Internship Report

Christian Ang

Part 1: Kyber Analysis

Kyber is a post-quantum cryptographic scheme which aims to encrypt messages in polynomials. It is built around the NP-hard Shortest Vector Problem (SVP), as well as perturbation brought about by the Learning with Errors (LWE).

As such, Kyber exists as a Lattice-based Cryptographic algorithm, with the modulo function implemented to further prevent the secret from being recovered through matrix inversion.

The parameters of Kyber-256 public key encryption can be defined as follows:

Algorithm 1: Kyber Transmission Process

Let n be the dimensions of square matrix A
Let d be the size of polynomial entries in A

p <- (x^256+1)
q <- 3329
x := x mod p mod q

Then:
A <- LatticeGeneration(n,d)
s <- SmallVectorGeneration(n,d)
e <- SmallVectorGeneration(n,d)

t = As + e

m <- binary(message) * q/2

Random Perturbation Values:
r <- SmallVectorGeneration(n,d)
e1 <- SmallVectorGeneration(n,d)
$e2$ <- SmallVectorGeneration(n,1)

u = transpose(A)***r** + **e1**
v = transpose(T)***r** + m + $e2$

Transmit <- (***u***, v, **A**, **t**, p, q)

m = **u** - transpose(**s**)*v

Where all randomly generated values follow {$x \in X \sim \text{NormPDF}(\mu, \delta^2) \mid \mu = 0, \delta^2 = 1 \text{ or } q/2$}. As such, the purposes of e1 and e2 are listed below:

Claim 1: The purpose of LWE parameter e is to protect the product of **As** and hence **s**, due to high probability of invertibility of A.

Proof 1: If A is randomly generated, the matrix A would be invertible (through Gaussian Elimination) with high probability (Bayesian) as given below:

$$P_{invertible} = 1 - (1/q\text{^}n\text{-}1)\text{^}n\text{-}1$$

As such, there is a need for perturbation parameter **e** to prevent secret key **s** from being recovered from **t** using **A**. Though **e** aims to prevent this as such as possible, several areas were investigated to exploit certain known problems when attempting to recover private key **s** from **A** and **t**, as explored in Part 5.

Claim 2: The purpose of LWE parameter **e1** is to protect the value of randomness parameter **r**, which can be reversed engineered through the inverting of $\mathbf{A}^{\mathbf{T}}$.

Proof 2: Referencing Proof 1, if A is invertible (through Gaussian Elimination) with high probability, it should come without a doubt that $\mathbf{A}^{\mathbf{T}}$ would also be invertible as $(\mathbf{A}^{\mathbf{T}})^{-1} = (\mathbf{A}^{-1})^{T}$. The discovery of **r** would lead to attackers having the ability to recover the message **m** through:

$$\mathbf{m} + e2 = \mathbf{v} - \mathbf{t}^{\mathbf{T}}\mathbf{r}$$

Generally, Kyber aims to perturbate values of secrets to allow for difficulty in reverse-engineering the message through either matrix inverting or subtraction, and is built around the fact that the shortest vector that exists within the Lattice **L**, where {$\mathbf{A} \in \mathbf{L}$}, cannot determined in Polynomial-time. Thus, current efforts for Kyber Secret Key recovery are for lower dimensions of Kyber, affectionately dubbed "Baby Kyber". However, this paper aims to introduce an alternate view to Kyber Key recovery, which could lead to future investigations and expansions.

Fun Facts about Kyber as I was coding it, with the code here:

1.Implementation of Kyber in Python

- Convolution of Polynomial via DFT as opposed to normal convolution

- Requirement of f to have low coeff, or there is a need to find mod inverse of fractions (Euclidean Algo / Bezout's Coefficients) – but f is already x^n + 1, as such, this could possibly improve Kyber?

- Max transmission is 32 characters, or transmissions can reveal xor

2. Breaking Kyber using SVP oracle

- Referencing Voronoi Cells, will find the nearest point with high probability, but the error term needs to be smaller than the reduced vector for correct rounding

- Importance of Multiplication by the determinant

- Secret key **s** will be scaled as well.

3. Implementation as a KEM

- Required to transmit random secret **ss**

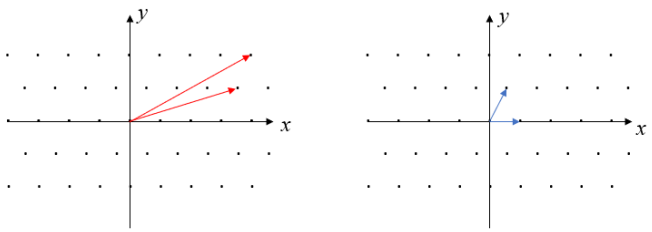
- Adopted in a "hybrid" model in encryption schemes

- Okamoto-Uchiyama

Part 2: Shortest Vector Problem

The shortest vector problem (SVP) is an unsolved problem that aims to find the shortest vector v within a given lattice L. The definition of "shortest vector" may vary, but in the case of this paper, it refers to the Euclidean Norm of v. [Mic05] This is particularly useful in recovering secrets in Kyber as shorter basis vectors can better describe the lattice at hand.

Figure 2.1: Example of a Bad Basis (Red) and a Good Basis (Blue)



Better basis can be described as shorter vectors that describe the same lattice. Since basis vectors are required to have the same determinant to describe the same lattice, a short basis could be said to be as orthogonal as possible, and hence the intuition of a good basis.

We can define basis matrix $B = [b_1, b_2, \ldots, b_n] \in \mathbb{Z}^{n \times n}$, we denote the Lattice L by:

$$L(B) := \mathbb{Z}^n \cdot B = \{\alpha_1 b_1, \alpha_2 b_2, \ldots \alpha_n b_n | \alpha_i \in \mathbb{Z}\}$$

Where we can say basis B describes lattice L, and basis B can be reduced to contain shortest vector v. With this shortest vector v, it allows for lattices to be described at a much smaller scale, and for small vectors to be found, which can be defined by small integer outputs in a vector. As such, recovering secrets in Kyber, such as secret key **s,** will have to involve a conversion of Kyber into some SVP problem, which contains **s** and **e** as its shortest vectors, which will be covered in Part 3.

So far, current efforts [Aji98] have proven that for L_2 lattices, the SVP is NP-hard. Ergo, it is non-deterministic in polynomial time. In other words, it is hard for find solve the SVP deterministically in polynomial-time, and Kyber is built around this NP-hardness. [LPR13] states that the LWE problem can only be broken should Search-SVP be approximated in polynomial time for Ideal Lattices ($\mathbb{Z}[x]/x^n$-1), and so far, no such algorithm has been able to do so. Examples include: the AKS Algorithm [AKS01], Kannan's SVP Algorithm [Kan83] and the BKZ Algorithm.

There are two schools of thought when attempting to solve the SVP. One way is through probabilistic means and another is through deterministic means. As such, I attempted to investigate both schools of thoughts to understand the limitations and drawbacks in either method. The paper will focus on Hasting's generalisation of the Metropolis Algorithm in Solving SVP for a non-deterministic method, and Lenstra-Lenstra-Lovastz (LLL) Algorithm for a deterministic approach, and some improvements to the LLL Algorithm.

Finally, I will propose a surprisingly simple method that could be explored to convert the erfine Kyber problem into a linear problem, which could probably lead to a non-conventional method of solving Kyber.

Part 3: Conversion of Kyber M-LWE to SVP Problems

For the erfine Kyber problem, it is actually an instance of Module-Learning with Errors (M-LWE). This idea originates from the Learning with Errors problem (LWE) introduced by Regev [Reg05], with the main difference between an M-LWE and LWE being that M-LWE deals with polynomials within a ring while LWE (or more exactly Ring-LWE) deals with integers within a ring. Hence, the M-LWE has to be converted into an LWE problem, as polynomials cannot be operated like integers. To emulate the expansion of polynomials in linear algebra, one would note that one coefficient of any given x^n term in **A** is multiplied to multiple terms in **s**. This many to one multiplication cannot be emulated in (n x 1) dimensional linear algebra, hinting at the need for a repeated shuffle of coefficients in an entry within **A** to allow for the many-to-one multiplication to be emulated.

ie. For $(a \quad b)\begin{pmatrix} c \\ d \end{pmatrix} = ac + bd$, $ac + bd$ incorrectly represents the polynomial obtained when multiplying $(a + bx)(c + dx)$. Hence, there is a need for the following:

$$\mathbf{As} = \begin{pmatrix} a & 0 \\ b & a \\ 0 & b \end{pmatrix}\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ bc + ad \\ bd \end{pmatrix}$$

Which accurately represents the coefficients for each monomial in $(a + bx)(c + dx)$. When taken by a modulo f, where $f := x^n + 1$, it can be concluded that the result $x^n \equiv -1$, allowing for the integer A matrix of dimension nk x nk+1 to be simplified into an nk x nk square matrix, at the cost of negating elements with degrees above or equal to n. This causes:

$$\mathbf{As} \pmod f = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac - bd \\ bc + ad \end{pmatrix}$$

This effectively reduces an M-LWE over a known polynomial ring to an LWE problem over an integer ring.

This can be further reduced from LWE to SVP, as LWE exists about a given integer ring $\mathbf{R}_q \in \mathbb{Z}_q$. As such, given a basis matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$, $\mathbf{s} \in \mathbb{Z}_q^n$, $\mathbf{t} \in \mathbb{Z}_q^n$, when converted out of MLWE, the vector subspace extends to $\mathbf{F}_q \in \mathbb{Z}_q^{nk}$, causing dimensions to increase to $2nk + 1$ dimensions, causing $\mathbf{A} \in \mathbb{Z}_q^{nk \times nk}$, $\mathbf{s} \in \mathbb{Z}_q^{nk}$, $\mathbf{t} \in \mathbb{Z}_q^{nk}$.

To solve the LWE, it could be said that:

$\mathbf{t} - \mathbf{A} \cdot \mathbf{s} \equiv \mathbf{e} \pmod q$

This can be rewritten into a linear system as:

$$\begin{pmatrix} \mathbf{I} & 0 \\ -\mathbf{A} & q\mathbf{I} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{s} \\ * \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \mathbf{t} \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{e} \end{pmatrix}$$

Where * is an arbitrary vector that corresponds to the quotient of entries mod q.

When represented homogenously after reduce row echelon, the system can be re-represented as:

$$\mathbf{B} = \begin{pmatrix} \mathbf{I} & 0 & 0 \\ -\mathbf{A} & q\mathbf{I} & \mathbf{t} \\ 0 & 0 & 1 \end{pmatrix} \text{ st. } \mathbf{B} \cdot \begin{pmatrix} \mathbf{s} \\ * \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{e} \\ 1 \end{pmatrix}$$

This now suggests that $\begin{pmatrix} \mathbf{s} \\ \mathbf{e} \\ 1 \end{pmatrix}$ exists as a short vector within **B** (where $\dim(\mathbf{B}) = 2n+1$). Since all columns have minimally n 0s of $2n$+1 entries, it could be said that the output short vector would be comparatively and relatively short.

This leads to the possible recovery of $\begin{pmatrix} \mathbf{s} \\ \mathbf{e} \\ 1 \end{pmatrix}$ or a surrogate $\begin{pmatrix} \mathbf{s}' \\ \mathbf{e}' \\ 1 \end{pmatrix}$ should the SVP for LWE instances be solved. There is, however, no

guarantee that **s** recovered is the same as the generated private key, as it could refer to a different linear combination which allow **As** = **As'** as the initial, random private key **s**. However, it can be used in the same way as the private key **s**.

Part 4: Non-deterministic Methods

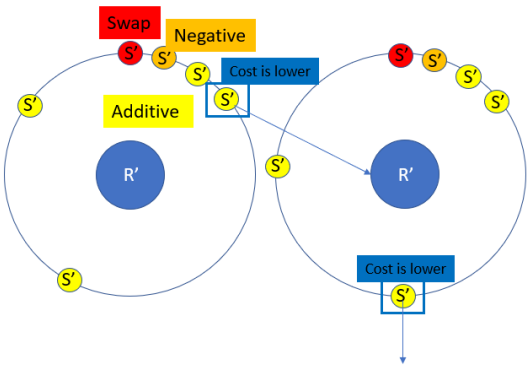
The primary probabilistic model that was implemented was in [ASP11] where a "path" from matrix **A** to reduced basis **B** was proposed. Essentially, the 5 steps involved in the process involved:

1. Generating the coefficient limit
2. Generating the multiplicative matrix
3. Choosing 1 of 3 possible elementary steps to reduce basis vectors in *B*
4. Checking for a vector in the resultant matrix to be lower than the initial shortest vector
5. Repeating steps 1-4 until a vector below threshold *K* is found.

The probabilistic model in question was the Metropolis Algorithm with Hastings' Generalisation for non-uniform probability inputs, which is especially important in Step 3 as it weights the probability towards less trivial possibilities.

The Metropolis Algorithm essentially take small "steps" in a heuristic search model towards a desired smaller vector length.

Figure 3.1: Diagrammatic approach as to how the Markov Chain Works



Operating on the idea that there will be some vector w for $Bw = u$, that will yield in a ||u|| < BestNorm. As such, vectors w can be repeatedly randomly generated to form matrix R = [w_1, w_2, … , w_m], with an identity matrix concatenated to form $R' = [R|I]$.

In [ASP11], the upper limit for the threshold of *w* was defined as:

$$M = (an)^n$$

Where *a* is the largest value between all entries in B, compared with K. {a = max($B_{1,1}$, $B_{1,2}$ … $B_{1,n}$, $B_{2,1}$, …$B_{n,n}$, K)}

However, this paper proposes an alternative to this claim and instead, proposes a lower bound to evaluate less neighbours at any one point. This can be done by using Cramer's Rule to determine the maximum entry in vector *w*, which would determine the number of neighbours around the initial Search Point, *R*'.

Proposition 3.1: The upper entry bound M can be defined as

$$M = \frac{(\alpha n)^n}{\det(A)}$$

Proof 3.1: Given $Bw = u$, by Cramer's Rule [Cra50], the entries of vector w would be defined as:

$$w_i = \frac{\det(B_i)}{\det(B)}$$

Where B_i is the matrix *B*, with column *i* replaced by vector *u*. The largest det(B_i) will result in the largest entry w_i. Naively, this can be proved in a 2x2 matrix, as:

$$\begin{aligned}\det(B) &= ad - bc \\ &\leq 5ad - bc \\ &\leq 5\alpha^2 - \alpha^2 \\ &\leq 4\alpha^2 \\ &\leq (\alpha n)^n \mid n=2\end{aligned}$$

As such, $(\alpha n)^n$ could be said to be a loose upper bound for $\det(B_i)_{max}$ and by applying Cramer's Rule, it could be possible to determine a more refined upper bound for $w_{i\,max}$. Through induction on *n*, this is true for higher values of *n*, and hence, true for higher dim(*B*). This reduces the possibility of trivial attempts of randomly generated matrix *R*, and less attempts will be required to check for all possible costs from a random matrix *R*, optimising the value of all neighbours, *d* to reduce the size of the search space effectively.

Referencing LLL proofs (which come in at a later stage), if the basis quality can be described by the following inequality:

$$||b_1|| \leq \sqrt{\frac{2}{\sqrt{3}}\det(L)}$$

It could also be concluded that with lattices of higher determinants comes a more ambiguous and looser upper bound on the basis returned through LLL.

Definition 3.2: The Search Space for SVP can be defined as the following:

1. [Definition of Search Space] The Search Space of matrix *R*' refers to the augmented matrix [*R*|*I*], where *R* is the *m* x *n* randomly generated matrix with entry value of maximum *M*, and *I* is the *n* x *n* identity matrix.
2. [Definition of Neighbourhood] The neighbourhood of matrix *R'* can be said to be the matrix *S*'. It is defined that each *R*' has $d = {}^mC_2 + m + {}^mC_2 \cdot 2\,(\log(M) + 1) + 2mn\,(\log(M) + 1)$ neighbours, of which one is the possible value of S'. The transition from R to S is described with probability as:
    a. Swap two columns of *R* with probability ${}^mC_2/d$
    b. Multiply a column of *R* by -1 with probability m/d
    c. Choose a random column in *R* and add a random power of 2 multiplied to a column in *R*' with residual probability
    ie. $r_i, <\text{-}\ r_i \pm c^* r_j, \{i \neq j,\ 1 \leq i \leq n,\ 1 \leq j \leq m+n\}$;
    $c \in \{2^0, 2^1, ..., 2^k\}$; $P_i = 1/(k+1)$; $k = \log(M)$
    Probability Proof:
    Number of possible *k* for $2^k = \log(M) + 1$
    Number of possible selections for *i* and *j*: ${}^nC_2 + mn$
    Number of possible *i* and *j* operations: 2
    Multiply all
3. [Definition of Cost] For neighbour *S'*, its cost is defined to be c(*S'*), where *t* is the Euclidean Norm of the shortest amongst the *m* + *n* vectors in *B*[*S*|*I*].

Algorithm 2: Metropolis Algorithm with Hastings' Generalisation

Input : Matrix B for lattice L
Output : Vector v in $B*R'$ where $||v|| < K$

K <- upper limit for shortest vector
n <- len(B)
m <- multiple of n

$B*w = u$, where $||u|| < K$
a = max(B, K)
$M = (a*n)^n/|\text{determinant}(B)|$

Calculate Search Space Element
Input: M, n, m
Output: Multiplicative matrix A'
A = randomint(M) **for** i **in** range(n) **for** i **in** range(m)
Let I be an n x n identity matrix
A' = [$A|I$]

Calculate Cost
Input: Neighbouring Matrix R', Basis matrix B
Output: Length c(R')
Compute ||v|| **for** all column vectors in $B*R'$
c(R') <- Shortest ||v||

R' = SearchSpaceElement(M, n, m)
c(R') = Cost(R')
Set *BestNorm* = c(R')
while *BestNorm* > K **do**:
  Select a neighbour S' of R' by doing one of three elementary operations below.
  $d = {}^mC_2 + m + {}^mC_2 \cdot 2\,(\log(M) + 1) + 2mn\,(\log(M) + 1)$
  1. (Trivial) Swap two columns of R' with probability ${}^mC_2/d$
  2. (Trivial) Multiply a column of R' with probability m/d
  3. Choose a random column in R and add a random power of 2 multiplied to a column in R' with residual probability ie. r_i ,<- $r_i \pm c*r_j$', $\{i \neq j,\ 1 \leq i \leq n,\ 1 \leq j \leq m+n\}$;
     $c \in \{2^0, 2^1, \ldots, 2^k\}$; $P_i = 1/k+1$; $k = \log(M)$
  if c(S') < *BestNorm* **then**
    $P_{transition} = \min(\frac{e^{-c(S')/T}}{e^{-c(R')/t}}, 1)$
    Set $R' = S'$ with $P_{transition}$
    BestNorm = c(R')
  end if
end while

Theorem 3.3: Let B ∈ $\mathbb{Z}^n$ in the lattice L, and K is the threshold for the SVP instance.

1. For every element [$R|I$] in the Search Space, each of the (m + n) vectors in B[$R|I$] is a vector in L.
2. If L contains a vector of norm K or less, then the search space for SVP with L will contain an element [$R|I$] such that one of the (m + n) column vectors of B[$R|I$] will be of norm K or less.

Definition 3.4: The probability stated:

$$P_{transition} = \min(\frac{e^{-c(S')/T}}{e^{-c(R')/T}}, 1)$$

can be attributed to Discrete Gaussian Sampling.

Due to the nature of the random walk model, it suggests that at times, moving in a counterintuitive direction could lead to more possibilities.

As such, by weighting the Probability distribution function of the Discrete Gaussian Sample to be:

$$P(X = x): f(x) = e^{-||x||/T}$$

With smaller values of x, $f(x)$ that has a negative gradient, will describe a larger probability with smaller values of $||x||$, allowing for a higher probability weighted towards smaller vectors. That is, given an appropriate temperature parameter T is chosen.

Temperature parameter T describes the "variance" of a discrete Gaussian distribution (DGD), influencing how "spread out" points are about a certain n-dimensional space, ie. A DGD with larger values of T will a distribution which is more evenly spread out and vice versa. This is problematic as T cannot be determined, and too few or too many probability points could be selected, making the algorithm redundant or inefficient .

When implemented in code here, when threshold d was reached, the random matrix R was regenerated to define a new search space. Additionally, a condition was added to prevent the initial cost t from being 0.

This method proves to be slow. [ASP11] claims that for every input, there is an O(mnlog(M)) long path for every basis to be reduced, through the optimisation to this algorithm given in Definition 3, the time complexity could be reduced slightly as the value of M reduces.

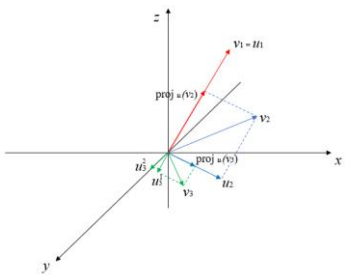
Additionally, an idea would be to generate larger R matrices to allow for comparison over more terms at any one point in time. This would increase computational complexity by O($n^{2.3}$) approximately [DWZ22], and the chance of linear dependency spawning within the randomly generated R matrix. As such, it would be advisable to keep the m-parameter to low multiples of n.

However, there is no guarantee that a vector below the predefined value of K exists, and the algorithm will just not find any answer to basis vector B. As such, the trial-and-error nature of finding the value of K comes to show the sheer difficulty in obtaining the shortest vector when using probabilistic models, but it does not discount the fact that it could be used as a "push" when deterministic algorithms are "stuck" which will be elaborated more in Part 5.

Part 5: Gram-Schmidt Orthogonalization (GSO)

For most lattice reduction algorithms, there needs to be a known value to dictate the transformations for the n vectors within **B**. This can usually be determined by finding the vector projection of v_i onto v_j {$i < j$}

Figure 5.1: GSO when $n = 3$



With increasing values of j there will be a need for increased number of projections to orthogonalize vectors in (j-1) dimensions, as there will be an increasing number of vectors that have been orthogonalised, which will be taken alignment to. As such, the following can be defined:

$$\text{proj}_{\mathbf{a}}\mathbf{b} = \mu_{a,b}(\mathbf{a}) = \frac{\langle \mathbf{b},\mathbf{a}\rangle}{\langle \mathbf{a},\mathbf{a}\rangle} \cdot \mathbf{a}$$

The Gram-Schmidt Process can be described below:

$u_1 = v_1$
$u_2 = v_2 - \text{proj}_{u1} v_2$
$u_3 = v_3 - \text{proj}_{u2} v_3 - \text{proj}_{u1} v_3$
...
$u_n = v_n - \text{proj}_{u(n-1)} v_n - \text{proj}_{u(n-2)} v_n - \ldots - \text{proj}_{u1} v_n$

It can be observed that orthogonal vectors can also be seen as a translation of the perpendicular between v_j and u_i.

As such, basis matrix $\mathbf{V}$ $\{\mathbf{V} = [v_1, v_2, \ldots, v_i, \ldots, v_n]\}$ can undergo GSO into matrix $\mathbf{U}$ $\{\mathbf{U} = [u_1, u_2, \ldots, u_i, \ldots, u_n]\}$, and returns values of $\mu_{i,j}$.

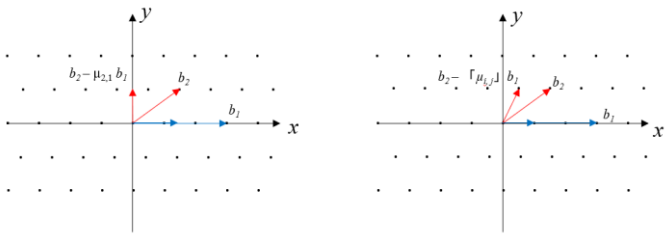
Geometrically, $\mu_{i,j}$ could be defined as the scale of the lateral movement needed for u_k to achieve an orthogonal position, which is key in the LLL algorithm

Part 6: LLL Reduction Algorithm

The Lenstra-Lenstra-Lovastz (LLL) Algorithm is a deterministic algorithm which returns a matrix of short basis vectors within a lattice space L.

The intuition behind the LLL Algorithm relies on the Gram-Schmidt Orthogonalization, where $\mu_{i,j}$ values refer to the projection of previous vectors, as a length to orthogonalise the current vector. This in infeasible in an actual lattice, as such scales will have to be integers, and $\mu_{i,j}$ values are usually decimals. As such, rounding $\mu_{i,j}$ to the nearest integer guarantees that the resultant point will lie within the lattice. Subsequently, "reducing" the vector using previous vectors multiplied to $\lceil \mu_{i,j} \rfloor$, which parallels the Gram-Schmidt Process would be a way to obtain shorter and orthogonal-ish vectors.

Figure 6.1: Basis Reduction



However, this is dependent on the fact that that the Gram-Schmidt vector behind of the current Gram-Schmidt vector is shorter than the current Gram-Schmidt vector. This is the case for two reasons:

1. The $\mu_{i,j}$ could possibly be more than 1, allowing for more "reduction" in the next step.
2. The re-ordering allows for shorter vectors to appear in front, "pushing" shorter vectors to the front.

Should the Gram-Schmidt vector ahead be shorter than the current Gram-Schmidt vector, a "swap" of the vectors could be performed to reorder the vectors, yielding in a descending order of Gram-Schmidt vectors.

Algorithm 3: LLL Algorithm

Input : Matrix $\mathbf{B} = [b_1, b_2, \ldots, b_n]$ for lattice L
Parameter $\boldsymbol{\delta} \in (0.25, 1)$
Output : Matrix B for lattice L which contain LLL-reduced bases

B* ← GramSchmidt (**B**)
$k \leftarrow 2$ » *serialiser*
while $k < n$ **do**
  for j **from** k-1 **to** 1 **do**
    $\mu_{k,j} \leftarrow \langle b_j, b_k^* \rangle / \langle b_k^*, b_k^* \rangle$ » *update* $\mu_{k,j}$
    $b_k \leftarrow b_k - \lceil \mu_{k,j} \rfloor b_j$ » *update* b_j
    Update **B*** *where necessary, or naively recompute* **B*** *through:*
    B* ← GramSchmidt (**B**)
  end for
  if $\delta \left|\left|b_{k-1}^*\right|\right|^2 < \left|\left|b_k^* + \mu_{k,k-1} b_{k-1}^*\right|\right|^2$ **then**: » *check Lovasz Condition*
    $k \leftarrow k+1$
  else:
    Swap b_k and b_{k-1};
    Update **B*** *where necessary, or naively recompute* **B*** *through:* » *update* b_j
    B* ← GramSchmidt (**B**)
    $k \leftarrow \max(k\text{-}1, 2)$
  end if
end while

These ideas of "Reduction" of a basis vector and "Swapping" if Gram-Schmidt vectors are too short form the baseline idea of finding a reduced basis. These are formalised in the LLL Algorithm, which rely on referencing two matrices as the reduction steps proceed. It takes the following parameter:

δ – scale of the Gram-Schmidt Vector compared

B – basis matrix with linearly independent columns.

The "reduction" step is conducted in the in the first half of the algorithm to reduce the norm of basis vectors. As $\mu_{i,j}$ values are rounded, it also guarantees that each vector will differ from the ideal orthogonal vector by a scale of maximally 0.5.

The "swap" step is done based on the length of the Gram-Schmidt vectors. This is formalised as the Lovasz Condition, which is stated below:

$$\delta \left|\left|b_{k-1}^*\right|\right|^2 \leq \left|\left|b_k^* + \mu_{k,k-1} b_{k-1}^*\right|\right|^2$$

This ensures the intuition that followed earlier, as this essentially forces two neighbouring Gram-Schmidt vectors to be descending in length, allowing for a longer projection and hence, longer reductions. Hence, if the Lovasz Condition passes, the b_k and b_{k-1} remain. If it fails, b_k and b_{k-1} are swapped. The Lovasz Condition can be rewritten as:

$$\delta ||b_{k-1}^*||^2 \leq \left|\left|b_k^*\right|\right|^2 + \left[\left||b_k|\right| \cos(\theta)\right]^2, \frac{\pi}{3} \leq \theta \leq \frac{2\pi}{3}$$

Which reflects the Pythagoras Theorem quite well, ensuring that there is a maximum value to $||b_{k-1}^*||$ as a basis for a swap.

As such, an LLL-reduced basis could be said to be:

1. (Size Reduced) $|\mu_{i,j}| \leq 0.5$ for all $1 < j < i \leq n$.
2. (Lovasz Condition) $\delta\left||b_{k-1}^*|\right|^2 \leq ||b_k^* + \mu_{k,k-1}b_{k-1}^*||^2$ is true for all $1 \leq k-1 \leq n$.

Theorem 6.1: The LLL Algorithm guarantees that:

$$||b_1||^2 \leq \left(\frac{4}{4\delta - 1}\right)^{n-1} (\lambda(L))^2$$

Proof: From the Lovasz Condition:

$$\delta\left||b_{k-1}^*|\right|^2 \leq ||b_k^* + \mu_{k,k-1}b_{k-1}^*||^2$$

$$(\delta - (\mu_{k,k-1})^2)\left||b_{k-1}^*|\right|^2 \leq ||b_k^*||^2$$

Considering the worst case of $|\mu_{k,k-1}|$ is 0.5, the worst case for the inequality above can be written as:

$$(\frac{4\delta - 1}{4})\left||b_{k-1}^*|\right|^2 \leq ||b_k^*||^2$$

$$\left||b_{k-1}^*|\right|^2 \leq (\frac{4}{4\delta - 1})||b_k^*||^2$$

By extending this inequality globally from 1 to *i*, a geometric progression is formed, wherein:

$$\left||b_1|\right|^2 = \left||b_1^*|\right|^2 \leq (\frac{4}{4\delta - 1})^{i-1}||b_i^*||^2 \leq (\frac{4}{4\delta - 1})^{n-1}||b_i^*||^2$$

Where b_i^* is the shortest Gram-Schmidt vector. Since $\lambda(L)$ is the shortest vector and can be obtained by some integer linear combination *Z* to **B,** it could be said that:

$$\lambda(L) = \sum_{j=1}^{n} z_j b_j = \sum_{j=1}^{n} z_j \sum_{l=1}^{j} \mu_{j,l} b_l^*$$

Since $\mu_{n,n} = 1$,

$$\lambda(L) = z_n b_n^* + z_n \sum_{l=1}^{n-1} \mu_{j,l} b_l^* + \sum_{j=1}^{n-1} z_j \sum_{l=1}^{j} \mu_{j,l} b_l^*$$

Lemma 6.2: Hence, $\left||\lambda(L)|\right| \geq ||b_n^*|| \geq ||b_i^*||$, and:

$$\left||b_1|\right|^2 \leq (\frac{4}{4\delta - 1})^{n-1}||\lambda(L)||^2$$

Theorem 6.2: The LLL Algorithm also guarantees a Polynomial runtime of the algorithm, in $O(n^2)$ time.

Proof: This can be naively attributed to the maximal $(n\text{-}1)^2$ number of swaps performed in the worst case of the algorithm, where $\left||b_{k-1}^*|\right|^2$ is always longer than $||b_k^*||^2$. This suggests the importance of the number of swaps in the practical runtime of the LLL Algorithm. As such, this accounts for the reliability of the LLL Algorithm when solving for an estimate of the SVP, in a stark contrast to the Hastings-Metropolis Algorithm.

Alternatively, the reduction step takes Polynomial Time, while Swap Step is done in Linear time. Now consider a monovariant S that describes the length of Gram-Schmidt vectors:

$$S \to \prod_{i=1}^{n} ||b_i^*||$$

In the reduction step, S does not change as the order of Gram-Schmidt Vectors remain the same. In the Swap step, since a condition of

$$\left||b_{k-1}^*|\right|^2 \leq (\frac{4}{4\delta - 1})||b_k^*||^2$$

Has to be met for a swap, it guarantees that S will decrease by at least a factor of $\sqrt{\frac{4\delta-1}{4}}$ at each swap. Since the maximum number of swap can be defined as $0.5n(n\text{-}1)$, the maximum number of times the monovariant reduces by is $log_{\sqrt{\frac{4\delta-1}{4}}} ||b_{min}^*||^{0.5n(n-1)}$ times, which is Polynomial in nature.

However, practically, the LLL Algorithm often does not return basis vectors that are short enough, and in the context of the $(2nk+1)$ dimensional matrix of Kyber SVP instances, the coefficient of $||\lambda(L)||^2$ can get slightly large for higher dimensions of Kyber, causing a "secret key" recovered to still be large. As such, the next part investigates different ways to optimise the LLL Algorithm at higher dimensions.

A working version of LLL code can be found here, and some observations when coding includes:

- Angular guarantee of B[k] and B[j] between 60 and 120 degrees, due to the rounding coeff, causing a projection of maximally 0.5 |a| to be formed. (mu <= 0.5, 60<arccos(mu)<120)

Part 7: Optimisations of LLL

Given that LLL Algorithm starts facing time, and subsequently, length issues when attempting to recover keys at higher dimensions, there was a need for an algorithm that produced shorter basis vectors, while taking less time to compute as well.

As such, one drawback of LLL was observed, and that was that at any one point in time, only one neighbour of b_{k-1}^* is being compared to itself. This would restrict the comparison of a shorter vector to one neighbour only, when there could be a situation where the vector immediately after is shorter as well.

Part 7.1: *m*-LLL Algorithm

This would be the intuition behind the *m*-LLL Algorithm [DZY17], which aims to compare projections of *m* Gram-Schmidt Vectors ahead of the current b_k^* vector onto b_k^* to ascertain the most optimal swap for maximum reduction at a time. As such, for a more efficient "swap" to be performed, just as how the length of b_k is compared to b_{k-1}in the Lovasz Condition, lengths up to b_{k+m-1} can be computed and compared.

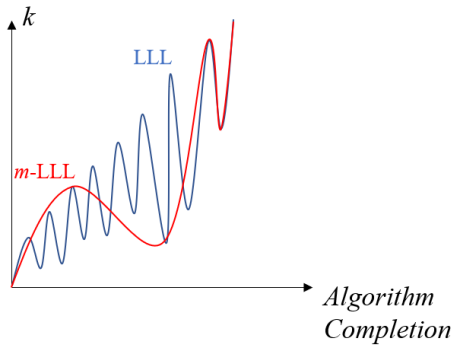
The Lovasz Condition can hence be extended to:

$$\delta||\mathbf{b}_{k-1}^*||^2 \leq \min\left(\left||\mathbf{b}_k^* + \mu_{k,k-1}\,\mathbf{b}_{k-1}^*\right|\right|^2, \ldots, \left|\left|\mathbf{b}_{k+m-1}^* + \sum_{j=k-1}^{\min(k+m-2,n)} \mu_{k+m-1,j}\,\mathbf{b}_j^*\right|\right|^2\right)$$

for $1 \leq k \leq n-1, 1 < i \leq m,$ and $\delta \in (0.25, 1]$

The impact of this is guaranteeing the same or a better basis vector in less swap steps, attributed to larger reduction steps.

Figure 7.1.1: *m*-LLL vs LLL *k* progression



Algorithm 4: *m*-LLL Algorithm

Input : Matrix $\mathbf{B} = [b_1, b_2, \ldots, b_n]$ for lattice L
Parameter $\boldsymbol{\delta} \in (0.25, 1)$
Output : Matrix B for lattice L which contain LLL-reduced bases

$\mathbf{B}^* \leftarrow$ GramSchmidt $(\mathbf{B})$
$k \leftarrow 2$ » *serialiser*
while $k < n$ **do**
  for j **from** k-1 **to** 1 **do**
    $\mu_{k,j} \leftarrow \langle b_j, b_k^* \rangle / \langle b_k^*, b_k^* \rangle$ » *update* $\mu_{k,j}$
    $b_k \leftarrow b_k - \lceil \mu_{k,j} \rfloor \, b_j$ » *update* b_j
    Update $\mathbf{B}^*$ *where necessary, or naively recompute* $\mathbf{B}^*$ *through:*
    $\mathbf{B}^* \leftarrow$ GramSchmidt $(\mathbf{B})$
  end for
  if $\delta \|\mathbf{b}_{k-1}^*\|^2 \leq \min\left(\left\|\mathbf{b}_k^* + \mu_{k,k-1}\,\mathbf{b}_{k-1}^*\right\|^2, \ldots, \left\|\mathbf{b}_{k+m-1}^* + \sum_{j=k-1}^{\min(k+m-2,n)} \mu_{k+m-1,j}\,\mathbf{b}_j^*\right\|^2 \right)$ » *check Extended Lovasz Condition*
  then:
    $k \leftarrow k+1$
  else:
    Swap b_{min} and b_{k-1}; » *update* b_j
    Update $\mathbf{B}^*$ *where necessary, or naively recompute* $\mathbf{B}^*$ *through:*
    $\mathbf{B}^* \leftarrow$ GramSchmidt $(\mathbf{B})$
    $k \leftarrow \max(k\text{-}1, 2)$
  end if
end while

This allows for larger reductions, given that:

$$\left|\|b_i^*\|\right| \leq \|b_{k-1}^*\|$$

$$\text{(from the Definition of } \mu_{i,j}) \quad \frac{\langle b_i^*,\ b_k\rangle}{\langle b_i^*,\ b_i^*\rangle} \geq \frac{\langle b_{k-1}^*,\ b_k\rangle}{\langle b_{k-1}^*,\ b_{k-1}^*\rangle}$$

Allowing for longer units of b_j (from Algorithm 2) to be reduced from basis vectors at any reduction step, which could be said to "compensate" for additional reduction steps that would have been taken if b_{k-1}^* was taken as the shorter vector.

However, the worst case of the m-LLL Algorithm can still be attributed to the scenario where $(n\text{-}1)^2$ swaps are performed, as at every stage of the algorithm, a shorter vector is found. Additionally, there is an $\left(\frac{m}{2}(m+1)\right)$ number of computations for a decision at the Extended Lovasz Condition. This could cause the runtime to exceed $O(n^2)$, but, since $m<n$, the guaranteed runtime should exceed $O(n^2)$ very slightly.

As such, a basis could be said to be m-LLL reduced if:

1. (Size Reduced) $|\mu_{i,j}| \leq 0.5$ for all $1 < j < i \leq n$.
2. (Extended Lovasz Condition)

$$\delta \|\mathbf{b}_{k-1}^*\|^2 \leq \min\left(\left\|\mathbf{b}_k^* + \mu_{k,k-1}\,\mathbf{b}_{k-1}^*\right\|^2, \ldots, \left\|\mathbf{b}_{k+m-1}^* + \sum_{j=k-1}^{\min(k+m-2,n)} \mu_{k+m-1,j}\,\mathbf{b}_j^*\right\|^2 \right)$$

for $1 \leq k \leq n-1, 1 < i \leq m,$ and $\delta \in (0.25, 1]$

Practically, the *m*-LLL Algorithm runs faster than an LLL Algorithm due to the "skipping" of redundant reductions, which can often compensate for the increased conditions computed at the Extended Lovasz Condition, with an implementation here.

One could observe that *m*-LLL is a generalisation of comparisons for LLL. Most notably, 2-LLL will be akin to the original LLL algorithm and $(n\text{-}1)$-LLL could be akin to a global LLL function, dubbed "DeepLLL".

Part 7.2: *Deep*LLL Algorithm

*Deep*LLL is another variant of the LLL Algorithm, which adds on to the first LLL-reduced basis vectors. In most practical cases, due to $0.25 < \delta < 1$, the first vector returned would usually not be the shortest vector, as it will fall within the threshold of a "short" vector, as a tradeoff for Polynomial Runtime. As such, the intuition for this optimisation lies in the fact that this shortest vector could be "inserted" to the position of vector b_1 in an LLL-reduced basis matrix to bypass this limitation, followed by another round of LLL-reduction. This reflected in Algorithm 5, and is a simplification of the *Deep*LLL Algorithm suggested by Schnorr.

This proves to be effective at returning shorter vectors as vectors are reduced to the shortest LLL basis vector b_i. However, should the first LLL vector be the shortest LLL-reduced vector, this algorithm will be nullified, leading to the need to compare the further reduced vectors between every LLL-reduced basis vector, which can be found in [SE94], this is the full implementation of *Deep*LLL, which can be found in Algorithm 6.

A practical implementation is written here, for both LLL+*Deep*LLL and *m*-LLL+*Deep*LLL, and results can be compared. On average, *m*-LLL+*Deep*LLL returns the same, if not, shorter vectors than LLL+*Deep*LLL algorithm in half the time taken. At higher dimensions, *Deep*LLL faces a similar issue to *m*-LLL, where the need for multiple comparisons causes the algorithm to appear slower practically.

Comparing between LLL and m-LLL, it is apparent why m-LLL may practically run slower than the LLL Algorithm, as it boils down to the computational space costs at the Lovasz Condition:

Table 7.2.1: Comparison of LLL and Deep/*m*-LLL

	b_i vectors	b_i^* vectors	$\mu_{i,j}$
LLL	2	2	1
m-LLL	*m*	*m*	$\frac{m}{2}(m+1)$

The increased space requirements lead to the need to find a quicker and numerically more stable method of calculating $\mu_{i,j}$ values. This leads to the need for the Givens' Rotation.

Algorithm 5: *Deep*LLL Algorithm Lite

Input : Matrix $\mathbf{B} = [b_1, b_2, \ldots, b_n]$ for lattice L
Parameter $\boldsymbol{\delta} \in (0.25, 1)$
Output : Matrix B for lattice L which contain *Deep*LLL-reduced bases

B* ← GramSchmidt (**B**)
$k \leftarrow 2$ » *serialiser*
B ← LLL(**B**)
if $\|b_1\| \neq \|b_i\|$ **then**: » *i is minimum norm serial*
  B := $[b_i, \ldots, b_1, \ldots, b_n]$
  B* ← GramSchmidt (**B**)
  k := 2
  B ← LLL(**B**)
  else:
    return B
end if

Algorithm 6: *Deep*LLL Algorithm Full

Input : Matrix $\mathbf{B} = [b_1, b_2, \ldots, b_n]$ for lattice L
Parameter $\boldsymbol{\delta} \in (0.25, 1)$
Output : Matrix B for lattice L which contain *Deep*LLL-reduced bases

B* ← GramSchmidt (**B**)
$k \leftarrow 2$ » *serialiser*
B ← LLL(**B**)
$c := ||b_k||^2$, $i := 1$ » *k is the shortest vector serial*
if $\delta c_i \leq c$ **then**:
  $c := c - \mu_{k,i} c_i$; $i := i + 1$
  else:
    $\mathbf{B} := [b_1, b_2, \ldots, b_{i-1}, b_k, b_{i-1}, \ldots, b_n]$ » *Swap before serial i*
    B* ← GramSchmidt (**B**)
    B ← LLL(**B**); GOTO 4
end if

Part 7.3: QR Decomposition and Givens' Rotation

The Gram-Schmidt Process is known as a method of QR decomposition, where any square matrix can be broken down into a normalised, orthogonal matrix (Q matrix) as well as an upper triangular matrix (R matrix). There actually three distinct methods of QR decomposition that are stated below:

Table 7.3.1: QR Decomposition methods

	Time	**Space**
Gram-Schmidt Process	$O(nk^2)$	$O(n^2)$
Householder Transformation	$O(n^3)$	$O(n^2+0.5n)$
Givens Rotation	$O(n^3)$	$O(n^2)$

To show this decomposition, recall that the Gram-Schmidt Process computes vectors as such:

$u_1 = v_1$
$u_2 = v_2 - \text{proj}_{u1} v_2$
$u_3 = v_3 - \text{proj}_{u2} v_3 - \text{proj}_{u1} v_3$
...
$u_n = v_n - \text{proj}_{u(n-1)} v_n - \text{proj}_{u(n-2)} v_n - \ldots - \text{proj}_{u1} v_n$

This can be represented as a matrix multiplication between:

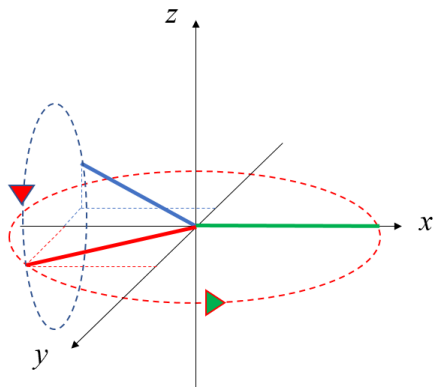
$$\begin{pmatrix} | & \cdots & | \\ v_1 & \cdots & v_n \\ | & \cdots & | \end{pmatrix} = \begin{pmatrix} | & \cdots & | \\ u_1 & \cdots & u_n \\ | & \cdots & | \end{pmatrix} \begin{pmatrix} 1 & \mu_{2,1} & \mu_{3,1} & & \mu_{n,1} \\ 0 & 1 & \mu_{3,2} & \cdots & \mu_{n,2} \\ 0 & 0 & 1 & & \mu_{n,3} \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Where $\mathbf{U} = [u_1, \ldots, u_n]$ is an orthogonal matrix. Normalising **U**, we get:

$$\begin{pmatrix} | & \cdots & | \\ v_1 & \cdots & v_n \\ | & \cdots & | \end{pmatrix} = \begin{pmatrix} | & \cdots & | \\ \widehat{u_1} & \cdots & \widehat{u_n} \\ | & \cdots & | \end{pmatrix} \begin{pmatrix} |u_1| & |u_1|\mu_{2,1} & |u_1|\mu_{3,1} & & |u_1|\mu_{n,1} \\ 0 & |u_2| & |u_2|\mu_{3,2} & \cdots & |u_2|\mu_{n,2} \\ 0 & 0 & |u_3| & & |u_3|\mu_{n,3} \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & |u_n| \end{pmatrix}$$

This is an example of QR decomposition. In particular, the low computational and space costs of the Givens Rotation piques our interest in optimising the orthogonalisation subroutine. Simply put, the Givens Rotation aims to "zero" basis vectors in as many dimensions as possible, while keeping the angles between the vectors, and length of the vectors the same. The intuiton behind this rotation would be that every row will require an (n-i) number of rotations to zero the vectors while keeping the angle the same. (ie. A 3x3 basis matrix will require 2 rotations in the 1st column, and 1 rotation in the 2nd column.) A visual of the transformation can be depicted below:

Figure 7.3.2: The Givens' Rotation for the 1st column of a 3x3 matrix



This will return the R Matrix after $(0.5n)(n+1)$ rotations, and the Q matrix can be obtain be multiplying all rotation matrices together (They are all Orthogonal and unitary anyway). Given a matrix:

$$\begin{pmatrix} 2 & 6 & 0 \\ 1 & 19 & 7 \\ 5 & 11 & 6 \end{pmatrix}$$

One would be able to define the first rotation matrix as such:

$$G_1 = \begin{pmatrix} I_{n-2} & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{pmatrix} \begin{pmatrix} 2 & 6 & 0 \\ 1 & 19 & 7 \\ 5 & 11 & 6 \end{pmatrix}$$

With the intention of zeroing the last entry in the first column ($B_{1,3}$)

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} 1 \\ 5 \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix}$$

Through trigonometric identities, we can then determine that:

$$c = \frac{a_1}{\sqrt{a_1^2 + a_2^2}}$$

$$s = \frac{a_2}{\sqrt{a_1^2 + a_2^2}}$$

Hence:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} 1 \\ 5 \end{pmatrix} = \begin{pmatrix} c+5s \\ -s+5c \end{pmatrix} = \begin{pmatrix} \sqrt{26} \\ 0 \end{pmatrix}$$

By repeating this process for different pairs of numbers, an upper triangular matrix will eventually be obtained.

$$\begin{pmatrix} |u_1| & \frac{\langle u_2 \cdot u_1 \rangle}{|u_1|} & \frac{\langle u_3 \cdot u_1 \rangle}{|u_1|} & & \frac{\langle u_n \cdot u_1 \rangle}{|u_1|} \\ 0 & |u_2| & \frac{\langle u_3 \cdot u_2 \rangle}{|u_2|} & \cdots & \frac{\langle u_n \cdot u_2 \rangle}{|u_2|} \\ 0 & 0 & |u_3| & & \frac{\langle u_n \cdot u_3 \rangle}{|u_3|} \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & |u_n| \end{pmatrix}$$

Another befit of QR decomposition is that the Lovasz Condition values can be directly read off the R matrix, reducing the number of floating point calculations and hence, the numerical stability of the Givens' Rotation.

$$||\mathbf{b}^*_{k+1}||^2 + (\mu_{k+1,k})^2 ||\mathbf{b}^*_k||^2 \geq \delta \left|||\mathbf{b}^*_k||\right|^2 = |u_{k+1}|^2 + \langle u_{k+1} \cdot u_k \rangle^2 \geq \delta |u_k|^2$$

Applying this *m*-LLL+ *Deep*LLL with Fast Givens actually enables Kyber secret keys to be recovered for up to 48 dimensions, and it was implemented here.

In terms of time complexity, *m*-LLL has a guaranteed Polynomial runtime, as it shares the same monovariant in a reducing Geometric Progression as the LLL Algorithm. However, *Deep*LLL by Schnorr has an exponential runtime, and at times, could yield in no solution being found.

Part 8: Integer Least Squares and Dual Matrices

Another method of solving the Kyber problem could also lie in solving the Dual of the public key. This tells us the impact of increasing any entry of **s** linearly. Usually, Dual attacks are performed on LWE instances, and its process is as follows:

A Dual is defined as the vector subspace for which linear functionals of *L* take integer values, ergo:

$$L^* = \{v \in \text{span}(L), st. \nabla x = L, v \cdot x \in \mathbb{Z}\}$$

$$B^* = B(B^T B)^{-1}$$

In other words, each point represents a vector which can be multiplied to **A** to give an integer output. This is interesting as SVP algorithms can be run on B*, where $L^* = \{B^*\}$, allowing us to identify the shortest vector within B*, which can be called x_j in future parts.

Given (**A, t**), one can find a short vector x_j^T of dimension (1 x *n*) in the Dual Lattice. This allows for $x_j^T \cdot A \equiv y_j \bmod q$, where y_j is small as well, and this is important in the result:

$$x_j^T \cdot \mathbf{t} = x_j^T \cdot \mathbf{A} \cdot \mathbf{s} + x_j^T \cdot \mathbf{e} = \langle y_j, \mathbf{s} \rangle + \langle x_j, \mathbf{e} \rangle$$

Where $\langle y_j, \mathbf{s} \rangle, \langle x_j, \mathbf{e} \rangle$ follow Gaussian distribution of vectors and can be found using DGS. However, another result can be recovered [GJ21]:

$$x_j^T \cdot [\mathbf{A} \mid \mathbf{I_n}] \cdot \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} = x_j^T \cdot \mathbf{t} = [v^T | x_j^T] \cdot \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix}$$

As such, by finding a $[\mathbf{s}|\mathbf{e}]^T$ will allow for a pseudo-**s (s')** to be recovered in the 1st half before the mod q. This pseudo-**s** is independent of e, and in a sense, can convert the erfine LWE problem into a linear problem, akin to finding values of **As**. As such, by computing the difference in values of t when s = [1,0,…, 0], [0,1,…, 0]… will allow these "jumps" from solution to solution to be discovered. This can be called a search process.

Table 8.1: Example of Search Process

-	s = [1,1,1] e = [0,0,-1] [s'|e'] =[688, -298, -376, 987, -987, 0]	-
s = [0,0,1] e = [0,0,-1] [s'|e'] = [313, -136, -171, 448, -448, 0]	s = [0,1,1] e = [0,0,-1] [s'|e'] = [-50, 22, 27, -72, 72, 0]	s = [0,2,1] e = [0,0,-1] [s'|e'] = [-413, 179, 226, -592, 592, 0]
-	s = [-1,1,1] e = [0,0,-1] [s'|e'] = [65, -28, -35, 93, -93, 0]	-

By identifying the linear additions vertically and horizontally across the board, it will be possible to reverse engineer the value of **s**, with half the time required as the brute force algorithm.

This however, proved to be trivial to a small extent, as the permutations analysed are akin to running a brute force attack, just with longer leaps, and the $[\mathbf{s}|\mathbf{e}]^T$ obtained is extremely reliant on the values of $[v^T|x_j^T]$, which leads us back to a brute force attack. The only advantage this could have over a brute force attack is that **e** can be ignored, halving the worst-case number of permutations. This method was partially inspired by [Bor11], who looks at solving integer least squares (ILS) problems, and their impact on solving SVP. Since SVP is within a lattice, the ILS problem is all the more pertinent in finding solutions within a lattice space.

In the case of LWE and Kyber another type of lattice can be explored, a "q-ary" lattice, which can account for the mod *q* taken in Kyber. This could a future area to explore for future interns.

References:

[Aji98] Short Vector problem is NP-hard Link
[AKS01]
[ASP11] Metropolis Algorithm with Hasting's Generalisation for Solving SVP Link
[Bor11] Ordinary and Ellipsoid Integer Least Squares Problem Link
[Cra50] Cramer's Equations and Rules Link
[DWZ22] Faster Matrix Multiplication Link
[DZY17] nLLL Algorithm Link
[GJ21] Faster Dual Lattice Attacks Link
[Kan83]
[Kyber] Kyber reference Link
[LLL86] LLL Algorithm
[LPR13] On Ideal Lattices and LWE over Rings Link
[LPW12] Solving Hard Lattice Problems and the Security of Lattice-Based Cryptosystems Link
[LTJ22] PHLLL Algorithm Link
[LW21] Dual Lattice Attacks for CVP Link
[Mic05] Shortest Vector Problem Link
[SE94] DeepLLL Algorithm Link
[ZWQ11] Givens' Rotation in LLL Link