)^2\|\mathbf{p}\|^2 + \|\mathbf{v}\|^2 - 2(1 - \delta^2)\langle \mathbf{p}, \mathbf{v} \rangle > \delta^2\|\mathbf{v}\|^2.$$

One can verify that the left hand side is $\|(1 - \delta^2)\mathbf{p} - \mathbf{v}\|^2$. Taking the square root of both sides then proves the result. $\square$

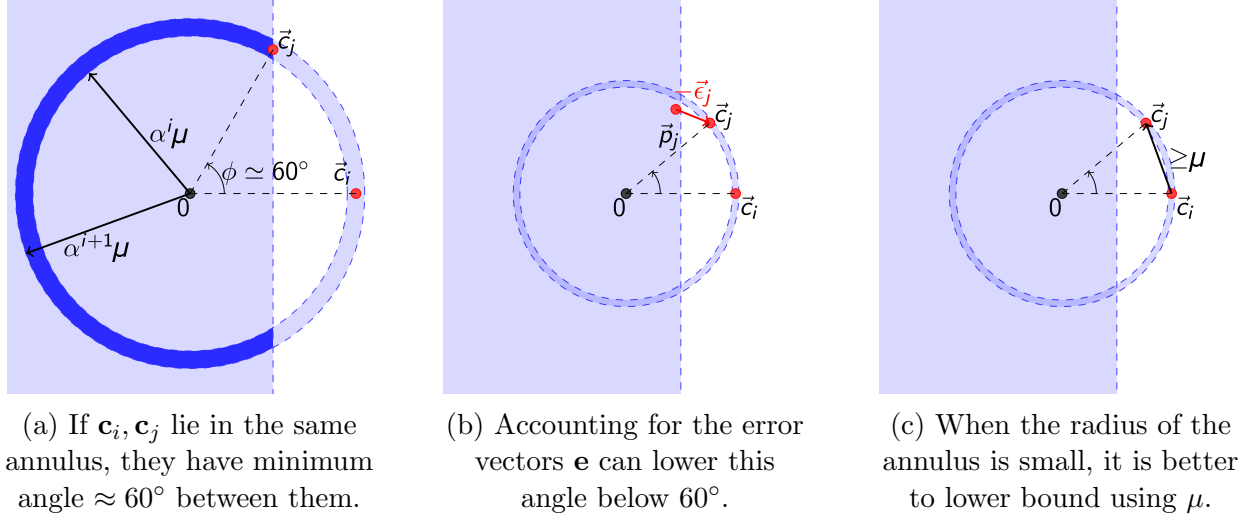


Figure 4: A visualization of how **ListReduce** admits a lower bound on the angle between any two list vectors $\mathbf{c}_i, \mathbf{c}_j \in L$ with similar norm. **ListReduce** forces the vectors in L to lie near each others' half planes, so if two vectors have similar norm, we can lower bound the angle between them.

There is one more preliminary result we need before we can prove Theorem 5. Because **ListReduce** “pushes” the vectors $\mathbf{p}$ into (or very close to) each of the half planes for every $\mathbf{v} \in L$, we can prove a lower bound on the angle between any two vectors of similar norm. Figure 4 gives a visual explanation for where this lower bound comes from.

This observation motivates our proof technique for upper bounding the number of points in L . First, we break up the plane into many annuli of inner radius $\alpha^i \mu$ and outer radius $\alpha^{i+1} \mu$ for some $\alpha > 1$ (to be determined). Then, because any list element $\mathbf{v}_j$ must lie close to the half plane defined by $\mathbf{v}_i$, we know that the angle between them must be at least ϕ , for some $\phi \approx 60^\circ$. However, as we can see by Figure 4, because **ListReduce** pushes the $\mathbf{p}$ into each others' half planes, not the actual list elements $\mathbf{v}_i$, we need to subtract the small error vectors $\mathbf{e}$ to reach the actual list elements $\mathbf{v}_j$. This compounded with the fact that the $\mathbf{p}_i$ themselves don't necessarily always lie in each others' half planes (due to the factor of δ used during **ListReduce**) means that the minimum angle between any two list elements in an annulus is quite a bit lower than 60° , which makes our upper bound on the size of L worse.

Furthermore, when the radius of the annulus gets quite small, the length of the error vectors will even more drastically reduce the minimum angle between list elements. So, in these cases, we will instead lower bound the angle between list elements using the fact that list elements must differ by at least μ , or else **ListSieve** would have already terminated. With these two lower bounds, we will be able to prove that there is an angle $\phi_{\text{list}} < 60$ which lower bounds the angle between any two list elements in the same annulus. Then, we can use a result from Kabatiansky and Levenshtein which comes from linear programming bounds for spherical codes to give a concrete bound on the number of possible list elements in each

annulus.

Theorem 7 (Kabatiansky and Levenshtein [KL78]). *Let $A(n, \phi)$ be the maximal size of any set C of points in $\mathbb{R}^n$ such that the angle between any two distinct vectors $\mathbf{v}_i, \mathbf{v}_j \in C$ (denoted $\phi_{\mathbf{v}_i, \mathbf{v}_j}$) is at least ϕ_0 . If $0 < \phi < 63^\circ$, then for all sufficiently large n , $A = 2^{cn}$ for some*

$$c \leq -\frac{1}{2} \log(1 - \cos(\phi)) - 0.099.$$

Now, we have all the necessary machinery to prove Theorem 5.

Proof of Theorem 5. As mentioned earlier, we will split the plane into a series of annuli S_i . The first annulus is $S_0 = L \cap \mathcal{B}(\mu/2)$, the ball around the origin of radius $\mu/2$. We can assume that all elements of L are of distance greater than μ from each other, as otherwise the algorithm would have already terminated. Therefore, S_0 contains at most one element, because the distance between any two points of $\mathcal{B}(\mu/2)$ is bounded above by μ .

Then, we let $\gamma = 1 + 1/n$ and define the rest of the annuli

$$S_i = \{\mathbf{x} \mid \gamma^{i-1}\mu/2 \leq \|\mathbf{x}\| < \gamma^i\mu/2\} \cap L.$$

Similarly to before during our analysis of the number of iterations of **ListReduce**, because all of the vectors $\mathbf{v} \in L$ are of length at most $n\|\mathbf{B}\|$, there are only $\log_\gamma(2n\|\mathbf{B}\|/\mu)$ annuli we must consider, and using the assumption that $\log \|\mathbf{B}\|$ is polynomial in n , we see that there are only polynomially many such annuli.

Now, we need to prove that each annulus contains at most an exponentially many points. This will suffice to show that L contains at most exponentially many points (with the same constant in the exponent), as there are polynomially many annuli. To show this, we will critically use the observation from Figure 4. In particular, we will show that for any annulus S_i and any two points $\mathbf{v}_1, \mathbf{v}_2 \in S_i$, that we can upper bound the cosine of $\phi_{\mathbf{v}_1, \mathbf{v}_2}$, the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$.

$$\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) \leq 1 - \frac{1}{2(\xi + \sqrt{\xi^2 + 1})^2} + o(1) \quad (1)$$

We will prove this equation later. First, we will show how (1) proves the theorem. We can observe that, because $(\xi + \sqrt{\xi^2 + 1})^2 > 1$, this bound on $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$ is greater than $1/2$, so $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) < 60^\circ$. Then, we can apply [KL78] to show that, for large enough n , the number of points in S_i is at most $2^{c_1 n}$, where

$$\begin{aligned} c_1 &\leq -\frac{1}{2} \log(1 - \cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})) - 0.099 \\ &\leq \log\left(\sqrt{2}(\xi + \sqrt{\xi^2 + 1})\right) - 0.099 \\ &= \log(\xi + \sqrt{\xi^2 + 1}) + 0.401. \end{aligned}$$

This suffices to prove the theorem. Now, we need to prove (1). To do this, note that we can use the law of cosines to express $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$ in terms of $\|\mathbf{v}_1\|$, $\|\mathbf{v}_2\|$, and $\|\mathbf{v}_1 - \mathbf{v}_2\|$.

$$\begin{aligned}\|\mathbf{v}_1 - \mathbf{v}_2\|^2 &= \|\mathbf{v}_1\|^2 + \|\mathbf{v}_2\|^2 - 2\|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\| \cdot \cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) \\ \cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) &= \frac{1}{2} \left(\frac{\|\mathbf{v}_1\|}{\|\mathbf{v}_2\|} + \frac{\|\mathbf{v}_2\|}{\|\mathbf{v}_1\|} - \frac{\|\mathbf{v}_1 - \mathbf{v}_2\|^2}{\|\mathbf{v}_2\| \cdot \|\mathbf{v}_1\|} \right)\end{aligned}$$

As suggested by Figure 4, we will bound this latter equation in two ways. Let $R = \gamma^{i-1}\mu/2$ be the radius of the inner annulus S_i , so $\|\mathbf{v}_1\|, \|\mathbf{v}_2\| \in [R, \gamma R)$. When R is small, we will use the fact that $\|\mathbf{v}_1 - \mathbf{v}_2\| > \mu$ to upper bound $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$. This corresponds to part (c) of Figure 4. When R is large, this bound will become quite weak, so we will use the fact that $\mathbf{v}_1, \mathbf{v}_2$ have been reduced by **ListReduce** to bound their angle. This corresponds to part (b) of Figure 4. At the end, we will optimize which of these bounds we use based on the radius R to show (1).

The first bound using $\|\mathbf{v}_1 - \mathbf{v}_2\| > \mu$ is straightforward. We can see that because $\mathbf{v}_1, \mathbf{v}_2 \in S_i$ that $\|\mathbf{v}_1\|/\|\mathbf{v}_2\| \leq \gamma$, and the same is true for $\|\mathbf{v}_2\|/\|\mathbf{v}_1\|$. So,

$$\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) \leq \frac{1}{2} \left(\frac{\|\mathbf{v}_1\|}{\|\mathbf{v}_2\|} + \frac{\|\mathbf{v}_2\|}{\|\mathbf{v}_1\|} - \frac{\mu^2}{\|\mathbf{v}_2\| \cdot \|\mathbf{v}_1\|} \right) \leq \gamma - \frac{\mu^2}{2R^2} = 1 - \frac{\mu^2}{2R^2} + o(1)$$

Now, we can see directly that this bound is poor when R is very large. In the case of large R , we use the fact that one of $\mathbf{v}_1, \mathbf{v}_2$ is the product of some $\mathbf{p}$ which was **ListReduced**. Say that $\mathbf{v}_2$ was added after $\mathbf{v}_1$, so if we define $\mathbf{p}_2 = \mathbf{v}_2 + \mathbf{e}_2$, we know that $\mathbf{p}_2$ is reduced with respect to $\mathbf{v}_1$. So, $\|\mathbf{p}_2 - \mathbf{v}_1\| > \delta\|\mathbf{p}_2\|$. Furthermore, we can observe that $\phi_{\mathbf{v}_1, \mathbf{v}_2} = \phi_{\mathbf{v}_1, (1-\delta^2)\mathbf{v}_2}$. Now, we can use the triangle inequality and Lemma 6 to lower bound $\|(1-\delta^2)\mathbf{v}_2 - \mathbf{v}_1\|$, which will correlated to a lower bound on $\phi_{\mathbf{v}_1, (1-\delta^2)\mathbf{v}_2}$.

$$\|(1-\delta^2)\mathbf{v}_2 - \mathbf{v}_1\| \geq \|(1-\delta^2)\mathbf{p}_2 - \mathbf{v}_1\| - (1-\delta^2)\|\mathbf{e}_2\| > \delta\|\mathbf{v}_1\| - (1-\delta^2)\xi\mu \quad (2)$$

Now that we've lower bounded the distance between $(1-\delta^2)\mathbf{v}_2$ and $\mathbf{v}_1$, we can plug this into our earlier equation for $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$ to get our second upper bound on the cosine of $\phi_{\mathbf{v}_1, \mathbf{v}_2}$.

$$\begin{aligned}\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) &= \cos(\phi_{\mathbf{v}_1, (1-\delta^2)\mathbf{v}_2}) \\ &= \frac{1}{2} \left(\frac{\|\mathbf{v}_1\|}{(1-\delta^2)\|\mathbf{v}_2\|} + \frac{(1-\delta^2)\|\mathbf{v}_2\|}{\|\mathbf{v}_1\|} - \frac{\|\mathbf{v}_1 - (1-\delta^2)\mathbf{v}_2\|^2}{(1-\delta^2)\|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|} \right)\end{aligned}$$

Now we can use (2), expand, and simplify.

$$\begin{aligned}&\leq \frac{1}{2} \left(\frac{\|\mathbf{v}_1\|}{(1-\delta^2)\|\mathbf{v}_2\|} + \frac{(1-\delta^2)\|\mathbf{v}_2\|}{\|\mathbf{v}_1\|} - \frac{(\delta\|\mathbf{v}_1\| - (1-\delta^2)\xi\mu)^2}{(1-\delta^2)\|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|} \right) \\ &= \frac{\|\mathbf{v}_1\|^2 + (1-\delta^2)^2\|\mathbf{v}_2\|^2 - \delta^2\|\mathbf{v}_1\|^2 + 2\delta(1-\delta^2)\|\mathbf{v}_1\|\xi\mu - ((1-\delta^2)\xi\mu)^2}{2(1-\delta^2)\|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|} \\ &= \frac{\|\mathbf{v}_1\|}{2\|\mathbf{v}_2\|} + \frac{\xi\mu\delta}{\|\mathbf{v}_2\|} + (1-\delta^2) \frac{\|\mathbf{v}_2\|^2 - (\xi\mu)^2}{2\|\mathbf{v}_1\| \cdot \|\mathbf{v}_2\|}\end{aligned}$$

Finally, we can use the fact that $\gamma = 1 + 1/n$ and $\delta = 1 - 1/n$ to group lower-order terms.

$$\leq \frac{1}{2} + \frac{\xi\mu}{R} + o(1)$$

Now, we have proven the two bounds on $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$ we sought to show: one when R is small, and one when R is large. Therefore, we know that

$$\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2}) \leq \min \left\{ 1 - \frac{\mu^2}{2R^2}, \frac{1}{2} + \frac{\xi\mu}{R} \right\} + o(1). \quad (3)$$

The first of these bounds gets worse as R increases, and the second of these bounds gets better. So, to have the best upper bound on $\cos(\phi_{\mathbf{v}_1, \mathbf{v}_2})$, we want these two quantities to be equal.

$$1 - \frac{\mu^2}{2R^2} = \frac{1}{2} + \frac{\xi\mu}{R}$$

Solving this as a quadratic in μ/R gives

$$\frac{\mu}{R} = \sqrt{1 + \xi^2} - \xi$$

which, when substituted in (3), gives the bound (1). $\square$

We will optimize ξ later, but once we do so, Theorem 5 will immediately give us the maximum size of L and therefore the space complexity of **ListSieve**. Now, all that remains to argue is the time complexity of **ListSieve**.

2.3.2 Time Complexity

As a reminder, **ListSieve** samples random perturbations $\mathbf{e}_i$ with small magnitude $\|\mathbf{e}_i\| \leq \xi\mu$, assigns $\mathbf{p}_i = \mathbf{e}_i \bmod \mathbf{B}$, performs **ListReduce** to reduce $\mathbf{p}_i$ using L , then considers the lattice vector $\mathbf{v}_i = \mathbf{p}_i - \mathbf{e}_i$. From there, **ListSieve** either adds $\mathbf{v}_i$ to L , skips adding $\mathbf{v}_i$ because $\mathbf{v}_i$ is already in L , or terminates because $\mathbf{v}_i$ is within μ distance of an element of L . We label each of these possibilities as follows.

1. Event **L**: $\mathbf{v}_i$ is a new list point ($\text{dist}(L, \mathbf{v}_i) > \mu$)
2. Event **C**: $\mathbf{v}_i$ is a collision ($\text{dist}(L, \mathbf{v}_i) = 0$)
3. Event **S**: $\mathbf{v}_i$ is a solution ($\text{dist}(L, \mathbf{v}_i) \in (0, \mu]$).

Moreover, **ListSieve** performs this loop K times. This will take a total of $K \cdot |L|$ time (ignoring polynomial and other subexponential factors), as **ListReduce** takes $O(L) \cdot n^{O(1)}$ time. Because the choice of the error vectors $\mathbf{e}_i$ is random, we aim to pick some $K = 2^{cn}$ large enough such that event **S** occurs with high probability. Furthermore, we know from Theorem 5 that $|L| \leq N$, so event **L** can only occur N times at most. So, it suffices to show that it is unlikely that we have too many **C** events.

In order to bound the probability of having **C** events, we will critically use the perturbations $\mathbf{e}_i$. So, although the $\mathbf{e}_i$ increase the space complexity of **ListSieve**, as seen in Figure 4, they are useful for bounding the running time of the algorithm. However, we'll see later a practical variant **GaussSieve** which does not use perturbations. Although it seems to run quickly in practice, we do not know any provable bounds on its running time. In this sense, the $\mathbf{e}_i$ seem useful for *proving* good runtime bounds, but not for creating fast algorithms. We will discuss this distinction more in Section 3.2.

Nevertheless, for **ListSieve**, we will upper bound the number of **C** events by showing that if $\mathbf{e}_i$ lie in a certain region of $\mathcal{B}(\xi\mu)$, then it is likely that the event **C** does not occur. Say that $\mathbf{s}$ is a lattice vector with length $\leq \mu$. We will look at the probability that $\mathbf{e}_i$ lies in $\mathcal{B}(\mathbf{s}, \xi\mu) \cap \mathcal{B}(\mathbf{0}, \xi\mu)$. This region is the intersection of two hyperspheres with radius $\xi\mu$, so it can be viewed as the disjoint union of two hypersphere caps. We use the following simple bound on ratio between the volume of such a cap and the volume of a hypersphere.

Lemma 8. *Let $\text{Cap}_{R,h}$ be the n -dimensional cap with height h of a hypersphere $\mathcal{B}(R)$. Then, if $R_b = \sqrt{2hR - h^2}$ is the radius of the cross section at this height h , then*

$$\frac{\text{vol}(\text{Cap}_{h,R})}{\text{vol}(\mathcal{B}(R))} > \left(\frac{R_b}{R}\right)^n \cdot \frac{h}{2R_b n}.$$

This can be easily verified by noticing that a hypersphere cap of height h is contained in a cylinder of radius R_b and height h . We omit the proof for brevity. There is one more useful fact that we will need for our later analysis: a simple application of the Chernoff bound for binomial distributions. We use Chernoff's inequality to show that the probability that we have many **C** events is very low.

Theorem 9 (Chernoff's Inequality). *Let $X \sim \text{Binomial}(K, p)$ be a binomially distributed random variable with mean $\mu = Kp$. Then $P(X \leq \mu/2) \leq e^{-\mu/8}$.*

Now, we can proceed to prove an upper bound on the number of samples it takes for **ListSieve** to find a suitably short lattice point (with high probability).

Theorem 10. *If there exists a lattice point $\mathbf{s}$ with $\|\mathbf{s}\| \leq \mu$, then **ListSieve** will output a lattice point with norm $\leq \mu$ with high probability after using $K = \tilde{O}(2^{(c_1+c_2)n})$ samples, where*

$$c_1 = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401 \quad c_2 = \log\left(\frac{\xi}{\sqrt{\xi^2 - 0.25}}\right)$$

Note that c_1, c_2 are the same as Theorem 5 and Theorem 4.

Proof. A quick high-level overview. First, we use Lemma 8 to show that, on any given iteration, the probability that it is a **C** event is not too high. Then, we use the fact that these samples are independent to bound our desired probability with that of the binomial distribution. Applying Chernoff's inequality then completes the argument.

Consider the hypersphere intersection previously discussed: $\mathcal{B}(\mathbf{s}, \xi\mu) \cap \mathcal{B}(\mathbf{0}, \xi\mu)$. Call this region I , and define $I' = I - \mathbf{s} = \mathcal{B}(\mathbf{0}, \xi\mu) \cap \mathcal{B}(-\mathbf{s}, \xi\mu)$. Note that both of these are subsets of the ball $\mathcal{B}(\mathbf{0}, \xi\mu)$ in which all of the $\mathbf{e}_i$ lie, and that they are disjoint (they lie on opposite sides of this ball and do not overlap as $\xi < 1$). Moreover, we can see that for any $\mathbf{e}_i \in I$, then $\mathbf{e}'_i - \mathbf{s} \in I'$, and vice-versa. Therefore, the perturbations $\mathbf{e}_i$ and $\mathbf{e}_i - \mathbf{s}$ are equally likely.

Critically, we can see that **Sample** gives the same $\mathbf{p}_i$ for both $\mathbf{e}_i$ and $\mathbf{e}'_i$. Here we use the fact that we sample the $\mathbf{p}_i$ using the $\bmod \mathbf{B}$ operation, and that any lattice point is $\mathbf{0} \bmod \mathbf{B}$.

$$\mathbf{p}'_i = \mathbf{e}'_i \bmod \mathbf{B} = \mathbf{e}_i + \mathbf{s} \bmod \mathbf{B} = \mathbf{e}_i \bmod \mathbf{B} = \mathbf{p}_i$$

This motivates our use of $\bmod \mathbf{B}$ as our method of sampling lattice vectors. This means that, given a fixed $\mathbf{p}_i$, **ListReduce** is oblivious to whether $\mathbf{p}_i$ was chosen using $\mathbf{e}_i$ or $\mathbf{e}'_i$. Let $\mathbf{v}_i = \mathbf{p}_i - \mathbf{e}_i$ and $\mathbf{v}'_i = \mathbf{p}_i - \mathbf{e}'_i$ be the corresponding lattice vectors for the two perturbations. Then, because the distance between $\mathbf{v}_i$ and $\mathbf{v}'_i$ is $\|\mathbf{s}\| \leq \mu$, they cannot both be in L . Therefore, we can organize the perturbations of $I \cup I'$ into pairs, where at least one of each pair does not give a collision. Because all perturbations are equally likely, this means that the probability of not $\mathbf{C}$ given $\mathbf{e}_i \in I \cup I'$ is $P[\neg \mathbf{C} \mid \mathbf{e}_i \in I \cup I'] \geq 0.5$.

From here, it is a routine conditional probability computation to compute a bound on $P[\neq \mathbf{C}]$. First, we can use Lemma 8 to lower bound the probability that $\mathbf{e}_i$ lies in $I \cup I'$, using the fact that $I \cup I'$ is made up of four hypersphere caps of height $\xi\mu - 0.5\mu$.

$$P[\mathbf{e}_i \in I \cup I'] = \frac{4\text{vol}(\text{Cap}_{\xi\mu-0.5\mu, \xi\mu})}{\text{vol}(\mathcal{B}(\xi\mu))} > \left(\frac{\sqrt{\xi^2 - 0.25}}{\xi} \right)^n \cdot \Theta(1/n) = 2^{-c_2 n} \cdot \Theta(1/n)$$

Then, using our earlier result that $P[\neg \mathbf{C} \mid \mathbf{e}_i \in I \cup I'] \geq 0.5$, we can conclude that

$$P[\neg \mathbf{C}] \geq P[\neg \mathbf{C} \mid \mathbf{e}_i \in I \cup I'] \cdot P[\mathbf{e}_i \in I \cup I'] \geq 2^{-c_2 n} \cdot \Theta(1/n) = p.$$

This tells us that $\mathbf{C}$ is “not too likely.” We have a lower bound $p = \tilde{O}(2^{-c_2 n})$ on the probability that $\mathbf{C}$ does not occur on any given sample. Therefore, if we take K samples, the number of occurrences of $\neg \mathbf{C}$ is at least the number of occurrences of a binomially distributed random variable $X \sim \text{Binomial}(K, p)$. Then, the probability that $X \leq N$ upper bounds the probability that there are fewer than N occurrences of $\neg \mathbf{C}$.

If we choose $K = 2Np^{-1} = \tilde{O}(2^{(c_1+c_2)n})$ so that $\mu = Kp = 2N$, we can then use Theorem 9 to see that

$$P[X \leq N] \leq e^{-N/4} \leq 2^{-O(n)}.$$

Therefore, there are at least $N + 1$ occurrences of $\neg \mathbf{C}$ with high probability. These events are either **L** or **S** events, but there can be at most N **L** events by Theorem 5. So an **S** event will occur with high probability, and **ListSieve** will terminate successfully. $\square$

Corollary 11. *The total running time of the algorithm is $\tilde{O}(KN) = \tilde{O}(2^{(2c_1+c_2)n})$, where*

$$c_1 = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401 \quad c_2 = \log \left(\frac{\xi}{\sqrt{\xi^2 - 0.25}} \right).$$

Now, all that remains is to choose ξ appropriately so that the constant in the exponent of the running time, $2c_1 + c_2$, is minimized. Numerical analysis indicates that this occurs when $\xi \approx 0.685$. This gives space complexity $\tilde{O}(2^{c_1 n}) = \tilde{O}(2^{1.325n})$ and time complexity $\tilde{O}(2^{(2c_1+c_2)n}) = \tilde{O}(2^{3.199n})$, proving Theorem 4.

If instead we aim to optimize the space complexity of **ListSieve**, we can choose ξ very near 0.5 so that c_1 , the constant defining N , is minimized. However, as ξ gets nearer and nearer to 0.5, the volume of the intersections of the hyperspheres gets very slim, so the constant in the exponent of K explodes.

3 Gauss Sieve

Now we introduce **GaussSieve**, a practical variant of **ListSieve** that gives up its provable runtime guarantees in return for drastically improved space complexity and fast runtime in practice. This is one of many algorithms in a class of exact SVP algorithms called “heuristic sieving algorithms,” which we will comment on later.

3.1 The GaussSieve Algorithm

Algorithm 2 GaussSieve ($\mathbf{B}, \mu$)	<i>Output:</i> $\mathbf{v} : \mathbf{v} \in \mathcal{L}(\mathbf{B}) \wedge \ \mathbf{v}\ \leq \mu$
function GAUSSSIEVE ($\mathbf{B}, \mu$) $L \leftarrow \{\mathbf{0}\}, S \leftarrow \{\}, i \leftarrow 0$ while $i < K$ do if S is not empty then $\mathbf{v}_{new} \leftarrow S.\text{pop}()$ else $\mathbf{v}_{new} \leftarrow \text{SampleGaussian}(\mathbf{B})$ end if $\mathbf{v}_{new} \leftarrow \text{GaussReduce}(\mathbf{v}_{new}, L, S)$ if $\mathbf{v}_{new} = \mathbf{0}$ then $i \leftarrow i + 1$ else $L \leftarrow L \cup \{\mathbf{v}_{new}\}$ end if end while end function	function GAUSSREDUCE ($\mathbf{p}, L, S$) while $(\exists \mathbf{v}_i \in L : \ \mathbf{v}_i\ \leq \ \mathbf{p}\ $ $\wedge \ \mathbf{p} - \mathbf{v}_i\ \leq \ \mathbf{p}\)$ do $\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$ end while while $(\exists \mathbf{v}_i \in L : \ \mathbf{v}_i\ > \ \mathbf{p}\ $ $\wedge \ \mathbf{v}_i - \mathbf{p}\ \leq \ \mathbf{v}_i\)$ do $L \leftarrow L \setminus \{\mathbf{v}_i\}$ $S.\text{push}(\mathbf{v}_i - \mathbf{p})$ end while return $\mathbf{p}$ end function

GaussSieve is a variant of **ListSieve** which sacrifices provable runtime bounds for better space performance. To accomplish this, it differs from **ListSieve** in a handful of ways.

1. There are no perturbations $\mathbf{e}_i$ and therefore no perturbed lattice vectors $\mathbf{p}_i$. In particular, the vector $\mathbf{p}$ that appears as input to **GaussReduce** is a (non-perturbed) lattice

vector. This change also means that we can sample our lattice vectors without using $\bmod \mathbf{B}$; any sampling method will do as we don't need to prove anything about running time.

2. **GaussReduce** fully reduces the vector $\mathbf{p}$ using the list vectors $\mathbf{v}_i$. That is, there is no δ factor in the inequality $\|\mathbf{p} - \mathbf{v}_i\| \leq \|\mathbf{p}\|$.
3. The termination parameter K no longer counts the total number of samples, but the total number of collisions before the algorithm halts.
4. **GaussReduce** not only reduces $\mathbf{p}$ using the list vectors $\mathbf{v}_i$, but also reduces the list vectors relative to $\mathbf{p}$ by adding any list vectors which are not reduced relative to $\mathbf{p}$ to the set S , which then go through the same reduction process on future iterations. The exact purpose of the set S will be clarified shortly.

We make changes (1), (2), and (3) because we are no longer interested in proving runtime bounds. The choices of using perturbations, the δ factor in reduction, and using K to count the total number of samples were all made primarily to assist in proving the runtime bounds of Theorem 10. Changes (1) and (2), which remove the δ factor and perturbations, can be thought of as a more “ala carte” form of reduction that more easily matches up with the intuitions presented in Figures 3 and 4.

In fact, it might be helpful now to think of **GaussSieve** as a more “honest” implementation of the list reduction strategies present in Figures 3 and 4, while **ListSieve** makes concessions in both space complexity and elegance so that we can prove Theorem 10. It is possible the **GaussSieve** has good provable runtime complexity as well—we just do not know any way of proving such claims without using perturbations to upper bound the probability of collision on a given iteration. Without the $\mathbf{e}_i$ perturbations, we do not know how often collisions occur, so we don't have any good strategy of bounding the runtime of **GaussSieve**. In fact, because **GaussReduce** sometimes *removes* elements from the list L , it could even be the case that we have nonterminating executions which repeatedly add and remove elements from the list. Empirical tests indicate, however, that this does not happen in practice.

The next modification (3), is one entirely of convenience as we no longer need to prove claims about runtime—in practice, it seems that terminating based off of the number of collisions is a good heuristic. There are other practical advantages to **GaussSieve** as well. In particular, because all operations happen between lattice vectors (with no error terms), there is an integer implementation of the algorithm. Furthermore, we can use whatever sampling method we like for the new lattice vectors $\mathbf{v}_{new}$, because we don't need to prove any runtime bounds.

Lastly, the modification (4) ensures that all lattice vectors in our list L are pairwise reduced. That is, for any two list vectors $\mathbf{v}, \mathbf{u} \in L$, if $\min(\|\mathbf{v} \pm \mathbf{u}\|) < \max(\|\mathbf{u}\|, \|\mathbf{v}\|)$, we replace the longer of $\mathbf{v}, \mathbf{u}$ with the shorter of $\mathbf{v} \pm \mathbf{u}$. This is achieved by adding current list vectors $\mathbf{v}_i$ into a “dummy set” S from which they are re-reduced using **GaussReduce**. As mentioned earlier, this has the potential to create non-terminating loops, but this behavior doesn't occur in practice. Nevertheless, this ensures the condition that for any two vectors

in our list, $\min(\|\mathbf{v} \pm \mathbf{u}\|) \geq \max(\|\mathbf{u}\|, \|\mathbf{v}\|)$. This condition is, in fact, the definition of a “Gauss reduced basis” which gives **GaussSieve** its name.

One can prove (using methods similar to the intuition of Figure 4) that this condition ensures that any two vectors that lie in L have angle at least 60 degrees. The number of points one can fit in n dimensional space with angle at least 60 between them is called the “kissing number” τ_n , which comes from the fact that 60° is the minimal angle between the centers of two nonintersecting equal spheres that touch (“kiss”) a third sphere of the same radius. Bounding the kissing number is a well-studied problem, and the best known upper bound on τ_n is $2^{(0.401+o(1))n}$ from [CS13], so we can upper bound the space complexity of **GaussSieve** by $\tilde{O}(2^{0.402n})$. However, in practice, **GaussSieve** tends to have a list of length $\tilde{O}(2^{0.21n})$ and runtime $\approx 2^{0.42n}$. Both of these are much better than **ListSieve** which can have a much larger list of length $\tilde{O}(2^{1.325n})$ and runtime $\tilde{O}(2^{3.199n})$ due to the perturbations shrinking this minimal angle between vectors. Of course, only **ListSieve** has provable runtime bounds.

Citation	c_{space}	c_{time}	Notes
[NV08]	0.2075	0.4150	First heuristic sieving algorithm.
[MV10]	0.2075	0.4150	Introduces GaussSieve . Large practical speedup over [NV08].
[WLTB11]	0.2557	0.3836	Variant of [NV08] with alternative tradeoff between space and time.
[ZPH13]	0.2833	0.3778	Explores different space/time tradeoff.
[BGJ14]	0.2925	0.3774	
[Laa15]	0.2075	0.3366	Uses nearest neighbor search techniques to improve both sides of space/time tradeoff.
[AB15]	0.2075	0.3112	Improves on [Laa15].
[LdW15]	0.2075	0.298	Improves on [AB15].
[BL15]	0.2075	0.298	Faster than [LdW15] in practice.
[BDGL16]	0.2075	0.292	Best known heuristic runtime.
[BLS16]	0.1887	0.4812	Uses tuple sieving to lower space complexity at the cost of runtime.
[HK17]	0.1887	0.3717	First algorithm to beat GaussSieve in both space and time.
[HKL17]	0.1887	0.3588	Improves runtime of [HK17].

Table 1: A list of recent heuristic sieving algorithms and their associated heuristic complexity. c_{space} refers to the constant term in the exponent of the space complexity, and c_{time} is the same for (heuristic) time complexity.

3.2 Other Heuristic Sieving Algorithms

Although **ListSieve** has strong provable runtime and space bounds, **GaussSieve** tends to run very fast in practice with much lower space requirements, making it a strong contender for practical applications. Until **GaussSieve**’s publication, most practical settings used enumeration style algorithms despite sieving algorithms having better asymptotic runtime ($2^{O(n)}$ instead of $O(n^n)$). Since then, there has been an explosion of research in so-called “heuristic sieving algorithms” based on **GaussSieve**.

These algorithms are called “heuristic” because they do not have any known proofs on their running time except under certain spurious “heuristic” assumptions, usually concerning the runtime behavior of the algorithms. That is to say, these algorithms’ runtimes are not “proven” in the usual sense—it would be more accurate to say that they are *hypothesized* to have the listed asymptotic runtimes under some “reasonable” algorithm-specific assumptions. Table 1 summarizes these heuristic sieving algorithms.

There are a handful of other developments in heuristic sieving algorithms that don’t fit quite as nicely into Table 1. [LM18] and [Duc17] are two recent results which show how solving a small number of lower-dimensional instances of SVP suffices for solving SVP in higher dimensions. These results only give sub-exponential speedup over the algorithms in Table 1, so c_{space} and c_{time} are unaffected, but these improvements have led to significantly faster implementations in practice. In particular, [ADH⁺19] combined these two speedups to create a sort of “meta” sieving algorithm which has very fast practical performance. In fact, this particular algorithm currently holds the top 4 spots on the exact SVP challenge hall-of-fame, where it has solved SVP problems in dimension 180 ([link](#)).

References

- [AB15] Antoine Joux Anja Becker, Nicolas Gama. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522, 2015.
- [ADH⁺19] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. Cryptology ePrint Archive, Report 2019/089, 2019.
- [BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 10–24. SIAM, 2016.
- [BGJ14] Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. *LMS Journal of Computation and Mathematics*, 17(A):49–70, 2014.
- [BL15] Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope lsh. Cryptology ePrint Archive, Report 2015/823, 2015.

- [BLS16] Shi Bai, Thijs Laarhoven, and Damien Stehle. Tuple lattice sieving. Cryptology ePrint Archive, Report 2016/713, 2016.
- [CS13] John Horton Conway and Neil James Alexander Sloane. *Sphere packings, lattices and groups*, volume 290. Springer Science & Business Media, 2013.
- [Duc17] Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. Cryptology ePrint Archive, Report 2017/999, 2017.
- [HK17] Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In Serge Fehr, editor, *Public-Key Cryptography – PKC 2017*, pages 16–40, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
- [HKL17] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. Cryptology ePrint Archive, Report 2017/1228, 2017.
- [KL78] Grigorii Anatol’evich Kabatiansky and Vladimir Iosifovich Levenshtein. On bounds for packings on a sphere and in space. *Problemy Peredachi Informatsii*, 14(1):3–25, 1978.
- [Laa15] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.
- [LdW15] Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 101–118, Cham, 2015. Springer International Publishing.
- [LM18] Thijs Laarhoven and Artur Mariano. Progressive lattice sieving. Cryptology ePrint Archive, Report 2018/079, 2018.
- [MV10] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1468–1480. SIAM, 2010.
- [NV08] Phong Q Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.
- [WLTB11] Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved nguyen-vidick heuristic sieve algorithm for shortest vector problem. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 1–9, 2011.

- [ZPH13] Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In *International Conference on Selected Areas in Cryptography*, pages 29–47. Springer, 2013.