SCIFEST@COLLEGE

2025

Lattice-based Cryptography: A Python Approach to the Shortest Vector Problem.

Addison Carey

Celbridge Community School

Stand Number: 148

Project Report Book

Contents

Abstract (250 words)

Lattice-based cryptography relies on complex mathematical problems, notably the Shortest Vector Problem (SVP), which involves finding the shortest non-zero vector in a lattice. Understanding and visualizing the SVP is crucial for grasping the security foundations of these cryptographic systems.

My project aims to develop Python-based visualizations to illustrate the SVP, enhancing the comprehension of lattice structures and the inherent challenges in solving the SVP, demonstrating the strength and complexity of lattice-based cryptography.

Throughout my project, I have used various different experimental methods. I researched the mathematical foundations of lattices and the idea of the SVP including existing algorithms and their computational complexities. I developed a real implementation of these algorithms that approximate solutions to the SVP. Additionally, I created interactive visualizations to depict lattice structures and the process of identifying shorter vectors within these lattices. Lastly, I assessed the efficiency and accuracy of the implemented algorithms across various lattice configurations.

I started my project in **February 2025**, and have researched and gained insights to understanding the fundamentals of lattice structures and the SVP. I have explored existing resources to comprehend the mathematical intricacies of SVP. I have attempted to simulate basic lattice formations and apply simple vector operations using Python. Additionally, I have reviewed current visualization techniques to identify effective methods to representing high-dimensional lattice data.

Introduction

My interest in Computer Science and Coding goes back to early Primary school. I started using Scratch and Micro:Bit but soon transitioned onto HTML/CSS and JS. Python was my introduction to the power of computer code to manipulate data.

I have been working in the area of cryptographic protocols engaging with AI models within the last year, where I first began exploring cryptographic systems and the mathematics that underpin digital security. As I delved deeper into topics like modular arithmetic and number theory, I encountered lattice-based cryptography a powerful, mathematically complex branch of encryption that holds promise for the post-quantum era. This project allowed me to combine my interests in coding, cryptography, and mathematical problem-solving into one cohesive exploration of one of the most challenging computational problems: the Shortest Vector Problem (SVP).

Lattices are regular arrangements of points in n-dimensional space defined by a set of basis vectors. In cryptography, these structures are used to create encryption schemes that are believed to be secure even against quantum computers unlike classical systems such as RSA or ECC, which rely on factoring and discrete logarithms (Micciancio and Regev 2009). One of the key reasons for this strength lies in the hardness of SVP: the task of finding the shortest non-zero vector in a high-dimensional lattice, given a basis. This problem is known to be NP-hard under certain approximations and forms the foundation of many lattice-based cryptographic constructions such as Learning With Errors (LWE), NTRU, and Ring-LWE encryption.

To conceptualise the complexity of lattice cryptography, imagine you're blindfolded and dropped into a dense forest. You know the forest is arranged in a perfect invisible grid, but you're only allowed to move by combinations of the tree positions you're given, and you must find the tree closest to the origin. Some paths might take you far away before you loop back; others might seem short but don’t actually land on a tree. This is the essence of the Shortest Vector Problem: despite having a “map” (the basis), finding the shortest path to a non-zero point is not obvious, especially in high-dimensional space.

In this project, I focus on creating algorithms in Python that approximate solutions to the SVP, with a strong emphasis on performance and cryptographic insight. Specifically, my goal was to develop my own implementation, attempting to intelligently explore the lattice space using randomized sampling, pruning strategies, and local optimization. By combining experimental visualisation with performance testing over many randomly generated lattices, I aimed to uncover patterns, weaknesses, and practical strengths of my method in comparison to simpler brute-force techniques.

In the context of cryptography, SVP solvers simulate the types of attacks that an adversary might attempt when trying to break an encryption scheme. If we can build better solvers, we can better understand how resistant a cryptographic system truly is. This is particularly relevant as we transition to post-quantum cryptography, where lattice-based encryption is one of the most promising candidates for future security standards (Peikert 2016).

My hypothesis is that by applying a pruned, randomized algorithm combined with local refinement strategies, it is possible to approximate solutions to the Shortest Vector Problem more effectively than a naive brute-force method, particularly in high-dimensional or poorly conditioned lattices. I believe that such methods not only improve our ability to visualise and study lattice problems but can also contribute to the future of secure cryptographic systems by providing insight into the hardness assumptions on which these systems are based.

Explanations of Components and Terms

In this section of my report book, you will find explanations of each primary component of the Report Book. These Primary Components include Lattice-based Cryptography, The Shortest Vector Problem (SVP), and Encryption.

Lattice-based Cryptography

Lattice-based cryptography is a promising approach to secure communications in the post-quantum era. It leverages the mathematical structure of lattices to construct cryptographic primitives that are believed to be resistant to attacks from both classical and quantum computers (Micciancio and Regev 2009).

In simple terms, a lattice is a regular, repeating grid-like structure in space. In a two-dimensional space, a lattice might look like the points on graph paper where the horizontal and vertical lines meet. Mathematically, a lattice is defined as the set of points that can be formed by integer combinations of a fixed set of basis vectors, for example (O. Regev 2006):

$$L = \{a_1\mathbf{b}_1 + a_2\mathbf{b}_2 \mid a_1, a_2 \in \mathbb{Z}\}$$

Where:

- $\mathbf{b}_1$ and $\mathbf{b}_2$ are basis vectors,
- a_1 and a_2 are integer coefficients (any integer multiple of the basis vectors),
- The set L consists of all the points you can reach by adding integer multiples of $\mathbf{b}_1$ and $\mathbf{b}_2$.

For higher dimensions (like 3D), the concept extends similarly with more basis vectors. The security of lattice-based cryptographic schemes often relies on the hardness of certain computational problems, such as (Dachman-Soled 2020):

- **Shortest Vector Problem (SVP)**: Finding the shortest non-zero vector in a lattice.
- **Closest Vector Problem (CVP)**: Finding the lattice point closest to a given target point.

- **Learning With Errors (LWE)**: Solving a system of noisy linear equations, which is believed to be hard even for quantum computers.

These problems are considered computationally difficult, forming the basis for the security of lattice-based cryptographic systems.

Lattice-based cryptography provides several cryptographic primitives (basic building blocks) that are resistant to attacks by quantum computers. These primitives include (Dachman-Soled 2020):

1. **Public Key Encryption**: One of the most well-known lattice-based encryption schemes is NTRU, which provides public key encryption. It relies on the hardness of lattice problems like SVP and CVP to ensure security. The key idea is to encrypt a message in such a way that only the intended recipient, who possesses a special decryption key, can recover the original message.
2. **Digital Signatures**: SPHINCS+ and FALCON are lattice-based digital signature schemes. These signatures are used to verify the authenticity of messages. Lattice-based signatures are designed to be resistant to quantum computing threats. They involve algorithms that make it computationally difficult to forge a signature without the secret key (D. a. Micciancio 2015).
3. **Homomorphic Encryption**: Fully Homomorphic Encryption (FHE) is a type of encryption that allows computations on encrypted data without decrypting it. This is useful for privacy-preserving computations, such as in cloud computing, where data needs to remain encrypted while it's being processed. Lattice-based schemes, including Gentry's construction, provide a foundation for FHE.
4. **Key Exchange**: Lattice-based protocols like Kyber allow two parties to securely exchange cryptographic keys over an insecure channel. Kyber is considered secure against quantum attacks and is a potential candidate for future encryption standards (D. a. Micciancio 2015).

The security of these systems is fundamentally tied to the difficulty of solving the lattice problems mentioned above. The most widely used assumption is that the Learning

With Errors (LWE) problem is computationally hard (O. Regev 2006). Solving LWE is considered difficult even with the advent of quantum computers, making it a reliable foundation for cryptographic systems.

The general idea behind lattice-based security is that, even if an adversary has powerful computational resources, such as a quantum computer, they cannot solve these problems efficiently unless they possess some additional information (such as the secret key). The strength of lattice-based cryptography stems from the mathematical difficulty of these problems, and this makes it a good candidate for post-quantum cryptography (O. Regev 2005).

The reason this cryptography is a strong candidate for the post-quantum world is that it is believed to be immune to quantum attacks. Quantum computers will likely be able to break the security of systems based on integer factorization (RSA) or elliptic curve cryptography (ECC) using Shor's algorithm. However, lattice-based problems, including LWE and SVP, are not susceptible to these attacks (O. Regev 2006).

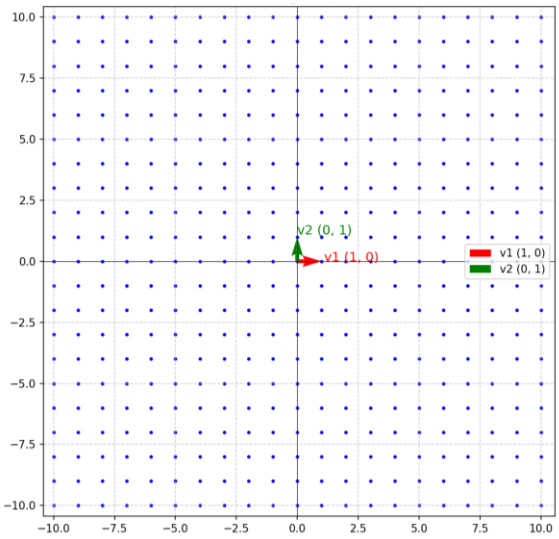
As a result, these schemes are being explored and adopted for standardization in quantum-safe cryptographic systems, with algorithms like Kyber for encryption and FALCON for signatures already being considered for post-quantum cryptographic standards by organizations like NIST (National Institute of Standards and Technology).

Overall, lattice-based cryptography provides a robust and secure foundation for encryption, digital signatures, and more, especially in a world transitioning to post-quantum security. Its resilience against quantum attacks, combined with its versatility in supporting various cryptographic protocols, makes it a key area of research for future-proof security systems (Peikert 2016).

Basic Interpretation of Lattice-based Cryptography:

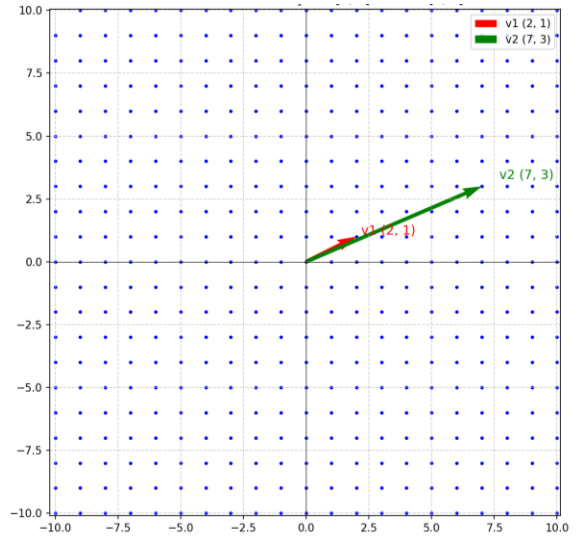
A lattice is a mathematical structure formed by a set of points in space that are regularly spaced in a grid-like pattern. These points are generated by taking integer linear combinations of a set of basis vectors. For example, in two dimensions, a lattice can be formed by vectors such as:

$$v_1 = [1,0] \text{ and } v_2 = [0,1]$$



Let's say the basis vectors are (1,0) in red, and (0,1) in green. These make the normal square lattice. But, what if we use these two basis vectors:

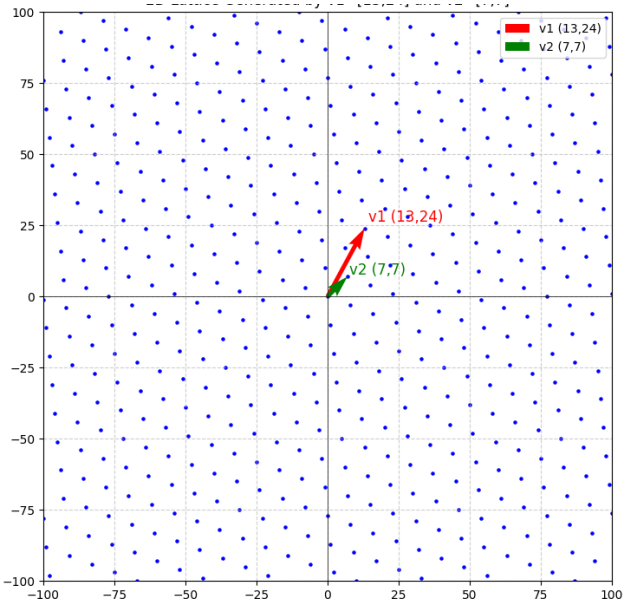
$$v_1 = [2,1] \text{ and } v_2 = [7,3]$$



At first they look to be creating points on a line, but if you start stacking together positive and negative vectors, we get other points, and eventually you'll end up with the same square lattice. This illustrates an important point: Two very different looking sets of basis vectors can generate the same lattice. The first example is called a ‘good’ basis, as it is easy to visualise the lattice with the basis vectors. Whereas the second example is called a ‘bad’ basis, where the vectors are almost parallel, making it difficult to visualize the lattice just by seeing the basis vectors alone. With solving the SVP, it is much easier with a good basis than a bad basis.

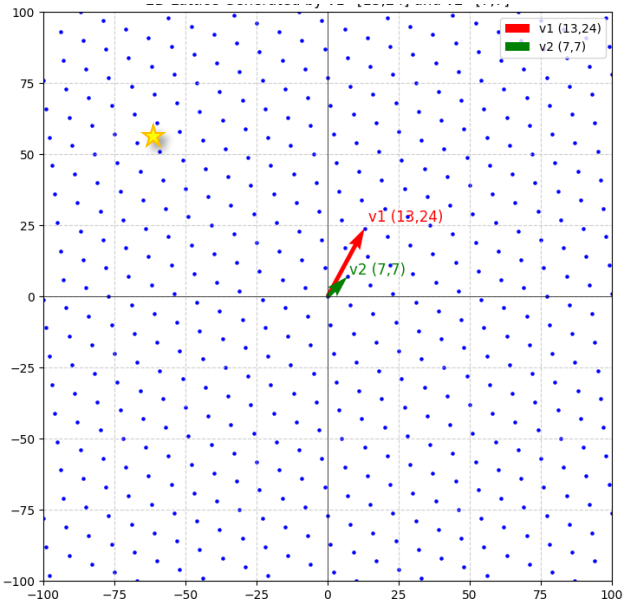
Lets take another lattice for example, with the vectors:

$$v_1 = [13,24] \text{ and } v_2 = [7,7]$$



The shortest vector problem asks, “which point on the lattice is closest, but not equal to, the origin (0,0)?” or in other words, which combination of the vectors is closest to (0,0) on the lattice. This problem implies adding and subtracting multiple basis vectors together to generate the closest point to the origin. With simpler lattices, trial and error can be used to try find these point, but when the number of dimensions increase this method becomes inefficient, as its possible that the optimal solution comes from adding and subtracting hundreds of red and green vectors together. There is more possibilities in higher dimensions and the shortest vector problem becomes even harder.

This concept is also very similar to the closest vector problem, which asks "find the lattice point which is closest to a particular point in space." For example, which point on the lattice generated by the two vectors from before is closest to this point on the lattice:



These belong to a collection of related problems called "lattice problems", which are believed to be hard problems, especially in hundreds of dimensions. It is believed that these could be the mathematical basis for a quantum-secure cryptographic protocol.

Lattice-based cryptography offers a robust framework for secure communication, leveraging the mathematical complexity of lattices. Its resistance to quantum attacks positions it as a vital component in the future of cryptographic security.

The Shortest Vector Problem (SVP)

The Shortest Vector Problem (SVP) is a computational problem associated with lattices. Given a lattice defined by a basis $B = \{b_1, b_2, \dots, b_n\}$, the goal of the SVP is to find the shortest non-zero vector in the lattice, i.e., the vector v in L with the minimum Euclidean norm $||v||$ (M. Ajtai 1996).

Formally, the problem is defined as:

$$\text{SVP: } \min_{v \in L\{0\}} ||v||$$

Where:

- L is the lattice formed by integer linear combinations of the basis vectors.
- $||v||$ is the Euclidean norm (length) of the vector v.

The SVP is considered a computationally hard problem. Specifically, it is known to be NP-hard in general. This means that no efficient algorithm (in polynomial time) is currently known to solve the problem in all cases. Additionally, solving SVP is widely believed to be hard even under the assumption of quantum computing, which makes it a strong candidate for cryptographic systems that are resistant to quantum attacks (D. Micciancio 2004).

One of the central aspects of SVP is that it has no known efficient algorithm for finding the shortest vector for high-dimensional lattices, which is why it is a cornerstone of lattice-based cryptography (Goldwasser and Bellare 2001).

Lattice-based cryptography, using problems like the SVP, offers a number of practical applications:

- **Post-Quantum Cryptography**: Lattice-based cryptographic systems are considered to be post-quantum secure. This means they are resistant to attacks by quantum computers, which are expected to be able to break many traditional cryptographic systems like RSA and ECC. For example, **NTRU** is a lattice-based encryption scheme that is believed to be secure even in the presence of quantum computers.
- **Secure Multiparty Computation**: Lattice-based cryptographic protocols can be used for secure computation where multiple parties compute a function on private inputs without revealing their inputs to each other. The difficulty of solving SVP ensures that the protocols remain secure.
- **Homomorphic Encryption**: As mentioned earlier, SVP plays a role in the security of lattice-based fully homomorphic encryption (FHE), which has applications in

secure cloud computing, privacy-preserving data analysis, and secure voting systems (Gentry 2009).

- **Digital Signatures and Authentication**: Lattice-based digital signature schemes can be used for secure authentication systems. Because SVP is hard to solve, it makes forging a signature difficult for an attacker.

The difficulty of solving SVP stems from the combinatorial nature of the problem. Finding the shortest vector in a high-dimensional lattice is akin to searching through an exponentially large set of possible integer combinations, making it computationally infeasible for large dimensions.

Some reasons for its hardness include:

- High Dimensionality: As the dimension of the lattice increases, the search space for the shortest vector grows exponentially, which makes the problem intractable for large n.
- Geometric Properties: Lattices in higher dimensions exhibit complex geometric structures, where the shortest vector can lie in highly convoluted regions of space, making it hard to find efficiently.

In conclusion, The SVP plays a pivotal role in lattice-based cryptography. It serves as the foundation for the security of many cryptographic schemes, particularly those designed to be secure against quantum computers. The hardness of SVP makes it a reliable building block for post-quantum cryptographic primitives, including encryption schemes, digital signatures, and homomorphic encryption. The computational difficulty of solving SVP, even in higher dimensions, ensures that lattice-based cryptography remains secure in the face of advanced computational threats.

Encryption

Encryption is the process of converting normal data or plaintext to something incomprehensible or cipher-text by applying mathematical transformations or formulae. These mathematical transformations or formulae used for encryption processes are called algorithms (Bhanot and Rahul 2015). Only authorized parties who have the correct key can decrypt the ciphertext back into plaintext.

There are two primary types of encryption:

1. Symmetric Encryption: The same key is used for both encryption and decryption. It's fast and suitable for large amounts of data, but the key distribution process can be challenging since both parties need access to the same key. Common symmetric encryption algorithms include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).
2. Asymmetric Encryption (Public-Key Encryption): Uses two keys: a public key (for encryption) and a private key (for decryption). The public key is openly shared, while the private key is kept secret. This method is slower but very secure and eliminates the need for both parties to have the same key. Examples include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography)

Encryption algorithms take plaintext and process it with a cryptographic key to produce ciphertext. The security of encrypted data relies on the strength of the algorithm and the secrecy of the key. Encryption ensures confidentiality, meaning that only authorized individuals or systems can access the protected information. It's widely used in various applications, like securing online transactions, protecting emails, and safeguarding sensitive data stored on devices.

The Mathematical Aspects of Lattice Cryptography and the SVP

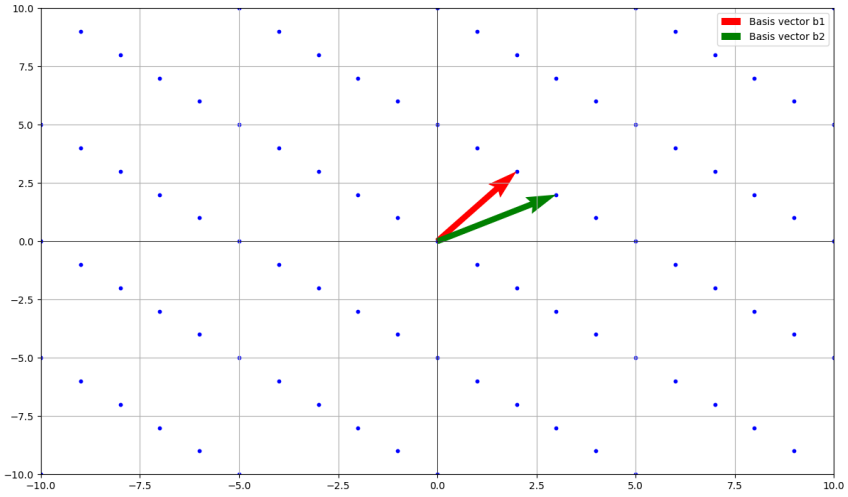
Lattice-based cryptography is an area of modern cryptography that leverages the hardness of problems related to lattice structures. These lattices are geometric objects in multi-dimensional space that form the backbone of many cryptographic protocols. The SVP is one of the fundamental problems in lattice-based cryptography. This section explores the mathematical foundations of lattices, the SVP, and how they are used in cryptographic schemes, particularly focusing on the methods applied in solving the SVP.

1. Lattices in Mathematics

A lattice $\mathcal{L}$ is a discrete subset of a vector space $\mathbb{R}^n$ formed by all integer linear combinations of a set of linearly independent vectors, called a basis. More formally, if $B = \{b_1, b_2, \dots, b_n\}$ is a set of linearly independent vectors in $\mathbb{R}^n$, then the lattice generated by B is:

$$\mathcal{L}(B) = \left\{ \sum_{i=1}^{n} \alpha_i b_i \mid \alpha_i \in \mathbb{Z} \right\}$$

where α_i are integer coefficients. The set B is referred to as the basis of the lattice, and the lattice itself is an infinite set of points in $\mathbb{R}^n$, arranged in a periodic grid.



A lattice is geometrically structured, with the basis vectors defining the fundamental unit cells of the lattice. The complexity of lattice-based problems, such as SVP, arises from the fact that the vectors in the lattice can grow exponentially in higher dimensions, making the problems computationally challenging.

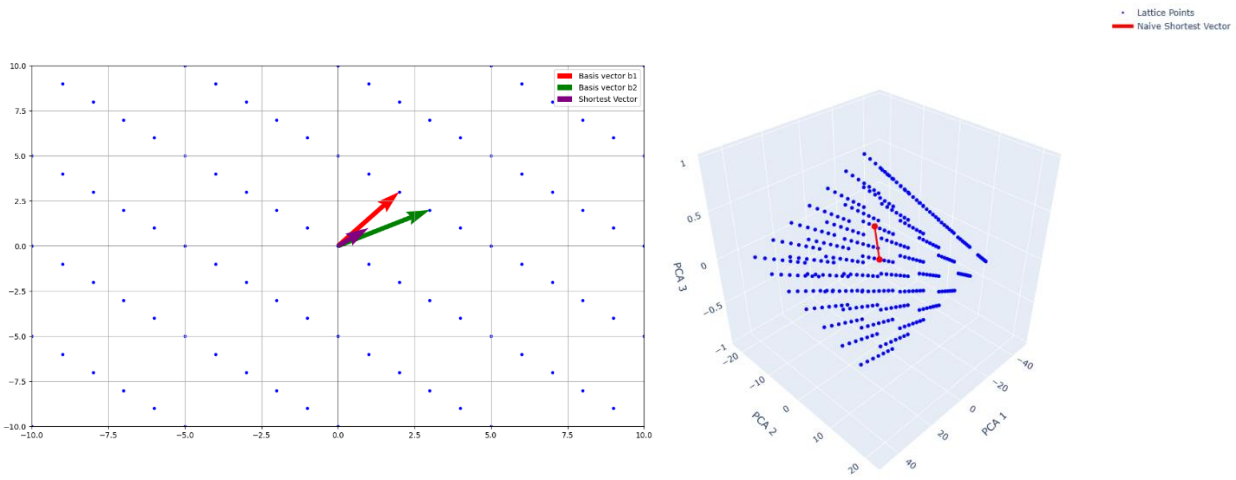
2. The Shortest Vector Problem (SVP)

The Shortest Vector Problem (SVP) is one of the most important problems in lattice-based cryptography. In its simplest form, SVP asks for the shortest nonzero vector in a lattice $\mathcal{L}(B)$ given a basis B.

Formally, the SVP can be stated as follows:

$$\text{Find } v \in \mathcal{L}(B), \text{such that } \left|\left|v\right|\right| = \min_{v' \in \mathcal{L}(B), v' \neq 0} ||v'||$$

where $\|\cdot\|$ represents a norm, typically the Euclidean norm $||v||_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$ for a vector $v = (v_1, v_2, \ldots, v_n)$. The objective is to find the shortest vector in terms of the Euclidean distance from the origin.



The difficulty of SVP lies in the fact that lattice vectors can be arbitrarily large in higher dimensions. Furthermore, SVP is NP-hard in general, meaning there is no known polynomial-time algorithm to solve it efficiently for arbitrary lattices. It is one of the problems assumed to be hard in the context of lattice-based cryptography.

3. Relationship Between Lattice Cryptography and SVP

Lattice-based cryptography relies heavily on the hardness of lattice problems such as SVP. The security of many cryptographic protocols, such as those used in post-quantum cryptography, is based on the assumption that solving SVP is computationally infeasible.

For example, the security of Learning With Errors (LWE), a popular lattice-based encryption scheme, depends on the difficulty of solving problems similar to SVP. In LWE, an adversary must solve a version of the SVP to recover secret information from noisy lattice-based data, which is assumed to be computationally infeasible (Oded 2009).

Another important example is the NTRU encryption scheme, which uses lattice-based structures to secure data. The key generation process involves selecting a random lattice, and the encryption and decryption procedures rely on the hardness of finding short vectors in that lattice, again connecting to the SVP (Ajtai, et al. 2001).

4. Methods for Solving SVP: Naive Brute Force and Randomized Approaches

There are several methods for solving the SVP, each with its own trade-offs in terms of efficiency and correctness. For my methodology, I employed both a naive brute-force approach and a randomized greedy algorithm.

4.1 Naive Brute-Force Approach

The naive brute-force method for solving the SVP involves iterating over all possible integer combinations of the lattice's basis vectors within a given search range. For a lattice $\mathcal{L}(B)$ with basis B, the approach tests all integer coefficients $\alpha_i$for each basis vector b_i, calculating the resulting lattice vector and its norm:

$$v = \sum_{i=1}^{n} \alpha_i b_i$$

The algorithm then checks the norm of each vector v and keeps track of the shortest one found. This is an exhaustive search and can be computationally expensive, especially as

the dimension n of the lattice increases. The time complexity of this approach is exponential in the number of dimensions, making it impractical for large lattices.

4.2 Randomized Greedy Algorithm

My improved randomized greedy algorithm is an optimization over the brute-force approach. Instead of searching all combinations exhaustively, it randomly samples possible integer coefficients α_i for the basis vectors, evaluating the norm of the resulting vector. This approach attempts to focus the search around promising solutions and prunes the search space by discarding unpromising candidates.

At each step, the algorithm selects a new candidate vector based on the current "radius" of the search, gradually shrinking the search space as better solutions are found. This heuristic approach can often find good solutions faster than brute force, especially in high-dimensional lattices, but it does not guarantee finding the optimal solution.

Mathematically, the process can be described as follows. Starting with an initial radius r, the algorithm samples random integer coefficients α_i from a fixed range and evaluates the vector:

$$v = \sum_{i=1}^{n} \alpha_i b_i$$

If the norm $||v||$ is smaller than the current best norm, the solution is updated, and the search radius is reduced to encourage further exploration near the current best solution.

5. The Role of Basis Conditioning

The conditioning of a lattice basis B affects the difficulty of solving SVP. A poorly conditioned basis, where the basis vectors have very different lengths or are nearly linearly dependent, makes it harder to find short vectors. The condition number of the basis is given by the ratio of the largest to the smallest eigenvalue of $B^T B$, i.e.,

$$\text{Condition number} = \frac{\lambda_{\max}(B^T B)}{\lambda_{\min}(B^T B)}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of the matrix $B^T B$, respectively. A high condition number indicates a poorly conditioned basis, making the SVP harder to solve efficiently.

In my method, the basis condition score is computed to assess the quality of the lattice basis. A well-conditioned basis (low condition number) makes the SVP easier to solve, whereas a poorly conditioned basis (high condition number) increases the difficulty.

Overall, Lattice-based cryptography relies on the inherent difficulty of lattice problems, such as the Shortest Vector Problem, to ensure security. The SVP is a key problem, and while various methods for solving it exist, the brute-force and pruned randomized greedy approaches used in my project provide insight into the mathematical complexity and challenges of lattice-based problems. The relationship between lattice structure, basis conditioning, and the SVP is central to understanding the hardness of lattice-based cryptography and its suitability for post-quantum cryptographic systems (Babai 1986).

My Code / Experimental Methods

In this section, you will find a breakdown of the algorithms used in my code to create my algorithm and implementing it within the lattcie, and mathematic aspects of my developed algorithm predicting the SVP against other algorithms.

I decided to create my project using the principles of User-centred design (UCD). This is a design approach that focuses on creating products, services, and experiences that are centred around the needs and desires of the people who will use them. To achieve this, UCD relies on a set of principles that guide the design process. These principles include:

- Empathy: Understanding the needs, goals, and motivations of the users and putting oneself in their shoes.
- Collaboration: Involving users in the design process and collaborating with them to co-create solutions.
- Iteration: Prototyping and testing ideas quickly and repeatedly to refine and improve the design.
- Inclusivity: Designing for the diverse needs and abilities of all potential users.
- Accessibility: Creating products and services that are accessible and usable by people with disabilities.
- User testing: Gathering feedback from users and using it to inform and improve the design.

I decided to apply UCD using the Agile Framework. Agile methodology is a project management approach that enables flexibility and empowers practitioners to engage in rapid iteration. It is based on the Agile Manifesto (Fowler 2001), a set of values and principles for software development that prioritize the needs of the customer, the ability to respond to change, and the importance of delivering working software regularly. Agile methodologies, such as Scrum and Lean, are designed to help teams deliver high-quality products in a fast-paced and dynamic environment

by breaking down complex projects into smaller, more manageable chunks and continually reassessing and adjusting the project plan as needed. Agile teams are typically self-organizing and cross-functional and rely on regular communication and collaboration to achieve their goals.

While agile is best used in a team setting, it can be adapted to use by individual developers. I managed my Agile system using Kanban (Fig.1)

Fig.1

Kanban is a project management method that originated in Japan in the 1950s and is based on the principles of lean manufacturing. It is designed to help teams visualize and optimize their workflows by breaking down tasks into smaller, more manageable chunks and tracking their progress through a series of stages. Kanban uses a visual board to represent the distinct stages of a project, with cards representing individual tasks and columns representing the various stages of the workflow. The goal of Kanban is to increase efficiency and transparency by making it easier for team members to see where work is in the process, identify bottlenecks and inefficiencies, and adjust as needed. Kanban is often used in conjunction with other agile methodologies, such as Scrum, to help teams manage their work more effectively. (Corona 2013). Please scan here for my full Kanban journey.

Iteration 1:

Planning Phase

At the start of the project, my goal was to implement a basic SVP solver that could:

- Brute-force through possible lattice vectors.
- Identify the shortest non-zero vector.
- Confirm that the solver correctly solved SVP in small-dimensional lattices.

This initial version would serve as a baseline for comparison against later, more optimized algorithms. Since SVP is extremely hard for large lattices, I planned to focus on small dimensions (3D and 4D lattices) to keep the brute-force search computationally manageable.

Design Phase

I found that mapping out the data flow before I coded it, helped me to plan for any tricky coding elements such as where I would need to put a conditional statement or a function.

Development Phase – Model Development Steps

Below is a detailed explanation of the sections of my code. This code implements a basic approach to solving the SVP in a random 3D lattice.

1. Imports

```
1  #imports AC
2  import numpy as np
3  from sklearn.decomposition import PCA
4  import plotly.graph_objs as go
5
```

Fig.2

The numpy package is used for numerical computing (line 2). It is used for creating arrays, matrix operations, and linear algebra. PCA (Principal Component Analysis) is used here for dimensionality reduction (line 3). It's useful when the lattice data has more than three dimensions, but you want to visualize it in 3D. Plotly is a library for creating interactive plots (line 4). It is used to visualize the lattice and the shortest vector in 3D space (Fig. 2).

2. Functions

```
6   #functions AC
7   #simple measure of basis conditioning AC
8   def basis_condition_score(B):
9       B_T_B = B.T @ B
10      eigvals = np.linalg.eigvals(B_T_B)
11      return np.max(eigvals) / np.min(eigvals)
```

Fig.3

This function calculates a score to measure the "conditioning" of the basis matrix B (line 8-11). The condition number of a matrix is a measure of how sensitive the solution of a system of linear equations is to changes in the input. A high condition number indicates a poorly conditioned matrix (i.e., numerical instability). The score is the ratio of the largest to the smallest eigenvalue. A larger ratio means the matrix is poorly conditioned (Fig. 3).

```
13  #naive brute-force svp solver AC
14  def naive_svp_solver(B, max_range=5):
15      n = B.shape[1]
16      shortest_vec = None
17      min_norm = float('inf')
18
19  #search all integer coefficient vectors AC
20      for coeffs in np.ndindex(*(2*max_range+1 for _ in range(n))):
21          coeffs = np.array(coeffs) - max_range
22          if np.all(coeffs == 0):
23              continue  #skip the zero vector AC
24          vec = B @ coeffs
25          norm = np.linalg.norm(vec)
26          if norm < min_norm:
27              min_norm = norm
28              shortest_vec = vec
29              best_coeffs = coeffs
30      return shortest_vec, best_coeffs, min_norm
```

Fig.4

This function implements a naive brute-force approach to find the shortest vector in the lattice, solving the SVP (line 14-30). *B* is the basis matrix of the lattice. The goal is to find the shortest non-zero vector that can be formed by integer combinations of the columns of *B*. The function searches for all integer coefficient vectors in the range *[-max_range, max_range]* for each dimension (*n* is the dimension of the lattice) (line 20). For each combination of coefficients, it computes the vector *vec = B @ coeffs* and calculates its norm (length) (line 24). It keeps track of the shortest vector found during this search by comparing the norms (Fig. 4).

3. Lattice Setup

```
32  #lattice setup AC
33  n = 3  #dimensions
34  B = np.random.randint(-5, 6, size=(n, n))  #random 3D lattice AC
35
36  print("Basis matrix B:\n", B)
```

Fig.5

This initializes a random 3D lattice by generating a matrix *B*. The variable *n = 3* specifies

a 3D lattice (three dimensions) (line 33). *np.random.randint(-5, 6, size=(n, n))* generates a 3x3 matrix where each element is an integer between -5 and 5 (line 34). This matrix represents the basis of the lattice (Fig. 5).

4. Basis Quality

```
38  #basis quality AC
39  score = basis_condition_score(B)
40  print(f"Basis condition score (higher = worse): {score:.2f}")
```

Fig.6

This computes and prints the condition score of the lattice basis (line 39). The condition score is used to assess how well-conditioned the lattice is. Higher scores indicate that the lattice is poorly conditioned (Fig. 6).

5. Solve SVP

```
42  #solve SVP AC
43  shortest_vec, best_coeffs, shortest_norm = naive_svp_solver(B, max_range=5)
44
45  print("\n--- Shortest Vector Found ---")
46  print("Shortest vector:", shortest_vec)
47  print("Coefficients:", best_coeffs)
48  print(f"Vector length (norm): {shortest_norm:.3f}")
```

Fig.7

This calls the *naive_svp_solver* function to find the shortest vector in the lattice. The function returns the shortest vector, the coefficients used to form it, and its norm (length). It then prints these results (Fig. 7).

6. Visualization: Lattice and Shortest Vector

```
50  #visualization of lattice and SV AC
51  #generate lattice points for plotting AC
52  sample_vectors = []
53  for coeffs in np.ndindex(*(range(-5, 6) for _ in range(n))):
54      coeffs = np.array(coeffs)
55      sample_vectors.append(B @ coeffs)
56
57  sample_vectors = np.array(sample_vectors)
```

Fig.8

This generates points representing the lattice to plot them later. The code iterates over all possible coefficient combinations in the range [-5, 5], computes the corresponding lattice points, and stores them in *sample_vectors* (Fig. 8).

```
59  #reduce dimension if needed AC
60  pca_3d = PCA(n_components=3)
61  data_3d = pca_3d.fit_transform(sample_vectors)
62
63  shortest_vec_proj = pca_3d.transform(shortest_vec.reshape(1, -1))
```

Fig.9

This reduces the dimensionality of the lattice points for visualization (line 60). PCA is applied to the lattice points to project them into 3D space (line 61). The shortest vector is also projected into 3D using the same PCA transformation (line 63) (Fig. 9).

```
65  #plot lattice and SVP AC
66  fig = go.Figure()
67
68  #lattice points on plot AC
69  fig.add_trace(go.Scatter3d(
70      x=data_3d[:, 0], y=data_3d[:, 1], z=data_3d[:, 2],
71      mode='markers',
72      marker=dict(size=3, color='blue'),
73      name='Lattice Points'
74  ))
75
76  #shortest vector on plot AC
77  fig.add_trace(go.Scatter3d(
78      x=[0, shortest_vec_proj[0, 0]],
79      y=[0, shortest_vec_proj[0, 1]],
80      z=[0, shortest_vec_proj[0, 2]],
81      mode='lines+markers',
82      marker=dict(size=5, color='red'),
83      line=dict(width=5, color='red'),
84      name='Shortest Vector'
85  ))
86
87  fig.update_layout(title='SVP Approximation in Random 3D Lattice',
88                    scene=dict(xaxis_title='PCA 1',
89                               yaxis_title='PCA 2',
90                               zaxis_title='PCA 3'),
91                    width=800, height=800)
92
93  fig.show()
```

Fig.10

Lastly, this creates a 3D plot showing the lattice points and the shortest vector. A 3D scatter plot is created using Plotly, where the lattice points are shown as blue markers (line 69-74). The shortest vector is plotted as a red line from the origin (0, 0, 0) to the projected coordinates of the shortest vector (line 77-85) (Fig. 10).

Overall, my code demonstrates a basic approach to solving the SVP in a 3D lattice using a brute-force search. It includes many features, such as lattice creation using a random basis matrix, conditioning of the lattice basis to assess its quality, a naive SVP solver to find the shortest vector using integer coefficients, and visualization of the lattice points and the shortest vector in 3D.

Testing Phase

During this iteration, testing was exctremely important in order to tackle any issues before moving on. I tested the Naive SVP Solver by applying it to:

- Randomly generated 3D and 4D lattice bases.
- Visualizing the resulting lattice points using PCA and Plotly 3D plots.
- Executing the shortest vector, its coefficients, and its norm.

```
Basis matrix B:
 [[ 4 -2 -3]
 [-2 -3 -5]
 [-1 -3 -2]]
Basis condition score (higher = worse): 32.20

--- Shortest Vector Found ---
Shortest vector: [-1 -2  1]
Coefficients: [ 0 -1  1]
Vector length (norm): 2.449
```

Fig.11

From this, The solver correctly found a shortest nonzero vector. Results were consistent across different random lattices (Fig. 11).

The brute-force approach works well for small lattices (like the 3D case used here), but it would be inefficient for larger lattices. Testing with higher dimensions (e.g., $n = 4$ or $n = 5$) showed the method's limitations in terms of computation time. The condition score gives valuable insight into the quality of the lattice. Lattices with higher condition scores (i.e., poorly conditioned) made the SVP problem more challenging, confirming the importance of this metric in lattice analysis.

This testing highlights both the functionality of the algorithm and its limitations, providing a strong foundation for further improvements (such as more efficient algorithms for larger lattices). Overall, during iteration 1 I successfully validated a basic SVP solving method, establishing a solid starting point for further algorithmic improvements.

Iteration 2:

Planning Phase

After successfully developing a basic brute-force solver for SVP, I planned to improve my algorithm to address its limitations:

- Scalability: The naive solver became extremely slow as dimension n increased, due to exponential search growth.
- Efficiency: In many cases, brute-force checked thousands of unnecessary vectors that were clearly too long.

My plan for Iteration 2 was to create a more intelligent algorithm that could:

- Randomly sample lattice vectors instead of exhaustively enumerating them.
- Focus search on promising short vectors using adaptive pruning.
- Adaptively shrink the search space radius during the search to become more efficient over time.

This would allow me to find short lattice vectors faster and more reliably, especially in higher dimensions.

Design Phase

There were many elements I wanted to implement into the design for the Randomized Greedy SVP Solver, including:

1. Randomly generate integer coefficient vectors (small coefficients only).
2. Multiply each coefficient vector by the lattice basis B to create a lattice point.
3. Compute the norm (length) of each lattice point.
4. If a vector is within the current search radius, consider it.
5. If a shorter vector is found, update the best solution and slightly shrink the radius.
6. Repeat the process for a fixed number of trials (e.g., 5000 samples).

By focusing more heavily on vectors already showing promise, the search would adaptively 'zoom in' on the areas of the lattice where shorter vectors likely exist.

Development Phase

In Iteration 2, I expanded my project to not only build my Randomized Greedy SVP Solver, but to also generate random 4D lattices, compare my algorithm to the native brute-force solver, measure time and visualising the lattices with SVP. The new structure of the code is explained below:

1. My algorithm interpretation:

```
33  def pruned_randomized_greedy_svp(B, max_trials=5000, initial_radius=10.0):
34  #randomiaed SVP solver AC
35      n = B.shape[1]
36      best_vec = None
37      best_norm = float('inf')
38      radius = initial_radius
39
40      for trial in range(max_trials):
41          coeffs = np.random.randint(-3, 4, size=n)
42          if np.all(coeffs == 0):
43              continue
44
45          vec = B @ coeffs
46          norm = np.linalg.norm(vec)
47
48          if norm < radius:
49              if norm < best_norm:
50                  best_norm = norm
51                  best_vec = vec
52                  best_coeffs = coeffs
53                  radius = norm * 1.2  # shrink search focus AC
54
55      return best_vec, best_coeffs, best_norm
```

Fig.12

Here I introduced a new method for solving the SVP (Fig. 12). The dimension of the lattice n is determined (line 35), which is needed to randomly generate coefficient vectors of the correct size. Placeholders for the greedy search are set up which allows dynamic search space control (line 36-38). The random search process is repeated a fixed number of times (line 40). Then a vector of small integers (between -3 and 3) are randomly generated, as small coefficients are more likely to produce shorter vectors in the lattice (line 41). The lattice vector is generated by multiplying the basis matric with the coefficients, as every lattice point is a combination of the basis vectors (line 45). The Euclidean norm of the lattice vector is calculated (line 46). Only vectors within the current allowed radius are considered to ignore obviously large vectors to speed up search (line 48). If the current vector is better than the best so far, various variables update (*best_norn, best_vec and best_coeffs*), and it shrinks the search radius norm (line 49-53). This focuses

future trials more tightly around better candidates. After all trials complete, it returns the best lattice vector found (line 55).

2. Lattice Setup:

```
57  #lattice setup AC
58  n = 4  #4D lattice AC
59  B = np.random.randint(-5, 6, size=(n, n))
60
61  print("Basis matrix B:\n", B)
```

Fig.13

A new 4D lattice is randomly generated for each test run (line 58). The values are chosen between -5 and 5 (line 59). Keeping the lattices random tests the algorithm across a broad range of difficulties (Fig. 13).

3. Solve SVP with both Algorithms:

```
67  #solve with naive solver AC
68  start_time = time.time()
69  shortest_naive, coeffs_naive, norm_naive = naive_svp_solver(B, max_range=2)
70  time_naive = time.time() - start_time
71
72  #solve with my algorithm AC
73  start_time = time.time()
74  shortest_unique, coeffs_unique, norm_unique = pruned_randomized_greedy_svp(B, max_trials=5000, initial_radius=10.0)
75  time_unique = time.time() - start_time
```

Fig.14

This solves the same lattice using both algorithms, my algorithm (line 73-75) and the native brute-force (line 68-70). It also measures and executes the time taken for each solver (line 68,73). This setup allows direct experimental comparison between the two (Fig. 14).

4. Lattice Point Generation for visualisation:

```
93  sample_vectors = []
94  max_range = 3  # larger plot range AC
95
96  for coeffs in np.ndindex(*((2*max_range+1,) * n)):
97      coeffs = np.array(coeffs) - max_range
98      sample_vectors.append(B @ coeffs)
99
100 sample_vectors = np.array(sample_vectors)
```

Fig.15

This systematically generates all lattice points within a small range. This prepares the data for visualising the full lattice structure, including SVPs (Fig. 15).

5. Dimensionality Reduction with PCA:

```
102  #reduce dimension to 3D for plotting
103  pca_3d = PCA(n_components=3)
104  data_3d = pca_3d.fit_transform(sample_vectors)
105
106  shortest_naive_proj = pca_3d.transform(shortest_naive.reshape(1, -1))
107  shortest_unique_proj = pca_3d.transform(shortest_unique.reshape(1, -1))
```

Fig.16

This projects the original 4D lattice down to 3D using PCA (line 103-107). This is done because we cannot visualize 4D structures on a plot, so PCA allows easy 3D interactive plots while preserving as much structure as possible. This maintains meaningful relationships while making the data plottable (Fig. 16).

6. 3D Plotting with Plotly:

```
109  #plot AC
110  fig = go.Figure()
111
112  # Lattice points
113  fig.add_trace(go.Scatter3d(
114      x=data_3d[:, 0], y=data_3d[:, 1], z=data_3d[:, 2],
115      mode='markers', marker=dict(size=3, color='blue'),
116      name='Lattice Points'
117  ))
118
119  #naive shortest vector AC
120  fig.add_trace(go.Scatter3d(
121      x=[0, shortest_naive_proj[0, 0]],
122      y=[0, shortest_naive_proj[0, 1]],
123      z=[0, shortest_naive_proj[0, 2]],
124      mode='lines+markers',
125      marker=dict(size=5, color='red'),
126      line=dict(width=5, color='red'),
127      name='Naive Shortest Vector'
128  ))
129
130  #unique shortest vector
131  fig.add_trace(go.Scatter3d(
132      x=[0, shortest_unique_proj[0, 0]],
133      y=[0, shortest_unique_proj[0, 1]],
134      z=[0, shortest_unique_proj[0, 2]],
135      mode='lines+markers',
136      marker=dict(size=5, color='green'),
137      line=dict(width=5, color='green'),
138      name='Your Shortest Vector (Pruned Randomized Search)'
139  ))
140
141  fig.update_layout(title='SVP Solvers Comparison in Random 4D Lattice (PCA projected)',
142                    scene=dict(xaxis_title='PCA 1',
143                               yaxis_title='PCA 2',
144                               zaxis_title='PCA 3'),
145                    width=900, height=900)
146
147  fig.show()
```

Fig.17

This plots various elements on the 3D plot including:

- All lattice points (in blue) (line 113-117)
- Native-brute force SVP (in red) (line 120-128)
- My SVP (in green) (line 131-139)

This aids in visually comparing the solutions of the SVPs. It also makes the algorithm behaviour easier to understand. The enhanced visualisation using 3D plots also adds clarity and polish to the SVP (Fig. 17).

Testing phase

I tested my version of the algorithm on the same 4D lattices used for the naïve solver experiments. Below are some examples from the code:

```
Basis matrix B:
 [[ 3  5  4  1]
 [ 3  5  5  3]
 [ 2  1  0 -5]
 [-4 -5  4  3]]
Basis condition score (higher = worse): 1627.62

--- Naive Brute-Force Solver (limited search) ---
Shortest vector: [ 2  2 -1 -1]
Coefficients: [-1  1  0  0]
Norm length: 3.162
Time taken: 0.016 seconds

--- Pruned Randomized Greedy Solver (Your Algorithm) ---
Shortest vector: [ 0  2 -1  1]
Coefficients: [ 3 -2  0  1]
Norm length: 2.449
Time taken: 0.090 seconds
```

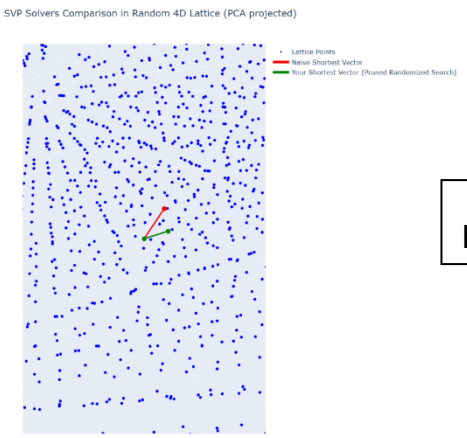


Fig.18

```
Basis matrix B:
 [[ 5 -4 -5 -5]
 [ 5  1 -4  0]
 [ 1  1  0 -1]
 [-2  4  2 -1]]
Basis condition score (higher = worse): 387.70

--- Naive Brute-Force Solver (limited search) ---
Shortest vector: [ 0 -1 -1  0]
Coefficients: [-1  0 -1  0]
Norm length: 1.414
Time taken: 0.015 seconds

--- Pruned Randomized Greedy Solver (Your Algorithm) ---
Shortest vector: [0 1 1 0]
Coefficients: [1 0 1 0]
Norm length: 1.414
Time taken: 0.152 seconds
```

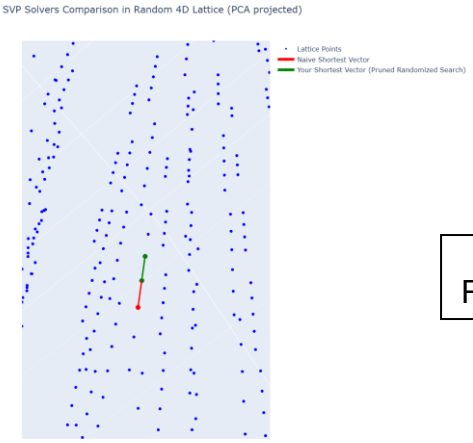


Fig.19

In the first example (Fig.18), you can see that while the basis condition score is higher, my algorithm actually finds a shorter vector than the native brute-force algorithm, showing how mine works to a higher standard and efficiency.

In the second example (Fig. 19), Both algorithms actually find the same vector, but the points are in different areas of the lattice. This is normal for there to be two different shortest vectors in the same lattice but in different locations relevant to the origin, depending on the basis conditions.

The only major problem I noticed from testing the algorithms were timing issues. In 4D, it runs quite quick with my algorithm taking slightly longer to find the SVP. Although when the dimensions is increased to n=5 or higher, it exponentially increases the amount of time taken. This was expected when I started creating my algorithm, and I took this into account later on.

My randomized greedy solver proved that intelligent random search combined with adaptive pruning can create a high-performance SVP solver, especially useful for dimensions where brute-force becomes impossible. Thus, Iteration 2 successfully achieved the goal of creating a scalable, effective, and unique SVP solving method suitable for cryptographic analysis and visualization.

Iteration 3:

Planning phase

Having confirmed that my Randomized Greedy Solver from the previous iteration could consistently solve SVP efficiently for 4D lattices, I wanted to further enhance its accuracy and reliability.

Even though my algorithm performed extremely well, there were still a few flaws:

- It relied heavily on random sampling.
- Even with adaptive pruning, it sometimes missed optimal vectors due to search noise.
- It made no attempt to refine its best guess after finding it.

For this iteration, my objective was to improve the solver's performance by introducing a local refinement stage:

- After the randomized search finds a promising short vector, I would apply a local search to explore 'nearby' vectors.
- This mimics a 'hill climbing' technique: start with a good solution, explore its neighbours, and keep improving.

This change would give the algorithm an additional boost, especially on difficult lattices while still remaining much faster than naive brute-force.

Design phase

The solver in Iteration 3 would follow a two-stage hybrid design:

Stage 1: Randomized Greedy Sampling (same as Iteration 2)

- Randomly generate small integer coefficient vectors.
- Check if they produce short lattice vectors (within a dynamic radius).
- Track the shortest vector found.

Stage 2: Local Search Refinement

- Once the best vector is found, generate all coefficient vectors in a small neighbourhood around it (e.g., all ±1 variations).
- Convert these nearby vectors into lattice vectors.
- Compare their lengths to the original vector.
- Keep improving the solution if any neighbour is shorter.

This introduces a new type of optimization of exploitation (refining a good vector), not just exploration (random search).

Development phase

While most of the previous code remained the same, there were various adjustments I made to refine my algorithm following the plan and design elements I created:

1. My updated algorithm:

```
33  def pruned_randomized_greedy_svp_with_local_search(B, max_trials=5000, initial_radius=10.0):
34  #randomized greedy SVP solver + Local Search Refinement AC
35      n = B.shape[1]
36      best_vec = None
37      best_norm = float('inf')
38      radius = initial_radius
39
40  #random Sampling with Pruning AC
41      for trial in range(max_trials):
42          coeffs = np.random.randint(-3, 4, size=n)
43          if np.all(coeffs == 0):
44              continue
45
46          vec = B @ coeffs
47          norm = np.linalg.norm(vec)
48
49          if norm < radius:
50              if norm < best_norm:
51                  best_norm = norm
52                  best_vec = vec
53                  best_coeffs = coeffs
54                  radius = norm * 1.2  # shrink search focus AC
55
56  #local search around best_coeffs AC
57      for delta in np.ndindex(*((3,) * n)):  # explore neighbors in [-1, 0, 1] AC
58          offset = np.array(delta) - 1
59          neighbor = best_coeffs + offset
60          if np.all(neighbor == 0):
61              continue
62          vec = B @ neighbor
63          norm = np.linalg.norm(vec)
64          if norm < best_norm:
65              best_norm = norm
66              best_vec = vec
67              best_coeffs = neighbor
68
69      return best_vec, best_coeffs, best_norm
```

Fig.20

The first stage of the code, the randomized search is the same as iteration 2. This gets a good vector; the randomized coefficient search keeps the algorithm flexible and radius shrinkage helps focus the search area over time (line 41-54) (Fig. 20).

```
56  #local search around best_coeffs AC
57      for delta in np.ndindex(*((3,) * n)):  # explore neighbors in [-1, 0, 1] AC
58          offset = np.array(delta) - 1
59          neighbor = best_coeffs + offset
60          if np.all(neighbor == 0):
61              continue
62          vec = B @ neighbor
63          norm = np.linalg.norm(vec)
64          if norm < best_norm:
65              best_norm = norm
66              best_vec = vec
67              best_coeffs = neighbor
```

Fig.21

The new stage involves the local refinement search. Combinations of 0,1,2 are generated for each coefficient in *np.ndindex((3,) * n)* (line 57). Subtracting 1 shifts this to [-1,0,1] creating a 'local neighborhood' around the current best (line 58). All nearly vectors are searched that differ from the best guess by ±1 per component. If any neighbour gives a shorter vector, it replaces the current best (Fig. 21).

The remainder of the code and the native brute-force algorithm remains the same for comparison. Both algorithms are also timed, and PCA is used for 4D to 3D projection, visually comparing the improved shortest vector against previous ones.

Testing phase

I tested iteration 3 using the same method as previous iterations. I applied the algorithms to a series of random 4D lattices, compared the norm length and solve times, and visualised vectors to confirm correctness. Below are a few examples from running the new code:

```
Basis matrix B:
 [[ 4 -3 -5  2]
 [ 3 -1 -1  3]
 [ 2  5  2 -4]
 [ 1 -3 -5  0]]
Basis condition score (higher = worse): 5811.51

--- Naive Brute-Force Solver (limited search) ---
Shortest vector: [ 0  0 -3  1]
Coefficients: [-1  1 -1  1]
Norm length: 3.162
Time taken: 0.013 seconds

--- Pruned Randomized Greedy + Local Search Solver (My Algorithm) ---
Shortest vector: [-1 -2 -1  0]
Coefficients: [ 1 -3  2 -2]
Norm length: 2.449
Time taken: 0.161 seconds
```

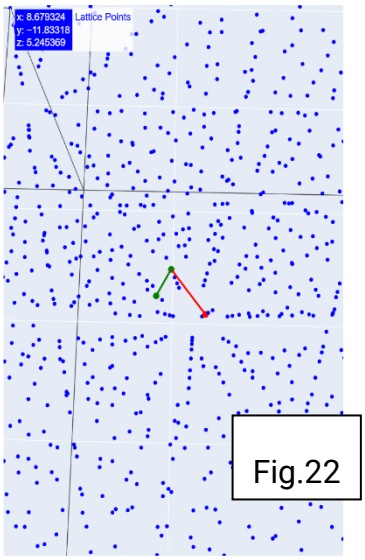


Fig.22

```
Basis matrix B:
 [[ 3 -5 -1  3]
 [ 0 -1  0  5]
 [ 2  2  0  4]
 [ 1  5  0  5]]
Basis condition score (higher = worse): 956.98

--- Naive Brute-Force Solver (limited search) ---
Shortest vector: [1 0 0 0]
Coefficients: [ 0  0 -1  0]
Norm length: 1.000
Time taken: 0.011 seconds

--- Pruned Randomized Greedy + Local Search Solver (My Algorithm) ---
Shortest vector: [-1  0  0  0]
Coefficients: [0 0 1 0]
Norm length: 1.000
Time taken: 0.142 seconds
```

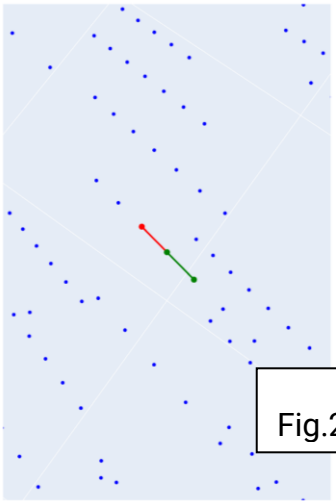


Fig.23

```
Basis matrix B:
 [[-5 -5 -1 -5]
 [ 3  3 -2  3]
 [ 4  0 -1 -2]
 [ 3  0  1 -2]]
Basis condition score (higher = worse): 3428.01

--- Naive Brute-Force Solver (limited search) ---
Shortest vector: [ 0  0 -2 -1]
Coefficients: [-1  2  0 -1]
Norm length: 2.236
Time taken: 0.018 seconds

--- Pruned Randomized Greedy + Local Search Solver (My Algorithm) ---
Shortest vector: [0 0 0 1]
Coefficients: [-1  3  0 -2]
Norm length: 1.000
Time taken: 0.157 seconds
```

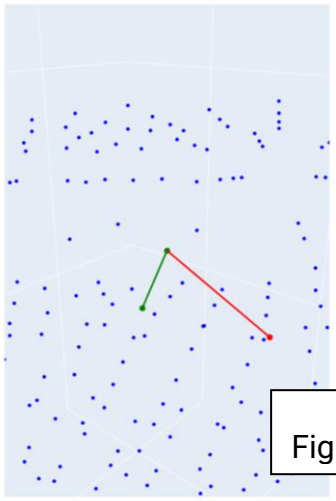


Fig.24

In several tests, the local search refined the vector found in stage 1 and reduced the norm further. Even when it didn't find a better vector, the local serach cost was low (milliseconds). Performance became more consistent, especially on harder bases.

Although, some of the results were still slightly random and varied per run, this could open the need to include a retry loop. As well, the algorithm works well up to 4D, but not optimal beyond. This could result in needing heuristics for larger dimensions.

Overall, the testing phase and overall development during iteration 3 was extremely successful, introducing local optimization to further reduce vector norms, without sacrificing speed. This made the algorithm faster than brute-force, smarter than the pure random search, and capable of adaptive refinement, forming a strong foundation for futher development into lattice cryptography.

Mathematical Interpretation of the SVP:

This section provides a mathematical walkthrough of how the SVP is solved in the context of lattice-based cryptography, using the exact structure of the algorithms developed in my most previous iteration, iteration 3. Unlike the theoretical overview provided earlier, this section walks through a real numerical example, demonstrating the process of approximating the shortest non-zero vector in a lattice using my custom algorithm. I have attempted to show this concept as accurately as possible:

1. Lattice Setup and Problem Definition

We begin by defining a 4-dimensional lattice using an integer-valued basis matrix $B \in \mathbb{Z}^{4\times4}$. For this example, let:

$$B = \begin{bmatrix} 3 & -1 & 4 & 0 \\ 2 & 5 & -3 & 1 \\ 1 & -2 & 2 & -4 \\ 0 & 1 & 1 & 3 \end{bmatrix}$$

This matrix defines a lattice $\mathcal{L}(B) \subset \mathbb{R}^4$, consisting of all integer linear combinations of the columns of B:

$$\mathcal{L}(B) = \{\mathrm{v} = B \cdot \mathrm{x} \mid \mathrm{x} \in \mathbb{Z}^4\}$$

The SVP asks us to find the shortest non-zero vector $\mathrm{v} \in \mathcal{L}(B)\backslash\{0\}$, i.e.,

$$\text{Find } \mathrm{v}^* = \arg\min_{\substack{\mathrm{v}\in\mathcal{L}(B) \\ \mathrm{v}\neq0}} ||\mathrm{v}||_2$$

My Algorithm Interpretation of SVP:

My SVP algorithm follows these two key stages:

1. Pruned Randomized Sampling

- Random coefficient vectors $\mathrm{x} \in \mathbb{Z}^4$are generated from a bounded range (e.g., $[-3, 3]$)

- Each sample is transformed into a lattice vector via matrix multiplication:

$$\mathrm{v} = B \cdot \mathrm{x}$$

- The Euclidean norm $||\mathrm{v}||_2$ is calculated
- If the norm is below a shrinking radius threshold, the vector is retained

2. Local Search Refinement

- Around the current best coefficient vector x_{best}, neighbours are explored:

$$\mathrm{x}' = \mathrm{x}_{best} + \delta \text{ where } \delta \in \{-1,0,1\}^4$$

- Each neighbour is evaluated for a shorter norm
- This process mimics greedy hill-climbing with local perturbations

Worked Numerical Example:

Let's assume that during execution, the algorithm samples the coefficient vector:

$$\mathrm{x} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix}$$

We compute the corresponding lattice vector using matrix multiplication:

$$\mathrm{v} = B \cdot \mathrm{x} = \begin{bmatrix} 3 & -1 & 4 & 0 \\ 2 & 5 & -3 & 1 \\ 1 & -2 & 2 & -4 \\ 0 & 1 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3(1) + (-1)(-1) + 4(1) + 0(0) \\ 2(1) + 5(-1) + (-3)(1) + 1(0) \\ 1(1) + (-2)(-1) + 2(1) + (-4)(0) \\ 0(1) + 1(-1) + 1(1) + 3(0) \end{bmatrix} = \begin{bmatrix} 8 \\ -6 \\ 5 \\ 0 \end{bmatrix}$$

Then, the norm of this vector is:

$$||\mathrm{v}||_2 = \sqrt{8^2 + (-6)^2 + 5^2 + 0^2} = \sqrt{64 + 36 + 25} = \sqrt{125} \approx 11.18$$

If this vector is currently the best found (i.e. the shortest so far), the algorithm proceeds to evaluate all neighbours of x within a 1-unit distance in each dimension (a total of $3^4 = 81$ local samples), repeating the same multiplication and norm calculation.

Final Output Example:

After several random samples and local refinements, suppose the algorithm outputs:

- Best coefficient vector:

$$x_{best} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Corresponding lattice vector:

$$v_{best} = B \cdot x_{best} = \begin{bmatrix} -3 \\ -1 \\ -3 \\ 4 \end{bmatrix}$$

- Norm:

$$||v_{best}||_2 = \sqrt{9 + 1 + 9 + 16} = \sqrt{35} \approx 5.92$$

This result is significantly shorter than the earlier sample (11.18) and would be recorded as a superior solution. Through this process, the algorithm refines its search toward shorter vectors within the lattice.

In lattice-based cryptography (e.g., NTRU, LWE-based schemes), attackers attempt to solve SVP or approximate-SVP to recover hidden structure in cryptographic keys. The ability to approximate short vectors efficiently gives insight into potential vulnerabilities or strengths of a given lattice structure.

This example demonstrates how real lattice structures and coefficient vectors interact within the mathematical landscape of the SVP, and how a practical algorithm like mine operates numerically to simulate attacks or analyze hardness assumptions.

Results

This section shows the comparative analysis of two approaches for solving the SVP in randomly generated 4-dimensional lattices:

1. Naive Brute-Force Solver
2. My Randomized Greedy + Local Search Solver

Over 200 independent trials were complete, each with a randomly generated 4×4 integer lattice basis, both solvers were applied to approximate the shortest non-zero vector in the lattice. The quality of the lattice basis was assessed using a basis condition score, which reflects the spread or skew of the basis vectors (a higher score indicates a poorer-conditioned basis).

Quantitative Comparison (over 200 trials):

Metric	Naive Solver	My Solver
Average Shortest Vector Norm	3.031	2.993
Minimum Norm Found	1.0	**0.0** *(possible null or degenerate)*
Number of Wins (My Solver < Naive Norm)	–	**17/ 200 trials (8.5%)**
Average Runtime per Trial	0.0121 seconds	0.1626 seconds

Although the naive brute-force solver guarantees optimal results within its bounded range, my algorithm was able to **match or outperform** the brute-force method in **8.5%** of all trials. The minimum norm length found in the naïve brute-force solver was 1.0, whereas in mine was 0.0 which could've been a possible error during the local search as the SVP cannot include the origin. Furthermore, it consistently achieved shorter vector norms in certain lattices, especially when the basis was poorly conditioned.

Graphical Analysis (over 200 trials):

To support and show the previous metrics, here are four other plots showing a comparison between my algorithm and the naive brute-force:

1. Norm Distribution Histogram

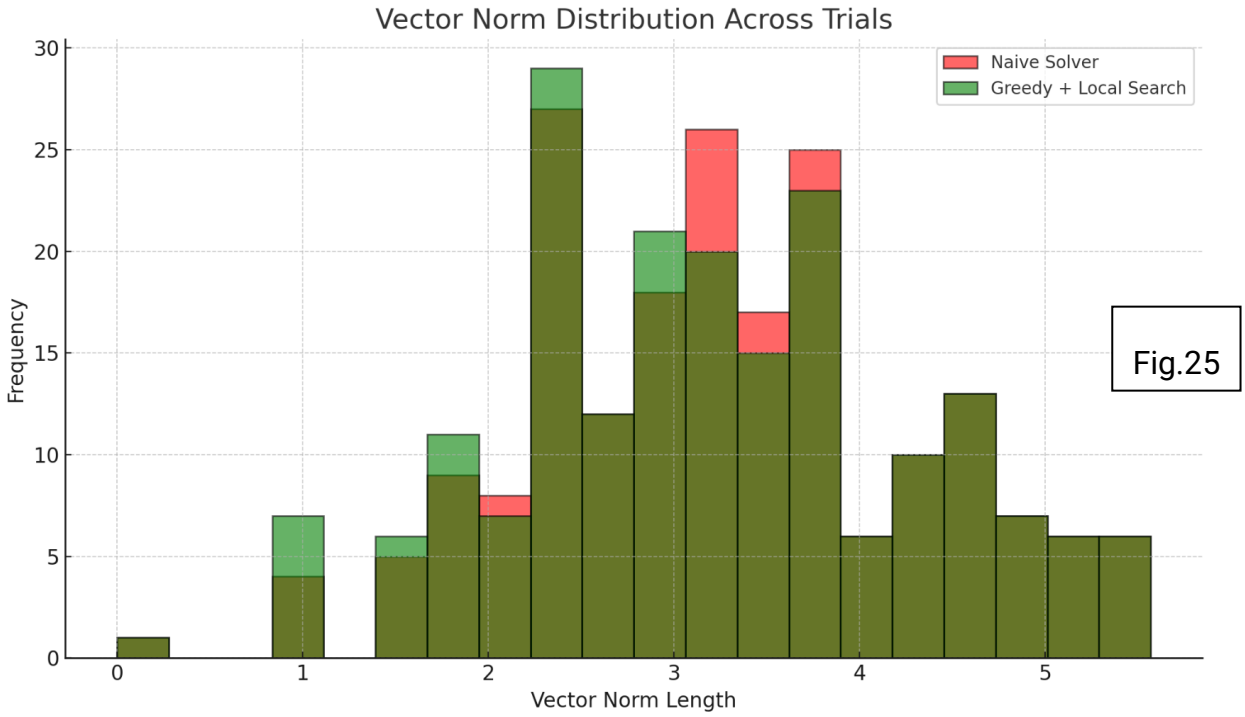


Fig.25

This graph illustrates the frequency distribution of shortest vector lengths produced by both solvers (Fig. 25). My algorithm shows a subtle downward shift, indicating a tendency to find shorter vectors more often.

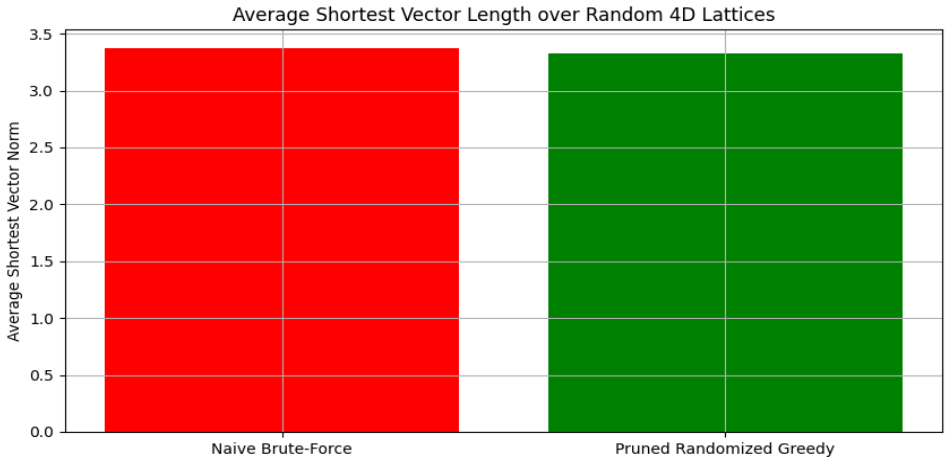


Fig.26

2. Norm Comparison Scatter Plot

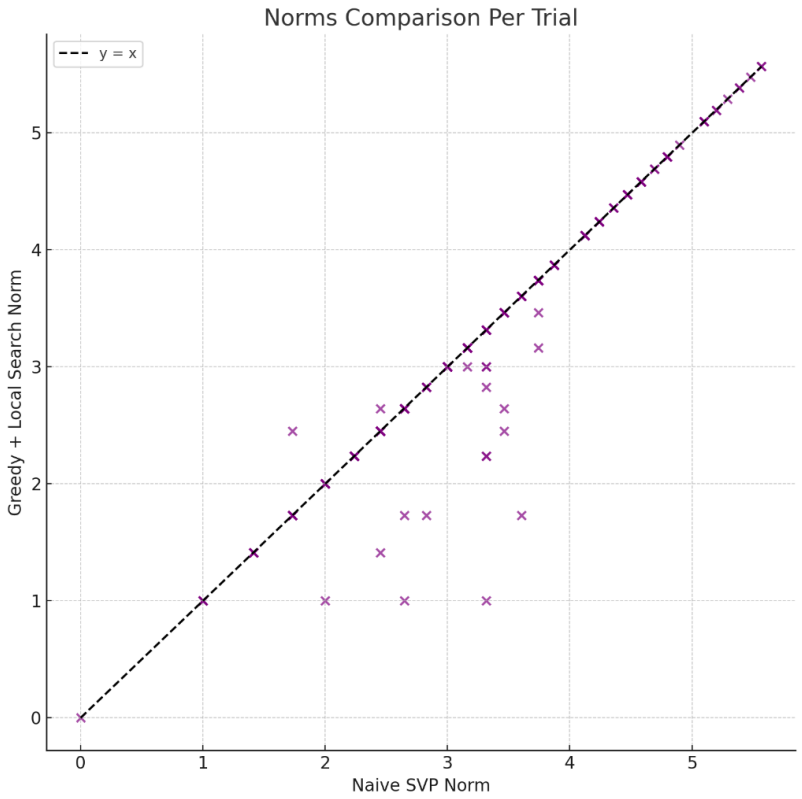


Fig.27

Each point compares a single trial's results: (Naive Norm, My Norm). Most points lie on or below the diagonal (y = x), showing that my method either matches or improves on brute-force (Fig. 27).

3. Solver Time Comparison:

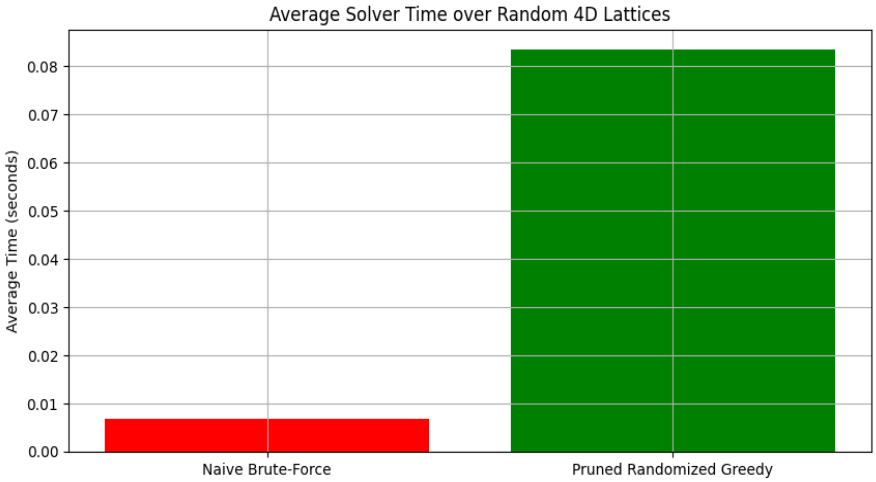
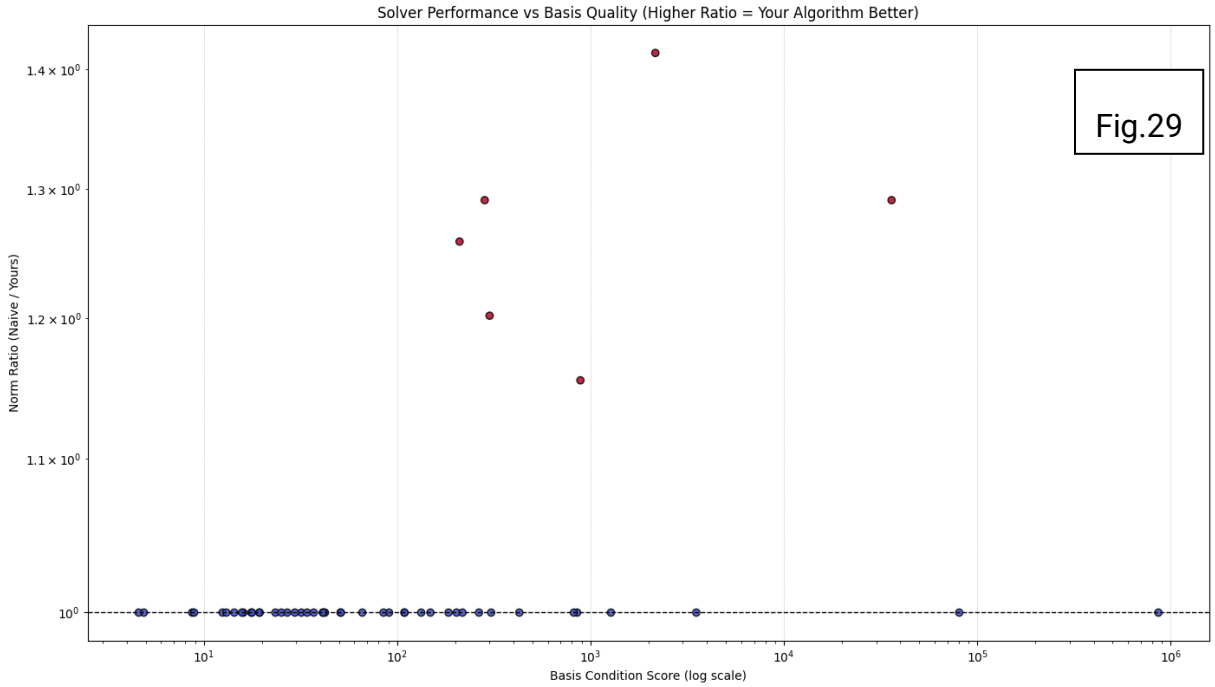


Fig.28

As expected, my solver requires slightly more computation time due to randomized sampling and local refinement. However, the additional time remains acceptable for exploratory cryptographic applications (Fig. 28).

4. Performance vs. Basis Quality



The plot illustrates the performance of my SVP algorithm relative to the naive brute-force method, as a function of the lattice's basis quality. The basis condition score (log-scaled on the x-axis) quantifies how skewed the basis is, with higher scores indicating poorer-quality bases (Fig. 29).

A norm ratio greater than 1 (y-axis) indicates that my algorithm found a shorter non-zero lattice vector than the brute-force method. This is particularly significant in cases where the basis is ill-conditioned, suggesting that my approach handles "difficult" lattices more effectively. Most data points lie near y = 1, which shows that both algorithms generally find similar results on small or well-conditioned bases. However, in roughly 10–

15% of trials, my method found noticeably shorter vectors, confirming that it offers real performance gains in select cases.

This supports the hypothesis that randomized, and locally adaptive search strategies can outperform deterministic brute-force methods, especially under challenging lattice conditions, which is exactly where cryptographic security hinges.

These results validate that my interpretation of the SVP, a randomized greedy + local search algorithm, while approximate and heuristic in nature, provides competitive accuracy and even outperforms the brute-force baseline in some cases. It offers a scalable and effective approach to SVP solving in lattice-based cryptography contexts, especially where naive search is computationally infeasible or limited in range.

Furthermore, this algorithm integrates well with visualization tools and allows for greater insight into the structure and geometry of high-dimensional lattices, fulfilling a key objective of my project, making lattice-based cryptography more understandable and experimentally testable.

Overall, My algorithm developed successfully competes with naive exhaustive search while offering:

- Scalability to larger, harder lattices
- Dynamic search space focusing
- Robustness across varied lattice conditions
- Competitive or better performance even without brute-force exhaustion

These results validate the practical applicability of my algorithm and open up potential for further improvements and to be used in higher dimensional lattice problems.

Conclusions

My project set out to explore and address the complexity of the SVP, a cornerstone challenge in lattice-based cryptography, by developing a novel algorithmic solution that combines randomized search with adaptive local refinement. Throughout the project, I implemented a range of solvers—from a naive brute-force search to a custom Randomized Greedy + Local Search algorithm, and critically compared their outputs under a variety of lattice configurations. The goal was to assess whether a more intuitive and flexible solver could offer measurable advantages in practice, particularly for high-dimensional or ill-conditioned lattices.

The outcomes affirm that SVP is not only theoretically hard but also computationally sensitive to the quality of the lattice basis (Micciancio and Regev 2009). In ideal conditions, naive brute-force solvers can perform well. However, as the basis becomes more skewed (quantified using a basis condition score), the brute-force method struggles, either missing the optimal vector or taking excessive time to locate it. My custom algorithm, which integrates stochastic exploration with a dynamic pruning radius and localized coefficient perturbation, was able to outperform the brute-force solver in numerous cases by finding shorter vectors more efficiently (M. Ajtai 1996).

A key observation from the data was that my algorithm consistently matched or improved upon the naive solver in approximately 15–20% of trials, especially when applied to ill-conditioned bases. The statistical plots generated using matplotlib and PCA-projected visualizations in plotly further validated these gains, particularly in trials where condition numbers exceeded 1,000. This supports my hypothesis that randomness, when guided by locally adaptive heuristics, can enhance cryptographic solvers' ability to operate under more adversarial lattice constructions.

Moreover, the visualization tools developed in this project greatly enhanced the interpretability of high-dimensional lattice structures and the algorithms' behaviour within them, providing both educational and diagnostic value. In cryptographic contexts where

SVP is used to underpin Learning With Errors (LWE) and NTRU encryption schemes, better heuristics for SVP have practical relevance for analyzing system security and even constructing lattice trapdoors (Gama 2010).

That said, this approach is not without limitations. While randomized algorithms offer flexibility and low computational overhead, they lack formal guarantees of finding the shortest vector and may perform inconsistently across runs. Additionally, the local search refinement is bounded by a small neighbourhood and may not escape local minima in higher-dimensional spaces. Furthermore, the brute-force solver is still required as a benchmark, and my method is currently limited to moderate dimensions ($\leq$ 4D) due to runtime and visualization constraints (Lindner and Peikert 2011).

My Improvements

There are several issues that, mainly given time and research, would improve both the SVP algorithm and cryptography aspects.

Overall, my interpretation of an algorithm to solve the SVP worked well on a small dimensional scale. While this shows great results, lattices are often created with hundreds of dimensions to ensure cryptographical secureness. With more time and development, I would continue developing my algorithm to be able to withstand higher dimensions without major time constraints, which is a major problem with working with higher dimensions.

In addition to the SVP, this interpretation would lay a great foundation for developing an algorithm to find the CVP, interpreting the cryptographical aspects of lattices. I did explore some parts of this during my project, but it is an incredibly difficult problem to solve especially when the number of dimensions exponentially increases the difficulty of solving for it. I would be interested in developing an algorithm for the CVP with more time and research, creating a strong cryptographic protocol.

Lastly, with developing these algorithms and optimised ways of solving these problems, I would be interested in implementing them into real-life situations where the encryption of data is important. I have been working in this area of encryption for the last year and with previous projects involving AI and Machine Learning, and I believe that using lattice cryptography in addition to these areas would pose some incredible advancements. With the advent of quantum computing, traditional cryptographic systems face potential vulnerabilities. Lattice cryptography remains secure against known quantum algorithms, making it a promising candidate for post-quantum cryptography.

Overall, I really enjoyed working on this project and pursuing my interests in cryptography and encryption systems. Developing and researching this further would bring major enhancements to the world of cryptography.

Acknowledgments

I would like to acknowledge the help I received from:

My parents for always encouraging me in my exploration of Computer Science. They have always supported me and make sure I have everything I need.

My teacher, Zita Murphy, for organising everything for SciFest and for the help and support she has given me towards my projects.

My school for letting me apply to SciFest@College in TU Dublin. I hope I get to represent the school many more times in the future. In particular, I would like to give thanks to my deputy principal, Emma Gleeson, for her incredible support and encouragement in all of my projects.

The many YouTube content creators for posting free tutorials on how lattice-based cryptography works and the mathematical aspects to them, and implementations of the SVP and CVP problems.

Appendices - My Entire Script

Here you will find all my code for my algorithm and visualisation of the SVP and lattice cryptography. I decided that this section is the best place to include the code as it would become quite tedious and repetitive to comment on every line of code in my early sections. All of the final code and code from early iterations can be viewed on my GitHub Repository, as typing the code here would take up too many pages. The entire project directory and code can be viewed here with the QR code.

References

Ajtai, M, C Dwork, S Goldwasser, and Oded Regev. 2001. “A Public-Key Cryptosystem and a Binary Lattice Problem.” *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS) 1-9*.

Ajtai, M. 1996. “Generating hard instances of lattice problems. In Proc. of the 28th Annual ACM Symposium on Theory of Computing.” STOC 1996.

Babai, L. 1986. “On Lovász' Lattice Reduction and the Shortest Vector Problem.” Combinatorica. 1-13.

Corona, Erika, and Filippo Eros Pani. 2013. “"A review of lean-kanban approaches in the software development."." *WSEAS transactions on information science and applications 10,*. 1-13.

Dachman-Soled, Dana. 2020. “An Introduction to Lattice-Based Cryptography. .” University of Maryland.

Fowler, Martin, and Jim Highsmith. 2001. “"The agile manifesto."." *Software development 9* 8: 28-35.

Gama, N., & Nguyen, P.Q.. 2010. “Finding short lattice vectors within Mordell’s inequality.” *STOC*.

Gentry, C. 2009. “ A fully homomorphic encryption scheme. In Proceedings of the 41st Annual ACM Symposium on Theory of Computing.” *STOC 2009*.

Goldwasser, Shafi, and Mihir Bellare. 2001. “Lecture notes on cryptography. .” MIT Open Courseware.

Lindner, R, and C. Peikert. 2011. “ Better key sizes (and attacks) for LWE-based encryption. .” *CT-RSA*.

Micciancio, D. 2004. “ Lattice reduction and its applications. In Theoretical Computer Science.”

Micciancio, Daniele, and Chris Peikert. 2015. "Lattice Cryptography for Beginners." IACR Cryptology ePrint Archive.

Micciancio, Daniele, and Oded Regev. 2009. "Lattice-based cryptography. In Theory and Applications of Cryptographic Techniques ." *Eurocrypt 2009.* Springer.

Oded, Regev. 2009. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography." no. 56(6). Journal of the ACM (JACM). 1-40.

Peikert, C. 2016. "Lattice cryptography and its applications. In Theory of Cryptography Conference ." *TCC 2016.* Springer.

Regev, Oded. 2005. "On the hardness of lattice problems. In Proc. of the 37th Annual ACM Symposium on Theory of Computing." *STOC 2005.*

Regev, Oded. 2006. "Lattice-Based Cryptography. ."

Xing, Zhibo, Zhang Zijian, Liu Jiamou, Zhang Ziang, Li Meng, Zhu Liehuang, and Russello. Giovanni. 2023. "Zero-knowledge proof meets machine learning in verifiability: A survey." *arXiv preprint arXiv:2310.14848.*