Sieving algorithm for the shortest vector problem

October 11, 2021

Research question: How to practically implement a sieving algorithm for SVP

Word count: 3989

Subject: Mathematics

Contents

1 Introduction

Modern-day RSA encryption is based on the hardness of prime factorization. However, quantum computers pose a risk to this type of encryption due to Schor's algorithm, a polynomial time algorithm for solving prime factorization[1]. Therefore, there is currently a lot of interest in creating and studying cryptographic schemes which base their security on problems believed to be resistant to quantum computers. The U.S. national institute of standards and technology is already evaluating such options for future cryptography. The majority of candidate schemes in stage 2 of the evaluations are cryptographic schemes which base their security on the difficulty of lattice problems[2]. Hence, lattice-based cryptography is the most likely candidate for future encryption schemes. As such it is important to understand the possible attacks on such cryptographic functions. However, there are currently no papers or open source code of actual implementations of the theorised attacks on these schemes. Hence, the purpose of this essay is to offer an explanation on how to practically implement one of the theorised attacks on schemes that base their security on the shortest vector problem, specifically the sieving algorithm originally proposed in [6]. An open-source implementation of the algorithm in python is also offered in 5.3

The essay will begin with a few necessary definitions in section 2. Then an example cryptographic scheme which bases it's security on the shortest vector problem is introduced in section 3. This is followed by the core of this essay, section 4, which covers explicit explanations of the steps in the sieving algorithm along with derivations on what values to use for constants used in the algorithm in order to minimize runtime while mainting functionality.

2 Definitions

2.1 Big O notation

For functions $f(x) = O(g(x))$, there must exist some finite number $M > 0$ and x_0 such that

$$f(x) \leq Mg(x) \forall x > x_0 \tag{1}$$

In practice, this has the effect of only considering the fastest growing component of $g(x)$, ignoring for instance any constants which become insignificant for large input sizes. Often used in computer science to denote how the runtime of an algorithm changes with a varying input size.

2.2 Definition of a Lattice and the shortest vector problem

A lattice is the set of all points in a vector-space $\mathbb{R}^n$ which can be expressed as the set of all integer linear combinations of a set of given basis vectors.

$$\mathcal{L}(b_1, ..., b_n) = \left\{ \sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z} \right\}$$

For this essay the basis vectors will be limited to integer vectors $b \in \mathbb{Z}$. The set of basis vectors for a lattice can be represented with a matrix $B = [b_1, ..., b_n] \in \mathbb{R}^{n \times n}$ where the columns represent the basis vectors for the lattice. Then a lattice can be defined as containing each point which can be arrived at via multiplying the basis matrix with integer vectors with standard matrix vector multiplication $\mathcal{L}(B) = \{Bx : x \in \mathbb{Z}^n\}$

For lattice problems any norm can be used to define magnitude, for this essay only the euclidean norm $||v|| = (\sum_i v_i^2)^{1/2}$ is considered. With this we can define $\lambda_1(\mathcal{L}_i) = min||v||$ given that $v \in \mathcal{L}_i$ and $v \neq 0$. Thus, the shortest vector problem (SVP), is simply the problem of finding this shortest nonzero vector v for a given input lattice $\mathcal{L}_i$.

3 Ajtai's Hash Function

This section will cover the first lattice-based cryptographic function with security guarantees based on the worst-case hardness of lattice based problems originally presented by Ajtai in 1996[3]. Namely, Ajtai's cryptographic hash function. All one needs to know about hash functions to follow the following explanation is that hash functions must be injective, collision resistant and one-way functions.

3.1 The function

The essence of Ajtaj's function is as simple as doing standard matrix-vector multiplication onto the input, with a few qualifications. An input x is chosen in the form of a binary vector[1] of dimensionality m, $x = \{0,1\}^m$. A key for the function in the form of a uniform integer matrix $A_q^{n \times m}$ is then chosen with which the input is multiplied to get the output y, which is an n dimensional integer vector. The key matrix A is chosen uniformly from $\mathbb{Z}_q^{n \times m}$. In other words, each of it's members are chosen uniformly randomly from the set of integers modulo q. Such that, $A \in \mathbb{Z}_q^{n \times m}$. The output of the function is then simply the product of A and x.

$$f_A(x) = Ax \pmod{q} \tag{2}$$

So the output, y, will be in the form of an m dimensional integer vector where the values are modulo q, $y \in \mathbb{Z}_q^n$

3.2 Security guarantees

Given that $n < m$, it is clear that Ajtaj's function is injective, as the output will be a shorter vector than the input. The guarantees for collision resistance and the hardness of inverting the function on the other hand are based on the worst-case hardness of approximate SVP to an approximation factor equal to n [7]. This section will not delve deeply into this proof. The purpose is only to give an overview of how this hash function links to lattice problems as presented in Ajtai's original paper[3].

3.2.1 Difficulty of inversion

To invert the function an attacker would have to be able to find an input vector x, which leads to a given output y, given that the attacker knows the key matrix A. It is easy for an attacker to find an input t to the function such that $f_A(t) = y$ via traditional linear algebraic methods such as, gaussian elimination. One only needs to solve a single linear algebraic equation. However, t is unlikely to be short, unlike x which was defined to only have binary values, hence at most have

[1] It isn't strictly necessary for the message (input vector x) to contain only binary values. Other integer vectors can also be allowed. However, for succinctness of explanation of the function and its collision resistance guarantee, we use only binary vectors, without loss of generality as shown in [3]

a euclidean norm of $\sqrt{m}$. Therefore t is not a solution. However, once the attacker has an input t the problem of finding x is equivalent to the shortest vector problem. To see this equivocation we use a new type of construction for a lattice based on the matrix A called the kernel set of A, $\Lambda^{\perp}(A)$ which will consist of all the integer vector points which when multiplied with the matrix A will result in 0 $(mod\, q)$. So, that is:

$$\Lambda^{\perp}(A) = \{p \in \mathbb{Z}^m : Ap = 0 \pmod{q}\} \tag{3}$$

Considering any point p in this lattice we can say that $y + Ap \pmod{q} = y$, as by definition $Ap \pmod{q} = 0$. Therefore, due to the linearity of matrix multiplication we can say that $At + Ap \pmod{q} = A(t + p) \pmod{q} = y$ and since $\Lambda^{\perp}(A)$ contains all possible integer p, the solution x to the inversion problem must be equivalent to $t + p$ for some p. Therefore the solution x must be within a shifted copy of the kernel set of A, $x \in \Lambda^{\perp}(A) + t$. Therefore, finding x is equivalent to finding a short binary vector, $||x||_{\infty} = 1$, in the lattice $\Lambda^{\perp}(A) + t$ and so the problem of inverting $f_A(x)$ is equivocated to a lattice problem. It is easy to see how this lattice problem is similar to constant factor approximation SVP, since in both cases we are looking for a short vector in a lattice and as mentioned earlier the average case hardness of this lattice problem has been linked to the worst case hardness of constant factor SVP to a constant factor of n in [7].

3.2.2 Collision resistance

For an attacker to be able to find a collision in the hash function means that given that $f_A(x) = y$ an attacker is able to find an x' for which $f_A(x') = y$. Hence, $Ax \pmod{q} = Ax' \pmod{q} = y$. Which can be rearranged as $Ax - Ax' \pmod{q} = A(x - x') \pmod{q} = 0$. Let us define vector $z = x - x'$. For which it obviously must be true that $Az\ (mod\, q) = 0$. Furthermore, as x and x' must be binary vectors containing only values 0 or 1, any vector representing the difference of two such vectors must be a ternary vector, with possible values of -1, 0 or 1. Therefore $z \in \{-1, 0, 1\}^m$. Now, recalling the definition for a kernel set given in equation 3, we see that with these two properties of z it must be a short member, $||z||_{\infty} = 1$, of the kernel set of A. Therefore finding a collision to the hash function f_A is equivalent to finding a short integer vector in $\Lambda^{\perp}(A)$. The same SIS (short integer solutions) lattice problem as to which the difficulty of inversion is based on, but with allowing also members of value -1 and without the shift of the lattice.

4 Sieving algorithms for solving SVP

Sieving algorithms are one of the first types algorithms proposed for solving the shortest vector problem, the first version of which was proposed in 2001[6]. The original publication mostly contains proofs for the plausibility and functionality of the algorithm, while omitting detailed explanations and being opaque about the actual implementations of the proposed functions and steps. The purpose of the following explanation is to focus on the practical implementation. An open source example implemented in python is offered in Section 5.3. I will refer to [6] for proofs which have already been worked out and will mostly focus on new derivations for the sake of brevity.

4.1 General overview

The sieving algorithm generates a set of perturbed lattice points x_i, by taking a lattice points z_i and adding a perturbance $y_i \in R^n$ to them. It then assigns each an approximator lattice point a_i.

The heart of the sieving algorithm is the sieving procedure which increases the accuracy of these approximators until the shortest lattice point can be found from the set of differences between the approximators a_i and the original lattice points z_i from which any given x_i was derived from.

4.2 Choosing the starting set of vectors

The first step in the sieving algorithm is to choose a starting set of vectors $S = \{z_1, z_2, ..., z_N\}$, where each member z_i is a lattice point randomly sampled from within a half-closed parallelepiped. Firstly let's define N, the required size of the set S for the algorithm to work correctly. The only restriction on N is that it must be large enough such that $(N - O(n2^{2n}))2^{-O(\log n)} > 0$. This is given in lemmas 10 and 11 of [6]. In order to optimize run-time we want to minimize the value of N. In order to calculate the minimum viable N we define $N = 2^{c_1 n}$ and rearrange the above expression for the restriction of N with this definition, solving for the possible values of c_1 dependant on the dimensionality of the input n. Starting off by omitting the big O notation, since $O(n2^{2n}) = n2^{2n}$ and $O(\log n) = \log n$. Nets the equation

$$(2^{c_1 n} - n2^{2n})2^{\log n} > 0$$

Since $2^{log\, n}$ is always positive, we may derive that

$$2^{c_1 n} - n2^{2n} > 0$$

$$2^{c_1 n} > n2^{2n}$$

$$\log_2(2^{c_1 n}) > \log_2(n2^{2n})$$

$$c_1 n > \log_2 n + \log_2(2^{2n}) = 2n + \log_2 n$$

And since n is always positive, we may divide both sides by n

$$c_1 > 2 + \frac{\log_2 n}{n}$$

Therefore we get an equation for N of the form

$$N = \lceil 2^{(2+\log_2(n)/n+c)n} \rceil$$

Where c is some some small positive constant and the symbols $\lceil$ and $\rceil$ are used to denote that the value is rounded up to the nearest integer since N must be an integer value.

With a value for N we may now begin sampling vectors to add to the set S. The first step is to define the half-closed parallelepiped $\mathcal{P}$ from which we will be uniformly sampling lattice vectors from. A half-closed parallelepiped is described as the set of all points which can be obtained via linear combinations of a set of basis vectors $\{p_1, ..., p_n\}$ with weights from the half closed unit interval $(0, 1]$. So, that is, the set $\{\sum_{i=1}^{n} \alpha_i p_i | (\forall i \in \{1, ..., n\})(0 \leq \alpha_i < 1)\}$. In order to generate the basis for $\mathcal{P}$ we start by generating the standard orthonormal basis of $\mathbb{R}^n$ denoted by $\{e_1, e_2, ..., e_n\}$. Then we define a new set $\{f_i, ..., f_n\}$, which is an elongated version of the orthonormal basis, where each $f_i = De_i$ for a choice of $D \in \mathbb{Z}$, specified in section 4.5.2. A basis change is then performed for each f_i such that it is expressed in the basis of the input lattice $\mathcal{L}$. Then each value in f_i is rounded to the nearest integer, these lattice points make up the basis set for $\mathcal{P}$.

These sets can be represented as matrixes, where each column is one of the vectors in the sets, Hence, the set of orthonormal basis vectors $\{e_1, e_2, ..., e_n\}$ is the same as the identity matrix for

the dimension n, I_n and the elongated basis is the matrix $B = DI_n$. The basis change can then be achieved via multiplying the matrix B with the inverse of the lattice basis L^{-1}. Each value in the matrix is then iterated through and rounded to the nearest integer, then to get back to the orthonormal basis, we again apply standard matrix multiplication, now with the basis of the lattice, L. As such the procedure for getting the basis set for $\mathcal{P}$ in the form of a matrix P_s with vectors in the standard basis can be represented by the equation $P_s = L\lfloor L^{-1}DI_n \rceil$. Where L is the lattice basis in the form of a matrix and the symbols $\lfloor$ and $\rceil$ represent rounding each element in the contained matrix to the nearest integer.

The next step is to uniformly sample N vectors from $\mathcal{L}_i \cap \mathcal{P}$.

An inefficient approach

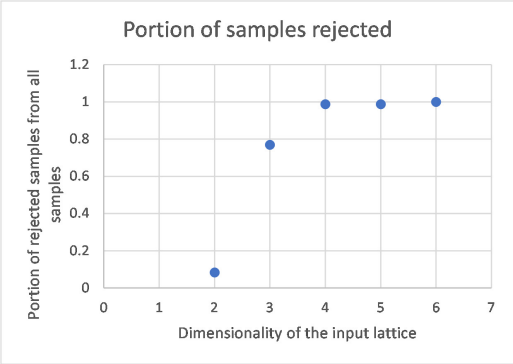
The approach used to sampling vectors from $\mathcal{L}_i \cap \mathcal{P}$ generates uniformly random lattice points from an area larger than $\mathcal{P}$ and then checks whether they are contained in the parallelepiped. The procedure starts by iterating through the columns of $P_l = \lfloor L^{-1}DI_n \rceil$, the matrix with the basis of $\mathcal{P}$ expressed in the lattice basis as its columns. For each column $c_i \in P_l$ a vector v_i of dimensionality n is generated via iterating through the values $k_j \in c_i$ and for each value choosing a random integer in $[0, k_j]$ which will constitute the jth value in vector v_i. Once all the columns have been iterated through, we define a final vector $v_f = \sum_{i=0}^{n} v_i$ which represents a uniformly sampled lattice point in the basis of the lattice. The next step is to check whether this v_f intersects with $\mathcal{P}$ and isn't a zero vector.

For the checking procedure we transform v_f into it's representation in the standard orthonormal basis. $v_s = Lv_f$. Checking whether this lattice point is the zero vector is as simple as taking the standard euclidean norm and confirming that $||v_s|| \neq 0$. In order to confirm whether the vector is in the parallelepiped $\mathcal{P}$ we can take the dot product of v_s with each basis vector p_i of $\mathcal{P}$ in the standard orthonormal basis, which in this case would be each column of P_s, then divide this by the norm of the basis of $\mathcal{P}$ and confirm that this value is within the unit interval $[0, 1)$. So, that is, $0 \leq \frac{p_i \cdot v_s}{||a_i||} < 1$ for each i in $\{1, ..., n\}$. This confirms that v_s can be described as a linear combination of the basis of $\mathcal{P}$ within multipliers in $[0, 1)$

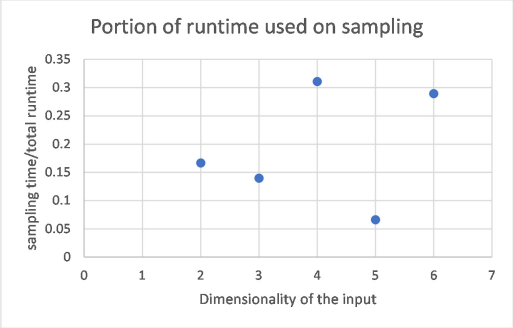
This procedure is likely not the fastest possible approach, since a large part of the generated lattice points will be rejected, this rejected portion seems to asymptotically become 1, 1a , for dimension 6 the rejected portion was already 0.9997. Regardless it is inconclusive whether this procedure takes up a significant amount of runtime at higher dimensions as can be seen in 1b, no obvious trend arises, more research is needed. However, the procedure functions as expected and so was implemented.

4.3 Adding the perturbations

The process of adding perturbations to the vectors in S to gain the set of perturbed vectors X is as simple as looping through each vector in S and then for each component of any given vector, adding independent and identically distributed random variables from a normal distribution with a mean of 0 and standard deviation of $1/\sqrt{Kn}$ and then appending the resultant vector to X. With a K specified by section 4.5.1.



(a) Portion of sampled vectors outside of $\mathcal{P}$ per dimension n



(b) Fraction of runtime dedicated for sampling procedure

Figure 1: The lattices used for these values are given in 5.2

4.4 The sieving procedure

The goal of the sieving procedure is to increase the accuracy of a set of approximator lattice points $a_1, ..., a_i$ to their corresponding surviving perturbed lattice points $x_1, ..., x_i$, until the input lattice's shortest vector $\lambda_1(L)$ can be found from the set of differences $(z_1 - a_1), ..., (z_i - a_i)$. Where $z_1, .., z_i$ are the original lattice points from which their respective $x_1, ..., x_i$ were derived from. In the beginning of this procedure all approximators are n-dimensional zero vectors.

4.4.1 Ball packing

The choice of representative vectors must be so that when subtracted from the vectors which they represent, the resultant vectors' euclidean norm will be less than $R/2$ where R is defined as the longest possible vector in the current set S for the first iteration of the sieving procedure. R is then calculated by taking the euclidean norm of the furthest vertex of the parallelepiped $\mathcal{P}$. This can be done by creating a vector s formed from adding each column of P_s and defining $R = ||s||$. One method for choosing these representative vectors and assigning survivor vectors to these representatives is to cover the parallelepiped $\mathcal{P}$, in which each vector from S is contained in, by n-dimensional balls with radius of at most $R/4$ and then assigning each ball with a randomly chosen vector to be the representative for the other vectors within that given ball. The simplest way to do this packing is to use a greedy packing technique. There is no need for the packing to be maximally efficient as long as it assuredly covers the entirety of $\mathcal{P}$. Therefore for the first packing we can create eight points along each basis vector p_j of $\mathcal{P}$ which are each $\frac{||p||}{8}$ apart, starting from the origin and then shifted by $\frac{p}{16}$. So that is $C_j = \{c_1, ..., c_8\}$ where $c_i = \frac{i*p_j}{8} - \frac{p_j}{16}$ and p_j is the jth column of P_s. Then we define C_s as the set of all vectors which can be described as a combination of elements, from each C_j, so that is, $C_s = \{a_1 + ... + a_j | a_1 \in C_1, ..., a_j \in C_j\}$. If we then define each point in C_s as a center for an n-ball with radius $R/4$ it covers $\mathcal{P}$ and that therefore we can use the set of balls with centers at C_s in our procedure for selecting the representatives.

4.4.2 Choosing representatives

The procedure for choosing the representatives takes as an input a set of centres for balls and the current R and a set of surviving vectors in the form $x_i - a_i$ which fall within these balls and the procedure outputs a set of representatives chosen from the set of inputted vectors along with the surviving vectors and a map from each representative, $x_j - a_j$ to each survivor $x_i - a_i$ such that for any vector/representative pair $||(x_j - a_j) - (x_i - a_i)|| \leq R/2$ for the given R. The first step is to assign each vector to one of the ball centres which it is inside of. This can be done by looping through each $x_i - a_i$ and for each $x_i - a_i$ looping through each ball centre c_i until one is found where $||c_i - (x_i - a_i)|| \leq R/4$. The second step is to loop through each ball and if there are vectors assigned to this ball, choose one of them, for instance, the one at index 0 to be the representative for the rest of the vectors assigned to that ball.

4.4.3 Assigning approximators

For this section let us consider the surviving perturbed lattice points $x_1, ..., x_i$, all the lattice points which have never been assigned as representatives. For each surviving perturbed lattice point x_i we consider its representative x_j. And we define the surviving point's new approximator to be it's old approximator plus the difference between the lattice point from which it's representative was derived and that same representative's current approximator, so that is, $a_i + z_j - a_j$. This step halves the maximum possible difference between each surviving lattice point and it's respective approximator.

4.4.4 The checking procedure

After the assignment of each approximator we check whether the euclidean norms of differences between each surviving perturbed lattice point and their representative approximators are all below a certain threshold, c_5, specified in lemma 12 of [6] to be $4 * c_3$ where c_3 is the maximum allowed perturbation, used in the selection of K in 4.5.1. Hence, $c_5 = 4 * 2 = 8$. If the check passes and for each surviving x_i it is true that $||z_i - a_i|| \leq 8$. The program prints out the shortest nonzero $z_i - a_i$ it found and then terminates itself, since $z_i - a_i = \lambda_1(\mathcal{L}_i)$, as per lemma 8 of [6] If not, it continues to the procedure specified in 4.4.5

4.4.5 Changes for further iterations of sieving

If the program did not terminate at the checking procedure, the next step will be to define a new R which will be half of the R value used in the previous iteration of the sieving procedure, so that is $R := R/2$, since we want this procedure to again half the average length of the surviving vectors.

Since we now have non-zero approximators, the set of surviving vectors can point in any direction so the ball packing procedure is also changed slightly, C_j now also contains points with negative values and these points are chosen from along the axes rather than a parallelepiped, so that is, $C_j = \{c_i | (e_j * (\frac{i*R}{8} - \frac{7*R}{16}) * 2)(\forall i \in \{1, ..., 8\})\}$ where e_j is the standard orthonormal basis in the jth dimension. With this, C_s can then be formed via the same procedure as previously defined in 4.4.1. For another iteration of sieving these ball centres gained along with the new R value are inputted into the procedure shown in 4.4.2. Then 4.4.3 is run again without changes, which prompts the checking procedure in 4.4.4. These steps of the sieving procedure are repeated until the program is terminated by the checking procedure.

4.5 Specified values for some variables

4.5.1 Choice of K

As per lemma 6 and 7 of [6] K should be chosen such that for all the perturbations $y_1, ..., y_i$ it holds that $||y_i|| \leq c_3$ with probability of at least $1 - 2^{-C'n}$ where $C' > 2$ and c_3 is the longest expected length for the shortest vector in the input lattice, which as per lemma 5 of [6] can be defined as $c_3 = 2$ without loss of generality. So this means for any $1 \leq i \leq N$, $\mathbb{P}(||y_i|| > c_3) < 2^{-C'n}/N \Rightarrow \mathbb{P}(||y_i||^2 > (c_3)^2) < 2^{-C'n}/2^{c_1 n}$, because by the union bound $\mathbb{P}(\bigcup_i [||y_i|| > c_3]) \leq \sum_i \mathbb{P}(||y_i|| > c_3) = N * \mathbb{P}(||y_i|| > c_3)$ and $N = 2^{c_1 n}$.

Lemma 21 of [6] proves that the above inequality is true as long as $\mathbb{P}(||y_i||^2 > (c_3)^2) < e^{-n((K/4)-(\frac{1}{2}(c_3)^2))}$. Hence, the choice of K has to be large enough that $e^{-n((K/4)-(\frac{1}{2}(c_3)^2)} < 2^{(-C'n-c_1 n)}$ yet it is preferable for K to be as small as possible while satisfying this equation in order to minimize the probability of finding zero vectors in the checking procedure specified in section 4.4.4. Hence, in order to define an appropriate K we simplify this equation.

$$e^{-n((K/4)-(\frac{1}{2}(c_3)^2))} < 2^{-n*(C'+c_1)} = (2^{(C'+c_1)})^{-n}$$

$$-n((K/4) - (\frac{1}{2}c_3^2)) < \ln((2^{(C'+c_1)})^{-n}) = -n * \ln(2^{(C'+c_1)})$$

Since $-n$ is always negative, dividing both sides by it nets us the inequality

$$(K/4) - (\frac{1}{2}(c_3)^2) > \ln(2^{(C'+c_1)})$$

$$K/4 > \ln(2^{(C'+c_1)}) + \frac{1}{2}(c_3)^2$$

$$K > 4 * (\ln(2^{(C'+c_1)}) + \frac{1}{2}(c_3)^2)$$

Substituting appropriate values for the constants, $C' = 2$, $c_1 = 2 + \frac{\log_2 n}{n}$ and $c_3 = 2$, and adding small positive constant c to the end of the equation, we may define K in terms of n as

$$K = 4 * (\ln(2^{(2+2+\frac{\log_2 n}{n})}) + \frac{1}{2}2^2) + c$$

$$= \ln(2^{\frac{4*\log_2 n}{n}}) + \ln(2^{16}) + 8 + c$$

Here C' was substituted by 2 and c_1 by $2 + \frac{\log_2 n}{n}$ even though earlier it was defined that $C' > 2$ and $c_1 > 2 + \frac{\log_2 n}{n}$. This was done for clarity and may be done since if C' was in stead defined as $C' = 2 + c_a$ and $c_1 = 2 + \frac{\log_2 n}{n} + c_a$ where c_a was some small positive constant, in aggregate it would add to the equation a constant component of $2 * \ln(2^{c_a})$. Now, choosing a small enough c_a such that for the previous choice of c it is true that $c_a < \log_2(e^{c/2})$ we can just choose a new $c_n = c - 2 * \ln(2^{c_a})$ for the original equation, which would equate it with the definition which includes c_a. Hence, the choice of c_1 and C' are appropriate.

4.5.2 Choice of D

There are two constraints on the value of D. It must be large enough to contain at least N lattice points, in order to lower the probability of gaining duplicate lattice points, and it must be large enough that all possible vectors $x_1, x_2, ..., x_N$ are within the area covered by the balls from 4.4.1. As per lemma 5 in [6] without loss of generality it may be assumed that the input lattice $\mathcal{L}(b_1, b_2, ..., b_n)$ can be represented in a basis which satisfies that $\max_{i=1}^{n} ||b_i|| \leq 2^{\epsilon n}$ for some $\frac{1}{2} < \epsilon < 1$. Let $M = 2^{\epsilon n}$ and $\mathcal{Q} = P(f_1, ..., f_n)$, recalling the definition for f_i from 4.2. In the process of rounding each member of f_i in the lattice basis to the nearest integer to gain a_i, each component could've shifted at most by $\frac{M}{2}$, half the length of the longest possible lattice basis vector. Hence, $\max_{i=1}^{n} ||f_i - a_i|| \leq (n * (\frac{M}{2})^{\frac{1}{2}})^2 = n^{1/2} * M/2$. Thus considering the distance from any given vertex of $\mathcal{Q}$ to its corresponding vertex in $\mathcal{P}$ it is at most $n * \max_{i=1}^{n} ||f_i - a_i|| \leq n^{3/2} * M/2$. Hence, if we then obtain a $\mathcal{Q}^-$ by shrinking $\mathcal{Q}$ about its centre by a factor of $1 - \frac{Mn^{3/2}}{2D}$. It must be true that $\mathcal{Q}^- \subset \mathcal{P}$. Consider the basis parallelepiped $\mathcal{B}$, which is defined as $\mathcal{P}(b_1, b_2, ..., b_n)$ for a lattice $\mathcal{L}(b_1, b_2, ..., b_n)$. For the set of parallelepipeds $W = \{v + \mathcal{B} | v \in \mathcal{L}\}$ all elements which intersect $\mathcal{Q}^-$ will be fully contained in $\mathcal{P}$. Hence $\frac{\text{vol}(\mathcal{Q}^-)}{\text{vol}(\mathcal{B})}$ is an accurate minima for the number of lattice points in $\mathcal{P}$. And since $\text{vol}(\mathcal{B}) = \det(L)$ and $\text{vol}(\mathcal{Q}^-) = (D * (1 - \frac{Mn^{3/2}}{2D}))^n$ and $N = 2^{c_1 n}$ we can work out a D which satisfies our first constraint from the equation

$$\frac{(D - \frac{Mn^{2/3}}{2D})^n}{|\det(L)|} > 2^{c_1 n}$$

$$D - \frac{Mn^{2/3}}{2D} > (2^{c_1 n} * |\det(L)|)^{1/n}$$

$$\frac{2D^2 - Mn^{2/3}}{2D} > 2^{c_1} * |\det(L)|^{1/n}$$

$$2D^2 - Mn^{2/3} > 2D * 2^{c_1} * |\det(L)|^{1/n}$$

$$\frac{-2}{|\det(L)|^{1/n}} D^2 + 2D * 2^{c_1} < \frac{-Mn^{2/3}}{|\det(L)|^{1/n}}$$

$$\frac{-2}{|\det(L)|^{1/n}} D^2 + 2^{c_1+1} D + \frac{Mn^{2/3}}{|\det(L)|^{1/n}} < 0$$

Since for $f(D) = \frac{-2}{|\det(L)|^{1/n}} D + 2^{c_1+1} D + \frac{Mn^{2/3}}{|\det(L)|^{1/n}}$ with some constant $n \in \mathbb{Z}^+$, $f'(D) < 0$ at $f(D) = 0$ for a $D > 0$, we may define

$$D = \frac{-2^{c_1+1} - \sqrt{2^{2*c_1+1} + \frac{8*Mn^{2/3}}{(\det(L))^{2/n}}}}{-4/|\det(L)|^{1/n}} + c$$

Which satisfies our first constraint given that c is some positive constant. For the second constraint to be satisfied given the ball packing procedure in 4.4.1 then R, the distance from the origin to the furthest vertex of $\mathcal{P}$ must satisfy the equation $c_3 + \frac{R}{16} < \frac{R}{4} \Rightarrow c_3 < \frac{3R}{16} \Rightarrow R > \frac{16c_3}{3}$. Because due to the linearity of the parallelepiped and the ball packing method, the furthest possible point from the centre of a ball inside of $\mathcal{P}$ is along the line from the origin to the furthest vertex, and

the worst case scenario is for there to be a point z_i near this vertex to which the maximum allowed perturbation c_3 is added along this same line and if the resultant x_i falls outside the area the closest ball covers, with a radius of $R/4$, the algorithm will fail. To assure a selection of D for which this is the case, we consider the longest vertex to vertex line of length r found in$\mathcal{Q}^-$. Since $r < R$ if r satisfies the constraint so will R and as we can write r in terms of D, $r = \sqrt{n * (D * (1 - \frac{Mn^{3l2}}{2D}))^2}$, we can then solve for a D that satisfies this constraint when $c_3 = 2$ as specified in Section 4.5.1.

$$\sqrt{n * (D * (1 - \frac{Mn^{3l2}}{2D}))^2} > \frac{16 * 2}{3}$$

$$\sqrt{n} * (D - \frac{Mn^{3l2}}{2D}) > \frac{32}{3}$$

$$D - \frac{Mn^{3l2}}{2D} > \frac{32}{3\sqrt{n}}$$

$$\frac{2D^2 - Mn^{3l2}}{2D} > \frac{32}{3\sqrt{n}}$$

$$2D^2 - Mn^{3l2} > \frac{64D}{3\sqrt{n}}$$

$$2D^2 - \frac{64}{3\sqrt{n}}D - Mn^{3/2} > 0$$

$$-2D^2 + \frac{64}{3\sqrt{n}}D + Mn^{3/2} < 0$$

Since for $g(D) = -2D^2 + \frac{64}{3\sqrt{n}}D + Mn^{3/2}$ with some constant $n \in \mathbb{Z}^+$, $g'(D) < 0$ at $g(D) = 0$ for a $D > 0$, we may define

$$D = \frac{-\frac{64}{3\sqrt{n}} - \sqrt{\frac{4096}{9n} + 8Mn^{3/2}}}{-4} + c$$

Let $h(n) = \frac{-2^{c_1+1} - \sqrt{2^{2*c_1+1} + \frac{8*Mn^{2/3}}{(\det(L))^{2/n}}}}{-4/|\det(L)|^{1/n}}$ and $p(n) = \frac{-\frac{64}{3\sqrt{n}} - \sqrt{\frac{4096}{9n} + 8Mn^{3/2}}}{-4}$, where $M = 2^{\epsilon n}$ with an $\epsilon = 0.75$ and $c_1 = 2.01 + \frac{\log_2 n}{n}$. $(\forall n \in \mathbb{Z}^+)(h(n) + 6.6 > p(n))$. Hence we may define $D = \lceil h(x) + 6.6 \rceil$. Furthermore, in the sample code D was defined as $\lceil 2^{1.4n} + 60 \rceil$, which can be done because $(2^{1.4n} + 60 > h(n) + 6.6)(\forall n \in \mathbb{Z}^+)$

4.6 Runtime

Graphing the runtime per dimension for my given implementation of the sieving algorithm and running an exponential regression algorithm on the data yields the function $g(x) = 6 * 10^{-6} * e^{3.6866x}$ seen in Figure 2. $O(g(x)) \approx e^{3.69x}$. This shows that for any reasonably dimensioned lattice, functions that base their security on the hardness of SVP are safe from an attack by an implementation of the type of sieving algorithm specified in this essay.

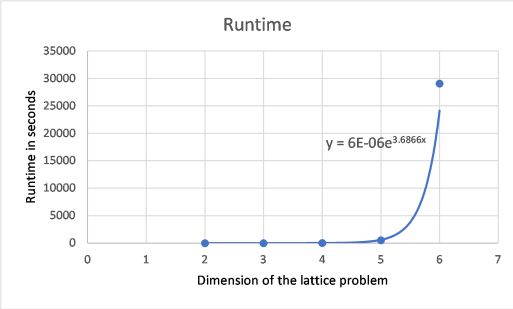


Figure 2: Runtime (seconds) per dimensionality of the input lattices specified in 5.2

References

[1] Chen, L., Jordan, S., Liu, Y., Moody, D., Peralta, R., Perlner, R. & Smith-Tone, D. (2016). Report on Post-Quantum Cryptography. NIST. http://dx.doi.org/10.6028/NIST.IR.8105

[2] Kanad, B., Deepraj S., Mohammed, N. & Ramesh K. (2019). NIST. NIST Post-Quantum Cryptography-A Hardware Evaluation Study. https://eprint.iacr.org/2019/047.pdf

[3] Ajtai, M. (1996). Generating hard instances of lattice problems. San Jose, CA. STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing, pp. 99–108. https://doi.org/10.1145/237814.237838

[4] Ajtai, M. (1998) The shortest vector problem in L2 is NP-hard for randomized reductions. San Jose, CA. Proc. 30th ACM Symposium on Theory of Computing, pp. 10-19. https://doi.org/10.1145/276698.276705

[5] Micciancio, D. (2001). The shortest vector in a lattice is hard to approximate to within some constant. SIAM J. Comput. https://doi.org/10.1137/S0097539700373039

[6] Ajtai, M., Kumar, R. & Sivakumar, D. (2001). A Sieve algorithm for the Shortest Lattice Vector Problem. San Jose, CA. STOC '01: Proceedings of the thirty-third annuaetcl ACM symposium on Theory of computing, pp. 601-610. https://doi.org/10.1145/380752.380857

[7] D. Micciancio and O. Regev. Worst-case to average-case reductions based on Gaussian measures. In Proc. 45th Annual IEEE Symp. on Foundations of Computer Science (FOCS), pages 372–381, 2004.

5 Appendix

5.1 Table of variables, constants and inputs.

Table 1 contains the variables used in the following example and explanation along with how they are defined and where they are presented for reference.

5.2 The lattices and hardware used for runtime data

The lattices used for all the runtime information in this essay, such as seen in figures 1a, 1b & 2, were randomly generated integer lattices mod 6. The exact lattices used can be seen in lines 6-10 of the code

Name	Value	Definition	Specified in
$\mathcal{L}_i$	$\mathcal{L}(b_1, b_2, ..., b_n)$	Input lattice	2.2
L	$\begin{pmatrix} b_1 & b_2 & ... & b_n \end{pmatrix}$	Input lattice in matrix form	4.2
n	$L \in R^{n \times n}$	Dimensionality of input lattice	2.2
M	$2^{\epsilon n}, \frac{1}{2} < \epsilon < 1$	$\max_i \|\|b_i\|\| \leq M$	lemma 5 of [6]
S	$\{z_1, z_2, ..., z_N\}$	Set of starting lattice points	4.2
N	$\lceil 2^{(2+\log_2(n)/n+c)n} \rceil$	Number of starting vectors	4.2
D	$\lceil \frac{-2^{c_1+1} - \sqrt{2^{2*c_1+1} + \frac{8*Mn^{2/3}}{(\det(L))^{2/n}}}}{-4/\lvert\det(L)\rvert^{1/n}} + 6.6 \rceil$	Elongation factor for the basis of a parallelepiped	4.5.2
$\mathcal{P}$	$\{\sum_{i=1}^{n} \alpha_i p_i \mid (\forall i \in \{1, ..., n\})(0 \leq \alpha_i < 1)\}$	A half closed parallelepiped with basis set $\{p_1, ..., p_n\}$	4.2
P_s	$\begin{pmatrix} p_1 & p_2 & ... & p_n \end{pmatrix} = L\lfloor L^{-1} D I_n \rceil$	Basis vectors for the parallelepiped $\mathcal{P}$ as columns of a matrix	4.2
K	$\ln(2^{\frac{4*\log_2 n}{n}}) + \ln(2^{16}) + 8 + c$	Perturbance factor, set of perturbance's s.d. $= 1/\sqrt{Kn}$	4.5.1
c_1	$c_1 > 2 + \frac{\log_2 n}{n}$	constant used in the definition $N = 2^{c_i n}$	4.2
z_i	$z_i \in \mathcal{L}_i$	a point in the input lattice	4.1
y_i	$y_i \in \mathbb{R}^n$	The perturbance added to z_i to gain x_i	4.5.1
x_i	$x_i \in \mathbb{R}^n$	A perturbed lattice point	4.1
a_i	$a_i \in \mathcal{L}_i$	An approximator for x_i	4.1
$\lambda_1(\mathcal{L}_i)$	$min\|\|v\|\| : v \in \mathcal{L}_i, v \neq 0$	Shortest vector in $\mathcal{L}_i$	2.2
c_3	$c_3 = 2$	max $\|\|\lambda_1(\mathcal{L}_i)\|\|$ and hence max perturbance	lemma 5 of [6]
c_5	$c_5 = c_3 * 4$	maximum allowed $z_i - a_i$ before final step	lemma 12 of [6]

Table 1: Variables and constants

5.3 Example code for an implementation of the Sieving algorithm

Github link to open source code: https://github.com/MiikaVuorio/SVP_algorithms

```
import numpy as np
import random
import math
import time

two_D = np.array([[0, 1], [6, 2]])
three_D = np.array([[3, 4, 4], [3, 2, 2], [3, 1, 6]])
four_D = np.array([[2, 5, 3, 4], [6, 3, 0, 0], [5, 0, 0, 2], [0, 4, 5, 6]])
five_D = np.array([[3, 5, 6, 5, 4], [2, 0, 0, 3, 5], [5, 0, 3, 6, 2], [2, 2, 0, 6, 4], [2, 2, 5, 3, 6]])
six_D = np.array([[1, 0, 6, 6, 0, 5], [3, 4, 3, 5, 1, 1], [3, 0, 4, 1, 3, 2], [3, 5, 6, 5, 6, 5], [2, 2, 0, 0, 5, 0], [4, 0, 2, 1, 5, 1]])
L = two_D
n = len(L[0])
c_0 = 1.4
c_1 = 2 + math.log(n, 2) / n + 0.000001
c_3 = 2
c_5 = 4 * c_3
D = math.ceil(2 ** (c_0 * n) + 60)
K = math.ceil(math.log(2**(4*math.log(n, 2)/n)) + math.log(2**16) + 8.000001)
min_num_vectors = 2 * 8 ** n  # Isn't an actual requirement, the exception from this has been removed
N = math.ceil(2 ** (n * c_1))
#print(N)

def unitary_random_vectors(B, L, N, n):
    failed = 0
    success = 0
    total = 0
    standard_basis_parallelepiped = np.matmul(B, L)
    uni_rand_vectors = []
    while True:
        rando_vec = np.zeros(n)
        for d in B:
            values = []
            for v in d:
                if v < 0:
                    min = v
                    max = 0
                else:
                    min = 0
                    max = v
                values.append(random.randint(min, max))
```

```
41              new_vec = np.array(values)
42              rando_vec += new_vec
43
44          rand_vec_standard_basis = np.matmul(rando_vec, L)
45
46          # checking whether vector is in parallelepiped
47          in_parallelepiped = True
48          for v in standard_basis_parallelepiped:
49              dot = np.dot(v, rand_vec_standard_basis)
50              v_norm = np.linalg.norm(v)
51              if dot < 0 or dot / v_norm > v_norm or np.linalg.norm(
                    rand_vec_standard_basis) == 0:
52                  in_parallelepiped = False
53                  failed += 1
54                  #total += 1
55                  break
56
57          #Check for duplicates
58          # for vec in uni_rand_vectors:
59          #     same_elems = 0
60          #     for val in range(len(vec)):
61          #         if vec[val] == rando_vec[val]:
62          #             same_elems+=1
63          #     if same_elems == n:
64          #         in_parallelepiped = False
65          #         print("vec already in")
66          #         print(vec)
67          #         print(rando_vec)
68
69          if in_parallelepiped:
70              uni_rand_vectors.append(rando_vec)
71              #total += 1
72              #success += 1
73          else:
74              pass
75          #print(str(failed/total) + str(success))
76          #if success % 100 == 0:
77          #    print(success)
78          if len(uni_rand_vectors) >= N:
79              break
80      print(failed)
81      print(failed + N)
82      print("portion of samples that failed:")
83      print(failed/(failed+N))
84      return uni_rand_vectors
85
```

```
86  def chec_for_duplicates(vectors):
87      index = 0
88      duplicates = 0
89      for vector in vectors:
90          position = 0
91          for against in vectors:
92              if position != index:
93                  if np.array_equal(vector, against):
94                      duplicates += 1
95              position += 1
96          index += 1
97      return duplicates
98
99
100
101 # Unused function, thought of a cool way to uniformly choose random vectors
        from inside, didn't work.
102 def each_vec(B, dimensions):
103     gcds = []
104     for i in range(len(B)):
105         values = B[i].tolist()
106         spot = 0
107         for num in values:
108             values[spot] = int(abs(num))
109             spot += 1
110
111         gcds.append(np.gcd.reduce(values))
112
113     vec_points = []
114     for vecs in range(gcds[0] + 1):
115         new_vec = B[0] * (vecs / gcds[0])
116         vec_points.append(new_vec)
117     for basis in range(1, len(B)):
118         new_vecs = []
119         for vecs in range(1, gcds[basis] + 1):
120             new_vec = B[basis] * (vecs / gcds[basis])
121             new_vecs.append(new_vec)
122         new_comb_vecs = []
123         for v in range(len(vec_points)):
124             for n in range(len(new_vecs)):
125                 new_comb_vecs.append(vec_points[v] + new_vecs[n])
126         for vec in new_comb_vecs:
127             vec_points.append(vec)
128
129     # min_num = 2 * 8 ** dimensions
130     #
```

```
131     # if len(vec_points) < min_num:
132     #     raise Exception("Not enough vectors")
133
134     return vec_points
135
136
137 def add_perturbation(S, K, n):
138     perturbed = []
139     v_perturbations = []
140     for biggoIndex in range(len(S)):
141         perturbation = np.random.normal(0, 1 / ((K * n) ** (1 / 2)))
142         new_v = np.array([S[biggoIndex][0] + perturbation])
143         v_perturbation = np.array([perturbation])
144         for index in range(1, len(S[biggoIndex])):
145             more_perturbs = np.random.normal(0, 1 / ((K * n) ** (1 / 2)))
146             new_boi = S[biggoIndex][index] + more_perturbs
147             new_v = np.insert(new_v, len(new_v), new_boi)
148             v_perturbation = np.insert(v_perturbation, len(v_perturbation),
                    more_perturbs)
149         perturbed.append(new_v)
150         v_perturbations.append(v_perturbation)
151     return perturbed, v_perturbations
152
153
154 def packing_balls(paralellepiped_basis):
155     circ_centres = []
156     for numerator in range(8):
157         new_cent = paralellepiped_basis[0] * (numerator / 8) +
                paralellepiped_basis[0] / 16
158         circ_centres.append(new_cent)
159     for basis in range(1, len(paralellepiped_basis)):
160         new_vecs = []
161         for vecs in range(1, 8):
162             new_vec = paralellepiped_basis[basis] * (vecs / 8) +
                    paralellepiped_basis[basis] / 16
163             new_vecs.append(new_vec)
164         new_comb_vecs = []
165         for v in range(len(circ_centres)):
166             for n in range(len(new_vecs)):
167                 new_comb_vecs.append(circ_centres[v] + new_vecs[n])
168         for vec in new_comb_vecs:
169             circ_centres.append(vec)
170
171     return circ_centres
172
173
```

```
174 def assign_vectors(circ_centres, vectors, R, n):
175     r = R / 4
176 
177     assigned_centres = []
178     vec_index = 0
179     centre_dic = {}
180     for i in range(len(circ_centres)):
181         centre_dic[i] = []
182 
183     for v in vectors:
184         centre_index = 0
185         for centre in circ_centres:
186             if np.linalg.norm(v - centre) <= r:
187                 assigned_centres.append(centre_index)
188                 centre_dic[centre_index].append(vec_index)
189                 vec_index += 1
190                 break
191             centre_index += 1
192             if centre_index == 8 ** n:
193                 print(v)
194 
195                 raise Exception("This ain't supposed to happen")
196 
197     return assigned_centres, centre_dic
198 
199 
200 def assign_dict_vectors(circ_centres, vectors, R, n):
201     final_centre = len(circ_centres)
202     r = R / 4
203 
204     assigned_centres = []
205     centre_dic = {}
206     for i in range(len(circ_centres)):
207         centre_dic[i] = []
208 
209     for v in vectors.keys():
210         centre_index = 0
211         for centre in circ_centres:
212             if np.linalg.norm(vectors[v] - centre) <= r:
213                 assigned_centres.append(centre_index)
214                 centre_dic[centre_index].append(v)
215                 break
216             centre_index += 1
217             if centre_index == final_centre:
218                 print(circ_centres)
219                 print(8 ** n)
```

```
220                 print(len(circ_centres))
221                 print(R)
222                 print(vectors[v])
223                 print(np.linalg.norm(vectors[v]-circ_centres[final_centre -
                        1]))
224                 print(r)
225 
226                 raise Exception("A vector wasn't in any of the balls")
227 
228     return assigned_centres, centre_dic
229 
230 
231 def change_from_basis(B, S):
232     basis_changed = []
233     for v in S:
234         basis_changed.append(np.matmul(v, B))
235     return basis_changed
236 
237 
238 def choose_reps(centre_dic, min_num_vectors):
239     reps = []
240     saved_from_execution = []
241     marked_for_execution = []
242 
243     for i in centre_dic.keys():
244         if len(centre_dic[i]) != 0:
245             index = len(centre_dic[i]) // 2
246             reps.append(centre_dic[i][index])
247             centre_dic[i].pop(index)
248             if len(centre_dic[i]) == 0:
249                 saved_from_execution.append(i)
250     for i in centre_dic.keys():
251         if len(centre_dic[i]) == 0 and i not in saved_from_execution:
252             marked_for_execution.append(i)
253     for identity in marked_for_execution:
254         del centre_dic[identity]
255 
256     # if len([item for subl in centre_dic.values() for item in subl]) <
            min_num_vectors:
257     #     raise Exception("not enough vectors")
258 
259     # reps has to be less than half of total, maybe add this check
260 
261     return centre_dic, reps
262 
263 
```

```
264 def to_dic(vecs):
265     xi_dic = {}
266     for i in range(len(vecs)):
267         xi_dic[i] = vecs[i]
268     return xi_dic
269 
270 
271 def sieve(current_xis, current_ais, zis, no_reps, reps):
272     new_ais = {}
273     temp_fix = 0  # better fix is to make reps into a dictionary form as well
274 
275     for i in no_reps.keys():
276         for index in range(len(no_reps[i])):
277             new_ais[no_reps[i][index]] = current_ais[no_reps[i][index]] + zis[
                    reps[temp_fix]] - current_ais[
278                 reps[temp_fix]]
279 
280         del current_xis[reps[temp_fix]]
281         del zis[reps[temp_fix]]
282         temp_fix += 1
283 
284     return current_xis, new_ais, zis
285 
286 
287 def post_sieve_ball_fun(R, n):
288     ball_centres_per_dimension = []
289     ball_centres = []
290     for dimension in range(n):
291         ball_centres_per_dimension.append([])
292         for multiplier in range(8):
293             ball_place = np.zeros(n)
294             ball_place[dimension] = (-R * 7 / 16 + R * multiplier / 8) * 1.2
295             ball_centres_per_dimension[dimension].append(ball_place)
296 
297     final_ball_centres = []
298     for bb in ball_centres_per_dimension[0]:
299         ball_centres.append(bb)
300     for other_dims in range(n - 1):
301         final_ball_centres = []
302         for b in ball_centres_per_dimension[other_dims + 1]:
303 
304             for ball in ball_centres:
305                 final_center = ball + b
306                 final_ball_centres.append(final_center)
307 
308         ball_centres = []
```

```
309             for biko in final_ball_centres:
310                 ball_centres.append(biko)
311         final_ball_centres.append(np.zeros(n))
312 
313         return final_ball_centres
314 
315 
316 def sample_vectors(L, n, D, K, c_5, min_num_vectors):
317         orthonormal_basis = np.identity(n)
318         fi_basis = orthonormal_basis * D
319         fi_in_lattice_basis = np.matmul(fi_basis, np.linalg.inv(L))
320         fi_rint_L_basis = np.rint(fi_in_lattice_basis)
321         #print("len of sample_points:")
322         sample_start = time.time()
323         each_sample_point = unitary_random_vectors(fi_rint_L_basis, L, N, n)
324         sample_end = time.time()
325         #print(len(each_sample_point))
326         #print("num of duplicates:")
327         #print(chec_for_duplicates(each_sample_point))
328         sample_time = sample_end - sample_start
329         print("time for sample: " + str(sample_time))
330         zis_in_standard_basis = change_from_basis(L, each_sample_point)
331         zis = to_dic(zis_in_standard_basis)
332         perturbed_samples, perturbations = add_perturbation(each_sample_point, K,
                n)
333         yis_in_standard_basis = change_from_basis(L, perturbations)
334         perturbed_in_standard_basis = change_from_basis(L, perturbed_samples)
335         current_xis = to_dic(perturbed_in_standard_basis)
336         parallelepiped_in_standard_basis = np.matmul(fi_rint_L_basis, L)
337         circ_centres = packing_balls(parallelepiped_in_standard_basis)
338         R = np.linalg.norm(parallelepiped_in_standard_basis)
339 
340         centre_indexes, centre_dic = assign_vectors(circ_centres,
                perturbed_in_standard_basis, R, n)
341         no_reps, reps = choose_reps(centre_dic, min_num_vectors)
342 
343         current_ais = {}
344         for i in current_xis.keys():
345             current_ais[i] = np.zeros(n)
346 
347         current_xis, current_ais, current_zis = sieve(current_xis, current_ais,
                zis, no_reps, reps)
348 
349         while True:
350             survivors = current_zis.keys()
351             smalls = 0
```

```
352             tot_sum = 0
353             zeros = 0
354             for key in survivors:
355                 diff = np.linalg.norm(current_zis[key] - current_ais[key])
356                 if diff <= c_5:
357                     smalls += 1
358                 if diff == 0:
359                     zeros += 1
360                 tot_sum += diff
361             #print(zeros)
362             print(tot_sum / len(survivors))
363 
364             if smalls == len(survivors):
365                 break
366             else:
367                 R = R / 2
368                 current_xis_minus_ais = {}
369                 for key in current_xis.keys():
370                     current_xis_minus_ais[key] = current_xis[key] - current_ais[
                        key]
371 
372                 circ_centres = post_sieve_ball_fun(R, n)
373                 centr_int, centre_dictionary = assign_dict_vectors(circ_centres,
                    current_xis_minus_ais, R, n)
374                 no_reps, reps = choose_reps(centre_dictionary, min_num_vectors)
375                 current_xis, current_ais, current_zis = sieve(current_xis,
                    current_ais, current_zis, no_reps, reps)
376 
377         possible_vectors = []
378         for key in current_zis.keys():
379             possible_vectors.append(current_zis[key] - current_ais[key])
380 
381         sh_vec = np.array([999999])
382         sh_len = c_5 + 1
383         for vector in possible_vectors:
384             vec_len = np.linalg.norm(vector)
385             if vec_len < sh_len and vec_len != 0:
386                 sh_len = vec_len
387                 sh_vec = vector
388                 sh_len = vec_len
389 
390         print(sh_vec)
391         print(sh_len)
392 
393 start_time = time.time()
394 sample_vectors(L, n, D, K, c_5, min_num_vectors)
```

```
395 end_time = time.time()
396 print("program time: " + str(end_time-start_time))
```