ESKOM EXPO FOR YOUNG SCIENTISTS INTERNATIONAL SCIENCE FAIR (ISF) 2025

Advancing Post-Quantum Lattice-based Cryptography by developing Efficient Shortest Vector Problem Approaches.

Addison Carey

Category: Computer Sciences and Software Development (COM)

Sub-category: Software Systems

Province and Region: Kildare, Ireland

School: Celbridge Community School

Grade: 10

Project Report

Table of Contents

1 Abstract (250 words)

Purpose

The purpose of this project was to create and evaluate a heuristic algorithm for the Shortest Vector Problem (SVP), a fundamental challenge in lattice-based cryptography. The engineering goal was to integrate an algorithm with a refined selection method that could find short vectors in lattices more effectively than basic methods. It was expected that this algorithm would give results similar to traditional brute-force search in small lattices, but with potential to scale better as dimensions increase.

Method

The algorithm was written in Python and tested on randomly generated lattices between three and six dimensions. Before execution, the lattice bases were conditioned to improve their structure. The refined algorithm was then compared with a brute-force solver by examining the length of the vector it produces, its runtime, and how well it handled complex lattice conditions. Randomness was assessed for using multiple trials.

Results

The refined algorithm reliably found short vectors of similar quality to brute-force, and in some challenging cases, it out-performed brute-force algorithms. Although this method experienced higher runtime in lower dimensions, it displayed statistically significant improvements in reliability when the lattices were harder to solve. Results varied between executions due to its randomised design, but overall patterns remained consistent.

Conclusion

The project successfully met its goal of developing a new heuristic solver for SVP. The results suggest this could become a useful approach for higher-dimensional lattices, where traditional methods are infeasible. This work provides a foundation for future research in post-quantum cryptography, where efficient solvers are critical.

2 Introduction

Over the past number of decades, cryptography has played a central role in securing communication, financial transactions, and data storage across various digital systems. Classical public-key cryptographical protocols, for example RSA and elliptic curve cryptography (ECC), rely on number-theoretic problems such as integer factorisation and the discrete logarithm problem (Bernstein, et al., 2009). These problems are considered computationally difficult for conventional computers, thereby ensuring the security of cryptographical protocols (Micciancio & Goldwasser, 2002). However, the anticipated emergence of quantum computing threatens these systems. Shor's algorithm, when executed on a sufficiently powerful quantum computer, has the capability to efficiently solve the integer factorisation and discrete logarithm problems, rendering widely used classical cryptography insecure (Shor, 1997). This builds an urgent need for post-quantum cryptographic schemes that can resist attacks from both classical and quantum adversaries (NIST, 2016).

One of the most promising directions in post-quantum cryptography is the use of lattice-based cryptographic schemes. Lattices are regular arrangements of points in Euclidean space defined by integer linear combinations of basis vectors (Micciancio & Goldwasser, 2002). Unlike number-theoretic assumptions, many hard problems in lattice theory remain resistant to both classical and quantum algorithms (Regev, 2005). Out of these, the Shortest Vector Problem (SVP) stands out as one of the most fundamental. The SVP asks that given a lattice basis, what is the shortest non-zero vector in that lattice? The difficulty of solving SVP derives the security of several cryptographic primitives, including Learning With Errors (LWE) and NTRU, both of which are leading candidates in the NIST Post-Quantum Cryptography standardisation process (Ajtai, 1996) (Regev, 2005) (NIST, 2016).

Regardless of its importance, solving the SVP is computationally challenging. It is NP-hard under standard reductions (Micciancio & Goldwasser, 2002), and the problem's difficulty escalates exponentially with the dimension of the lattice. In real-world cryptographic applications, lattices of hundreds or even thousands of dimensions are obligated to provide adequate security (Bernstein, et al., 2009). At these scales, exact algorithms for SVP are infeasible. Correspondingly, much attention has turned to the development of approximation algorithms and heuristics which can produce useful, is not always optimal, solutions. These methods typically seek to balance accuracy and computational efficiency.

A notably powerful pre-processing technique is basis reduction, where a given lattice basis is transformed into a "reduced" basis with a shorter and more orthogonal vectors. The Lenstra-Lenstra-Lovász (LLL) algorithm is the most recognised polynomial-time reduction method, widely used in both cryptography and in experimental studies of SVP (Lenstra, et al., 1982). By adopting LLL, the condition number of the lattice basis is improved, which can make subsequent heuristic approaches to SVP more effective. Nonetheless, even with reduction, the problem remains computationally demanding at high dimensions.

The motivation for this project lies in exploring heuristic strategies for solving SVP in low to moderate dimensional lattices, with a view to comprehending their strengths, limitations, and potential implications for cryptographic applications. While the cryptographic relevance of SVP lies in high dimensions, low-dimensional experimentation provides an accessible framework for developing and testing algorithmic behaviour, and a deeper insight of how different heuristics cooperate with lattice reduction (Micciancio & Voulgaris, 2010).

This project focuses on the design, implementation, and testing of a Pruned Randomised Greedy (PRG) heuristic algorithm for SVP. This algorithm incorporates randomised sampling, pruning strategies, and local search refinement, and is tested both on original and LLL-reduced bases. This project does not claim to solve SVP at cryptographically significant dimensions: rather, it seeks to investigate the potential of randomised heuristic approaches in small dimensions as a foundation for future scaling and adaption.

The importance of this research additionally lies in its position relative to existing research. Many classical algorithms for SVP, for example sieving, enumeration, and lattice reduction, trade efficiently for accuracy (Schnorr & Euchner, 1994) (Gama & Nguyen, 2008). Enumeration assures exact solutions but scales poorly. Sieving methods attain better asymptotic complexity but remain costly in practice. Heuristic methods such as randomised search and pruning are often employed as compromises, especially when studying dimensions too small for cryptographic deployment but large enough to provide insights into algorithmic structure. Establishing the PRG heuristic within this spectrum permits for a clearer understanding of its advantages and limitations.

Beyond its technical contribution, this project reflects the continuing challenge of bridging theory and practice in post-quantum cryptography. While most cryptographic protocols require lattices of very high dimension, progress often starts with careful study of low-dimensional cases, where algorithmic performance can be observed, compared, and refined (Hanrot, et al., 2011). These smaller experiments do not replicate real cryptographic settings, but they provide understanding of how reduction techniques and heuristics interact, and aid to evaluate whether a given approach shows potential for scaling.

This project also targets to contribute to the broader research dialogue by highlighting the limitations as well as the benefits of heuristic methods. In particular, heuristic strategies can sometimes identify shorter vectors more efficiently than constrained brute-force approaches, but they offer no guarantees of optimality (Kannan, 1987). Acknowledging this trade-off is essential: rather than presenting heuristics as final solutions, the work frames them as exploratory tools that may guide the design of more sophisticated methods in the future.

The expansive significance of this work lies in its contribution to post-quantum cryptographic research. By examining and experimenting with heuristic methods for SVP, the project objects to highlight the promise and challenges of lattice-based cryptography, an area that is expected to play a defining role in the future of secure communication in a post-quantum world (Campbell, et al., 2020).

The proposed hypothesis is that by applying refined algorithms, local search refinement, and LLL-like basis reduction, it is possible to efficiently approximate solutions to the SVP in lattices of varying dimensions, achieving shorter vectors and lower solver runtimes compared to standard brute-force methods.

3 Literature Review

The SVP is one of the fundamental problems in lattice theory, both from a theoretical viewpoint and as a foundation of modern cryptography. This literature review is divided into four thematic sections: theoretical foundations, classical algorithmic approaches, heuristic and randomised approaches, and the identified research gap.

3.1 Theoretical Foundations of the SVP

- Definition: The SVP is the problem of finding the shortest non-zero vector in a lattice with respect to the Euclidean norm. It is formally known to be NP-hard under randomised reductions (Ajtai, 1996) (Arora, et al., 1997).
- Importance: Its computational hardness grounds the security of lattice-based cryptography, along with encryption, digital signatures, and homomorphic encryption schemes (Micciancio & Regev, 2009) (Regev, 2005).
- Cryptographic relevance: Lattices of high dimensions (hundreds or thousands) structure the basis for post-quantum cryptography, which objects to resist attacks by quantum algorithms, such as Shor's algorithm (Shor, 1994) (Ajtai & Dwork, 1997).

3.2 Classical Exact Approaches

- Enumeration methods: These systematically test lattice vectors with a given radius. This guarantees correctness but expands exponentially, making them unattainable beyond small dimensions (Micciancio & Regev, 2009).
- Voronoi cell algorithms: They partition the lattice into cells to identify the shortest vector exactly (Nguyen & Stern, 2001). These are exceedingly precise, but computationally expensive.
- Complexity: While exact solvers remain critical for theory and benchmarking, they do not extend to cryptographically relevant dimensions (Gama & Nguyen, 2008).

3.3 Basis Reduction Approaches

- LLL Algorithm: The Lenstra-Lenstra-Lovász reduction (1982) was a breakthrough in making lattice problems approachable (Lenstra, et al., 1982). It presents reduced bases with shorter, more orthogonal vectors. While it does not solve SVP directly, it simplifies following heuristics.
- BKZ (Block Korkine-Zolotarev): This extends LLL with blockwise reduction, yielding bases closer to optimal. This is widely used in cryptographic analysis (Chen & Nguyen, 2011).
- Impact: Basis reduction is now a typical preprocessing step before applying other algorithms.

3.4 Heuristic and Randomised Approaches

- Sieving methods: Algorithms like GaussSieve and ListSieve approximate SVP by combining vectors iteratively. They are amidst the strongest heuristics known but demand large memory resources (Micciancio & Regev, 2009).
- Pruning techniques: These diminish enumeration search to promising areas, balancing efficiency with accuracy (Gama & Nguyen, 2008).
- Randomised sampling: Randomised approaches, frequently combined with local refinement, often find short vectors quickly, nevertheless without guarantees (Nguyen & Stern, 2001).
- Observations: These heuristics emphasise that in practice, exact optimality is often sacrificed for scalability and speed.

3.4 Identified Research Gap

From the literature, it is distinct that:

1. Exact algorithms are computationally infeasible beyond low dimensions.
2. Robust heuristics require substantial resources, careful implementation, or parameter tuning.
3. Educational or lightweight heuristics that allow experimentation and theoretical exploration in small dimensions are relatively underexplored.

This project aims to contribute to this gap by proposing a randomised, pruned greedy heuristic with local refinement that:

- Is implementable with modest resources,
- Can be benchmarked against brute-force search in small dimensions,
- Serves as a theoretical stepping stone toward more progressive heuristics.

3.5 Problem Statement

Regardless of decades of research, the SVP remains a central challenge in computational mathematics and cryptography. Exact algorithms assure correctness but scale exponentially and are infeasible past low dimensions. More advanced techniques have pushed the boundaries of what is computationally obtainable but require steady memory, parameter tuning, or advanced implementations that limit accessibility.

This leaves an open space for novel heuristic methods that incorporate efficiency and practicality, providing approximate solutions that can expand better than brute-force while maintaining competitive accuracy.

Problem statement:

Can a randomised, pruned greedy heuristic with local refinement provide an effective and competitive approach to approximating solutions to the SVP, in particular when combined with LLL basis reduction, and how does it function relative to brute-force methods across a range of lattice dimensions?

3.6 Aim

The aim of this project is to propose, enforce and evaluate a heuristic algorithm that integrates randomisation, pruning, and local refinement in order to approximate solutions to the SVP. This project further aims to compare its performance against brute-force solvers to access its scalability as lattice dimensions grow.

3.7 Engineering and Design Goals

1. Algorithm Development: Formulate and implement a randomised, pruned greedy heuristic augmented with local refinement for solving the SVP.
2. Preprocessing Integration: Employ LLL basis reduction to improve lattice conditioning and strengthen solver performance.
3. Comparative Evaluation: Test the proposed heuristic against brute-force search methods across differing lattice dimensions (up to the limits of computational feasibility).
4. Performance Analysis: Judge the effectiveness of the heuristic in terms of solution quality, runtime, and expandability, with particular attention to ill-conditioned lattices.
5. Cryptographic Relevance: Relate findings from small to medium dimensional lattices to their implications for post-quantum cryptography, acknowledging ongoing restraints while accentuating potential directions for higher-dimensional applications.

4 Method

The methodology for this project was designed to systematically develop, test, and refine an algorithm for approximating solutions to the SVP. The process mixed theoretical analysis with computational experimentation, progressing through a series of iterative models. Each iteration incorporated improvements based on observed performance, culminating in a final heuristic that integrates pruning, randomisation, local refinement and LLL-based preprocessing.

The work followed the Agile methodology cycle (Figure 1), typical of computational engineering projects. 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 and individuals 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-organising and cross-functional and rely on regular communication and collaboration to achieve their goals.

1. Initial prototypes were created to establish a functional baseline.
2. Successive refinements were introduced, tested, and evaluated.
3. A final version was compared against a benchmark brute-force solver to measure progress toward the engineering and design goals.

All code was written in Python and implemented in a high-level programming environment, while mathematical reasoning and worked examples were used to validate algorithmic choices.

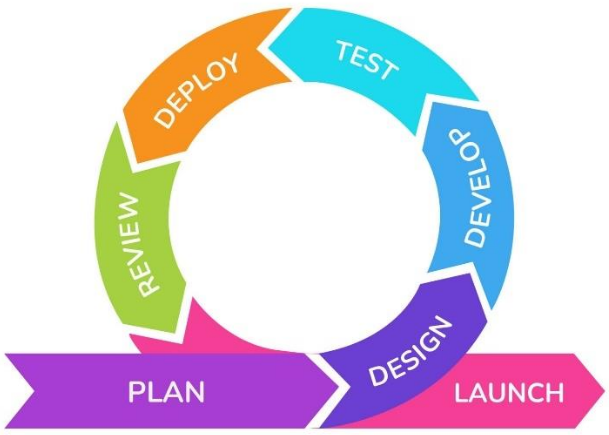


Figure 1: Agile Methodology Cycle

4.1 Procedure

The procedure involved four main phases: Planning and Designing, Developing, Testing, and Evaluation. Each iteration progressively advanced the capability of the algorithm.

4.1.1 Iteration 1

Plan and Design Rationale

The purpose of Iteration 1 was to establish a baseline solver for the SVP (Figure 2). This provided a controlled environment against which subsequent heuristic approaches could be evaluated. The brute-force method was selected because it guarantees correctness in low-dimensional lattices, even though it becomes computationally infeasible as the dimension increases.

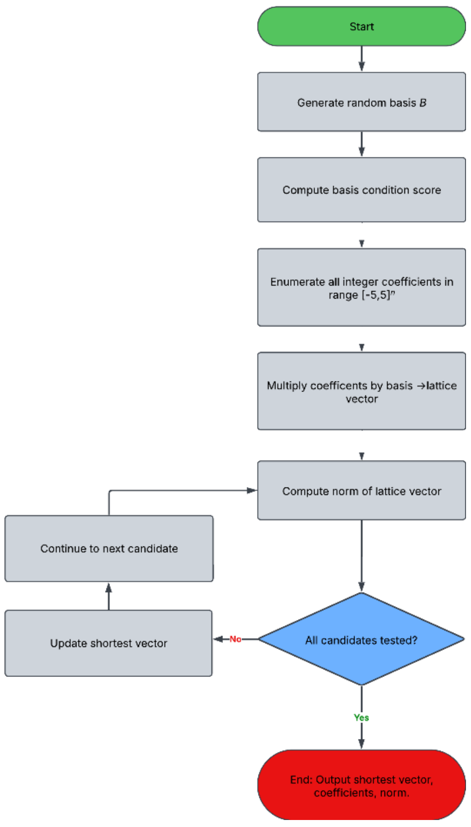
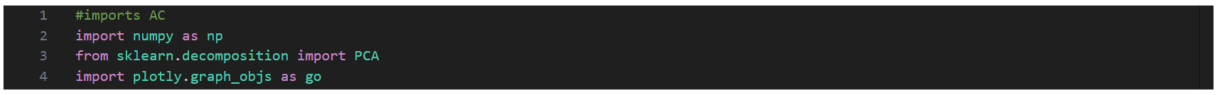

Start
Generate random basis B
Compute basis condition score
Enumerate all integer coefficients in range [-5,5]ⁿ
Multiply coefficents by basis →lattice vector
Compute norm of lattice vector
Continue to next candidate
Update shortest vector
No
All candidates tested?
Yes
End: Output shortest vector, coefficients, norm.

Figure 2: Flowchart for Iteration 1 code.

Development

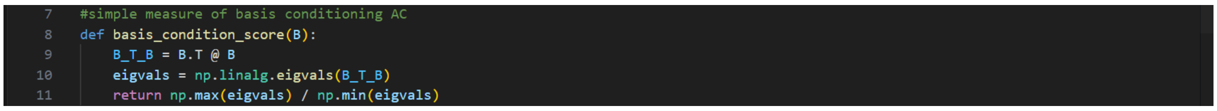
The Iteration 1 program was written as a structured Python script. Below is a breakdown of its main components:

1. Imports and Setup


```
#imports AC
import numpy as np
from sklearn.decomposition import PCA
import plotly.graph_objs as go
```

In these lines of code, *numpy* was used for matrix operations. *Principal Component Analysis* (*PCA)* from *scikit-learn* reduced high-dimensional data to three dimensions for visualisation purposes. Additionally *plotly* was used to handle 3D plotting to display the lattice.

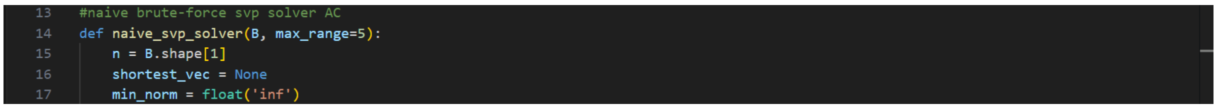
2. Basis Conditioning Function



```python
#simple measure of basis conditioning AC
def basis_condition_score(B):
    B_T_B = B.T @ B
    eigvals = np.linalg.eigvals(B_T_B)
    return np.max(eigvals) / np.min(eigvals)
```

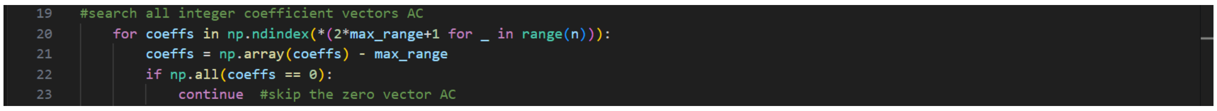
This function quantified the conditioning of a lattice basis by computing the eigenvalues of $B^T B$ and returning their ratio. A higher score implied a poorly conditioned basis (vectors nearly linearly dependent), while a lower score implied a better-conditioned, more orthogonal basis. This score became a significant diagnostic throughout the project.

3. Naive Brute-Force SVP Solver



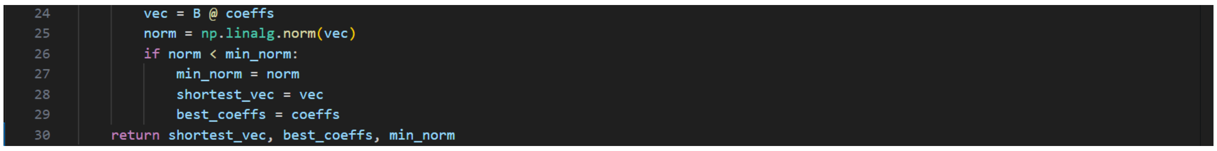
```python
#naive brute-force svp solver AC
def naive_svp_solver(B, max_range=5):
    n = B.shape[1]
    shortest_vec = None
    min_norm = float('inf')
```

The solver began by extracting the dimension of the basis n. It initialised *shortest_vec* to *None* and set *min_norm* to infinity, preparing for comparison as candidate vectors were generated.



```python
#search all integer coefficient vectors AC
    for coeffs in np.ndindex(*(2*max_range+1 for _ in range(n))):
        coeffs = np.array(coeffs) - max_range
        if np.all(coeffs == 0):
            continue  #skip the zero vector AC
```

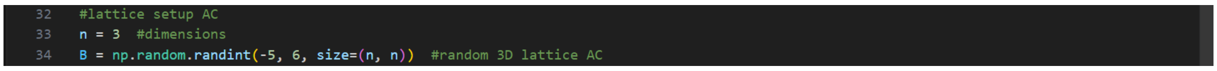
This loop systematically enumerated all integer coefficient vectors within the hypercube $[-5,5]^n$. The zero vector was excluded, since the SVP requires a non-zero shortest vector.



```python
        vec = B @ coeffs
        norm = np.linalg.norm(vec)
        if norm < min_norm:
            min_norm = norm
            shortest_vec = vec
            best_coeffs = coeffs
    return shortest_vec, best_coeffs, min_norm
```

Each coefficient vector was multiplied with the basis B to generate a lattice vector. Its Euclidean norm was computed, and if shorter than the current best, it was stored as the new candidate. At the end, the function returned the shortest vector, its coefficients, and its norm.

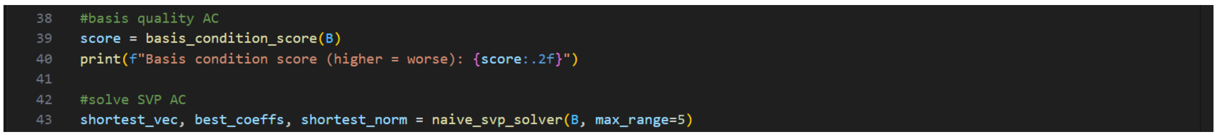
4. Random Basis Generation



```python
#lattice setup AC
n = 3  #dimensions
B = np.random.randint(-5, 6, size=(n, n))  #random 3D lattice AC
```

A random 3-dimensional lattice basis was generated with integer entries between -5 and 5. Small dimensions were chosen deliberately, as brute-force enumeration grows exponentially with dimension.

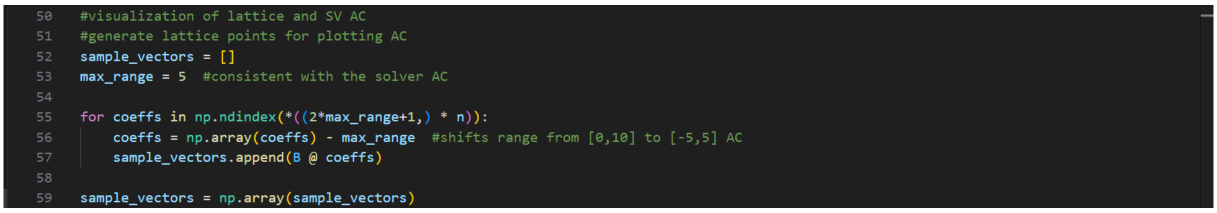
5. Solver Execution and Output



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

#solve SVP AC
shortest_vec, best_coeffs, shortest_norm = naive_svp_solver(B, max_range=5)
```

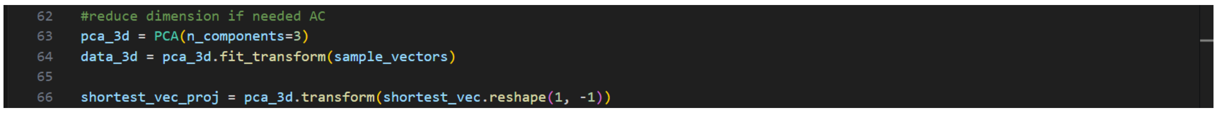
The conditioning score was calculated first, followed by the execution of the brute-force solver. Outputs included the shortest vector, its coefficient representation, and its Euclidean length.

6. Visualisation



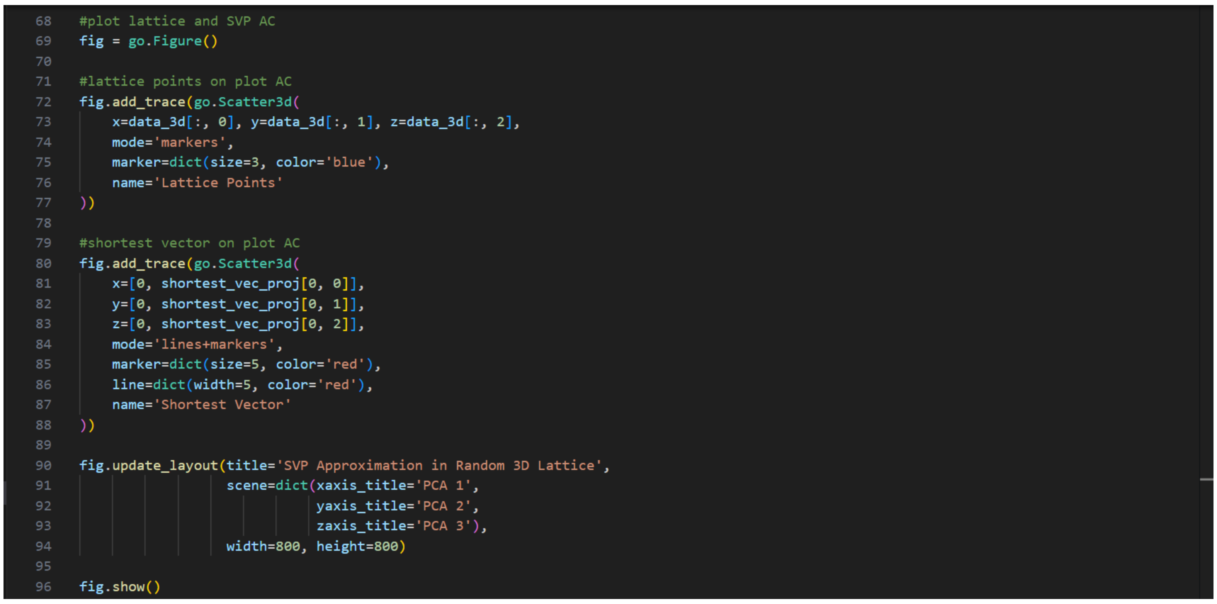
```python
#visualization of lattice and SV AC
#generate lattice points for plotting AC
sample_vectors = []
max_range = 5  #consistent with the solver AC

for coeffs in np.ndindex(*((2*max_range+1,) * n)):
    coeffs = np.array(coeffs) - max_range  #shifts range from [0,10] to [-5,5] AC
    sample_vectors.append(B @ coeffs)

sample_vectors = np.array(sample_vectors)
```

A set of lattice points was generated for visualisation by applying the basis to all coefficient combinations within the same range.



```python
#reduce dimension if needed AC
pca_3d = PCA(n_components=3)
data_3d = pca_3d.fit_transform(sample_vectors)

shortest_vec_proj = pca_3d.transform(shortest_vec.reshape(1, -1))
```

PCA projected the lattice points into 3-dimensional space, making them plottable regardless of original dimension. The shortest vector was also projected for display.



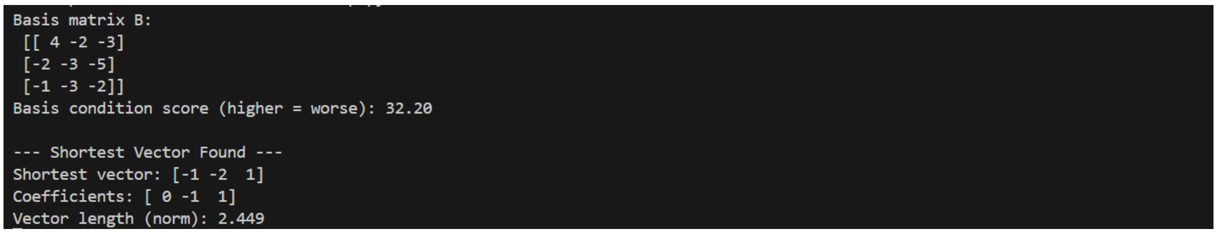
```python
#plot lattice and SVP AC
fig = go.Figure()

#lattice points on plot AC
fig.add_trace(go.Scatter3d(
    x=data_3d[:, 0], y=data_3d[:, 1], z=data_3d[:, 2],
    mode='markers',
    marker=dict(size=3, color='blue'),
    name='Lattice Points'
))

#shortest vector on plot AC
fig.add_trace(go.Scatter3d(
    x=[0, shortest_vec_proj[0, 0]],
    y=[0, shortest_vec_proj[0, 1]],
    z=[0, shortest_vec_proj[0, 2]],
    mode='lines+markers',
    marker=dict(size=5, color='red'),
    line=dict(width=5, color='red'),
    name='Shortest Vector'
))

fig.update_layout(title='SVP Approximation in Random 3D Lattice',
                  scene=dict(xaxis_title='PCA 1',
                             yaxis_title='PCA 2',
                             zaxis_title='PCA 3'),
                  width=800, height=800)

fig.show()
```

The lattice points were plotted in blue, while the shortest vector was displayed in red. This step confirmed visually that the algorithm was correctly identifying short lattice vectors.

Testing

The program successfully identified the shortest vector across multiple randomly generated lattices in three dimensions (Figure 3).

- The Basis Condition Score provided a quantitative measure of basis difficulty.
- The Shortest Vector Output returned both the coefficients and the vector norm, validating correctness.
- The Visual Confirmation implementing the PCA-based 3D plots clearly highlighted the shortest vector relative to the lattice structure.



```
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
```

Figure 3: Example Output of Iteration 1 Code

Evaluation

While accurate, the runtime scaled poorly with higher values of n or *max_range*, confirming the brute-force approach as a valid baseline but unsuitable for cryptographic-scale lattices. Some of the main limitations identified in this iteration include:

1. Scalability: The exponential runtime growth meant the solver became impractical beyond dimensions 4-5.
2. Cryptographic Gap: The lattices in practice require dimensions in the hundreds, highlighting the necessity of heuristics.
3. Dependency on Range: The maximum search range parameter (*max_range*) influenced whether the true shortest vector was found. Increasing it improved accuracy but at the cost of exponential runtime.

In conclusion, these limitations justified developing a heuristic algorithm in Iteration 2.

4.1.2 Iteration 2

Plan and Design Rationale

Iteration 1 created a baseline brute-force solver for the SVP. While it demonstrated effective in three dimensions, brute-force scales exponentially with lattice rank, quickly becoming impractical.

Iteration 2 aimed to address this limitation by introducing a heuristic solver: the PRG algorithm. Instead of exhaustively searching every possible coefficient vector, PRG randomly samples candidate coefficient combinations within bounded ranges, prunes those that exceed a working radius, and updates this radius dynamically as better solutions are found (Figure 4).

The design goals included:

1. To move from exact but inefficient methods, toward approximate but scalable methods.
2. To implement a solver that could demonstrate reduced runtime compared to brute force in higher dimensions (4D+)
3. To integrate both solvers (Brute-force and PRG) for direct comparison in terms of speed and solution quality.

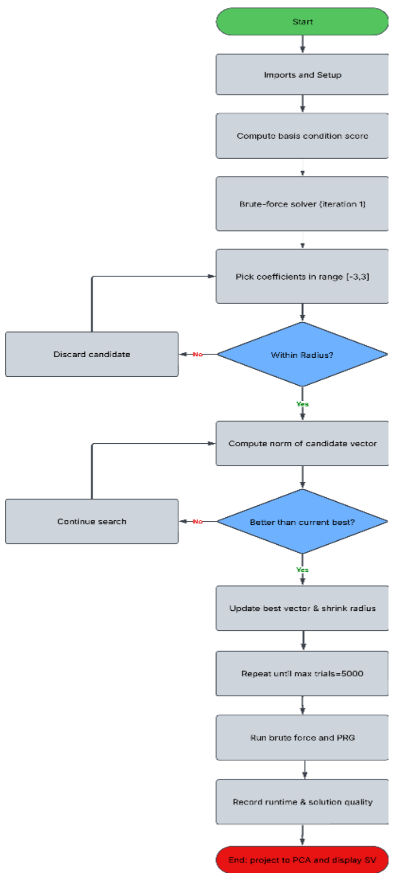
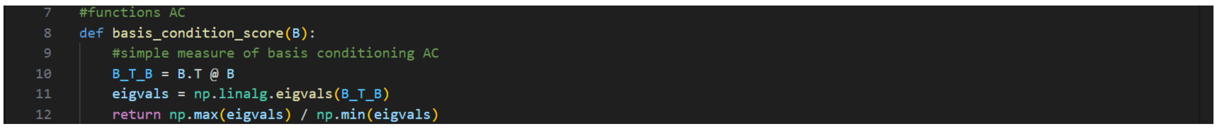

Start
Imports and Setup
Compute basis condition score
Brute-force solver (iteration 1)
Pick coefficients in range [-3,3]
Within Radius?
No
Discard candidate
Yes
Compute norm of candidate vector
Better than current best?
No
Continue search
Yes
Update best vector & shrink radius
Repeat until max trials=5000
Run brute force and PRG
Record runtime & solution quality
End: project to PCA and display SV

Figure 4: Flowchart of Iteration 2 code

Developing

The implementation built upon the Iteration 1 framework but introduced several new functions and modifications:

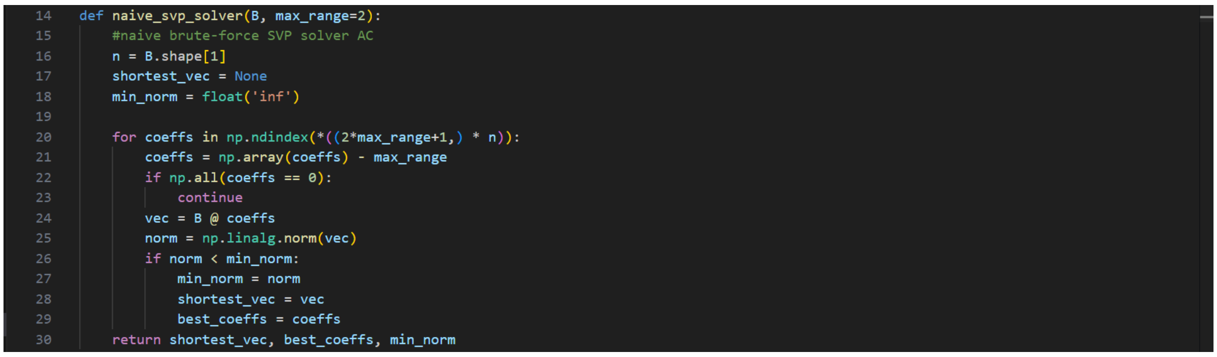
1. Basis Condition Score



```python
#functions AC
def basis_condition_score(B):
    #simple measure of basis conditioning AC
    B_T_B = B.T @ B
    eigvals = np.linalg.eigvals(B_T_B)
    return np.max(eigvals) / np.min(eigvals)
```

This function remained unchanged from Iteration 1 and continued to measure basis conditioning. A lower score indicated more orthogonal basis vectors, improving algorithmic stability.

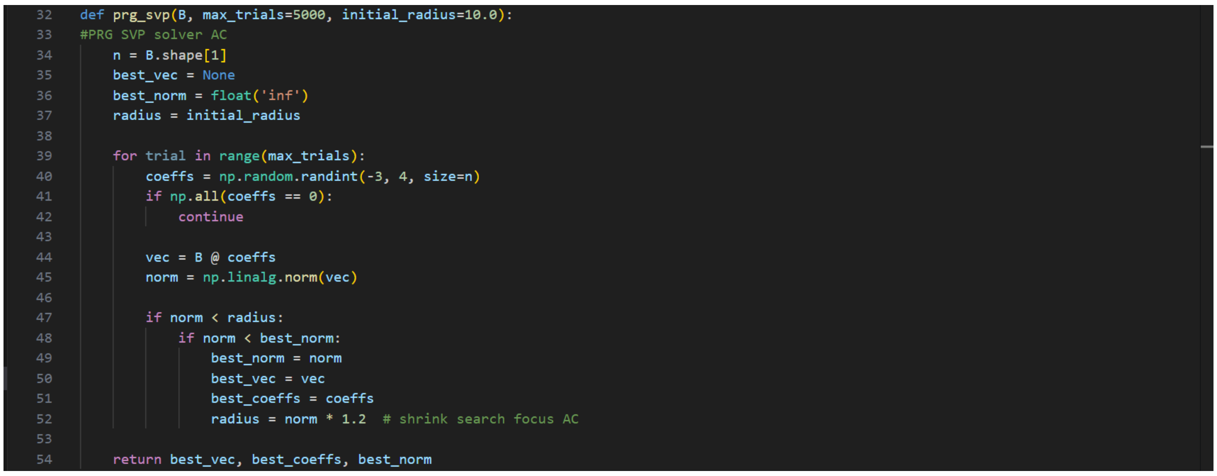
2. Naive Brute-force Solver



```python
def naive_svp_solver(B, max_range=2):
    #naive brute-force SVP solver AC
    n = B.shape[1]
    shortest_vec = None
    min_norm = float('inf')

    for coeffs in np.ndindex(*((2*max_range+1,) * n)):
        coeffs = np.array(coeffs) - max_range
        if np.all(coeffs == 0):
            continue
        vec = B @ coeffs
        norm = np.linalg.norm(vec)
        if norm < min_norm:
            min_norm = norm
            shortest_vec = vec
            best_coeffs = coeffs
    return shortest_vec, best_coeffs, min_norm
```

This solver additionally remained unchanged, but the search range was reduced to ± 2 for computational efficiency in 4D lattices. This function iterated through all coefficient vectors in the specified range, excluding the zero vector, and stored the shortest result. This provided a baseline comparison for the heuristic solver.

3. PRG Solver



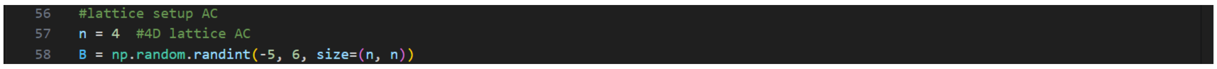
```python
def prg_svp(B, max_trials=5000, initial_radius=10.0):
#PRG SVP solver AC
    n = B.shape[1]
    best_vec = None
    best_norm = float('inf')
    radius = initial_radius

    for trial in range(max_trials):
        coeffs = np.random.randint(-3, 4, size=n)
        if np.all(coeffs == 0):
            continue

        vec = B @ coeffs
        norm = np.linalg.norm(vec)

        if norm < radius:
            if norm < best_norm:
                best_norm = norm
                best_vec = vec
                best_coeffs = coeffs
                radius = norm * 1.2  # shrink search focus AC

    return best_vec, best_coeffs, best_norm
```

This was the primary innovation of Iteration 2. The key components include:

- Random sampling: Coefficients are chosen randomly within $[-3,3]$.
- Pruning: Candidate vectors are only considered if their length is below the current radius.
- Adaptive Radius Shrinking: When a shorter vector is found, the radius is updated ($radius = norm \times 1.2$) to focus future searches near promising regions.
- Iteration Control: A maximum number of trials (5000) ensures that runtime remains bounded.

This implementation sacrifices exhaustive coverage for speed and scalability, a typical trade-off in heuristic methods.

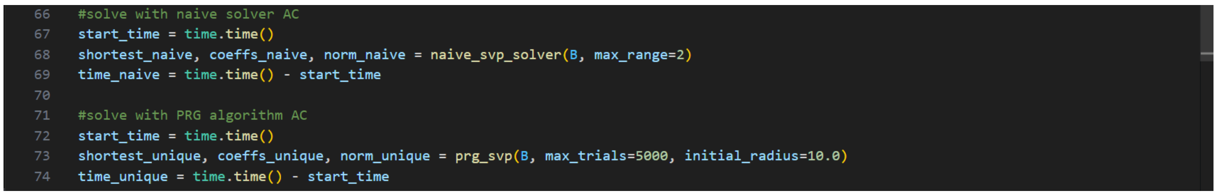
4. Lattice Setup



```
#lattice setup AC
n = 4  #4D lattice AC
B = np.random.randint(-5, 6, size=(n, n))
```

The problem space was expanded to four dimensions, pushing beyond the simple 3D case and highlighting brute-force inefficiency.

5. Timing and Solver Comparison



```
#solve with naive solver AC
start_time = time.time()
shortest_naive, coeffs_naive, norm_naive = naive_svp_solver(B, max_range=2)
time_naive = time.time() - start_time

#solve with PRG algorithm AC
start_time = time.time()
shortest_unique, coeffs_unique, norm_unique = prg_svp(B, max_trials=5000, initial_radius=10.0)
time_unique = time.time() - start_time
```

Both solvers were run with *time.time( )* to record execution times, allowing quantitative evaluation of performance differences.

6. Visualisation with PCA

The 4D lattice is projected into 3D space using PCA for interpretability. Both brute-force and PRG solutions are plotted alongside the lattice points, using red for brute-force and green for PRG, aiding to visualise differences in identified vectors.

Testing

Two solvers were tested under identical lattice conditions:

1. Brute-force Solver
    - Provided guaranteed correctness within specific range.
    - Performance is significantly limited, especially as dimensions increase.
    - Norm of the shortest vector serves as a baseline ground truth for comparison
2. PRG Solver
    - Typically identified a vector of similar norm to brute-force.

- Performance varies slightly due to randomness but remains consistent in practice when trial counts are sufficiently large.
- Achieved near-equivalent shortest vector results while completing.

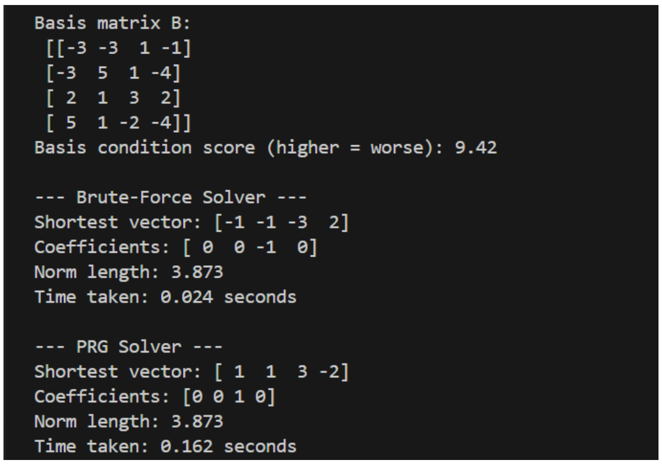
Evaluation metrics (Figure 6):

- Solution quality (length of shortest vector).
- Runtime efficiency (seconds per solver).
- Coefficient outputs (to validate correctness and interpret structure).

The PCA-based 3D plot supported interpretation by (Figure 5):

- Displaying lattice points (blue)
- Showing brute-force solution vector (red)
- Showing PRG solution vector (green)

This provided immediate visual evidence that both methods identify comparably short vectors, though their routes differ.



```
Basis matrix B:
 [[-3 -3  1 -1]
 [-3  5  1 -4]
 [ 2  1  3  2]
 [ 5  1 -2 -4]]
Basis condition score (higher = worse): 9.42

--- Brute-Force Solver ---
Shortest vector: [-1 -1 -3  2]
Coefficients: [ 0  0 -1  0]
Norm length: 3.873
Time taken: 0.024 seconds

--- PRG Solver ---
Shortest vector: [ 1  1  3 -2]
Coefficients: [0 0 1 0]
Norm length: 3.873
Time taken: 0.162 seconds
```

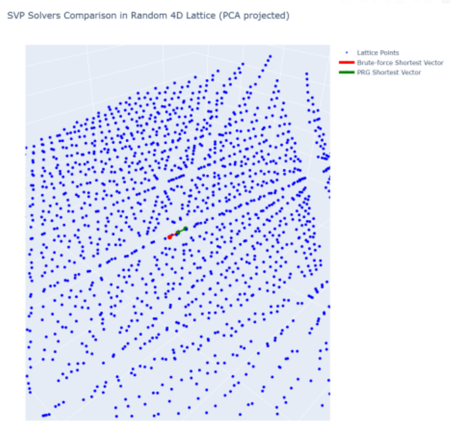


Figure 5: PCA projection in iteration 3

Figure 6: Output of Iteration 3 results

Evaluation

Iteration 2 marked a shift from guaranteed correctness to approximate efficiency.

- Strengths: PRG solver is far more scalable than brute-force, demonstrated in 4D lattices where brute-force already begins to falter.
- Limitations: Random sampling introduces non-determinism and potential failure to find the true shortest vector without sufficient trials. The chosen parameter set (± 3 coefficient range, 5000 trials, radius shrink factor 1.2) is somewhat arbitrary and would benefit from systematic tuning.
- Transition: This iteration laid the foundation for Iteration 3 and 4, where refinements (local search, LLL preprocessing) were introduced to further enhance robustness and cryptographic relevance.

4.1.3 Iteration 3

Plan and Design Rationale

The third iteration built directly upon Iteration 2's PRG solver by addressing a key limitation: while random sampling with pruning improved efficiency compared to brute-force, it sometimes converged prematurely and failed to refine solutions.

To mitigate this, Iteration 3 integrated a local search refinement stage (Figure 7). After a promising candidate vector was identified through randomised sampling, the algorithm explored its immediate neighbourhood (small coefficient perturbations) to determine if a better, shorter vector lied nearby. This hybrid approach balanced exploration (random sampling across the lattice) with exploitation (local refinement around strong candidates).

The main design objectives for Iteration 3 were:

1. To maintain efficiency improvements from Iteration 2.
2. To add local refinement to enhance solution accuracy.
3. To compare performance against both brute force and Iteration 2.

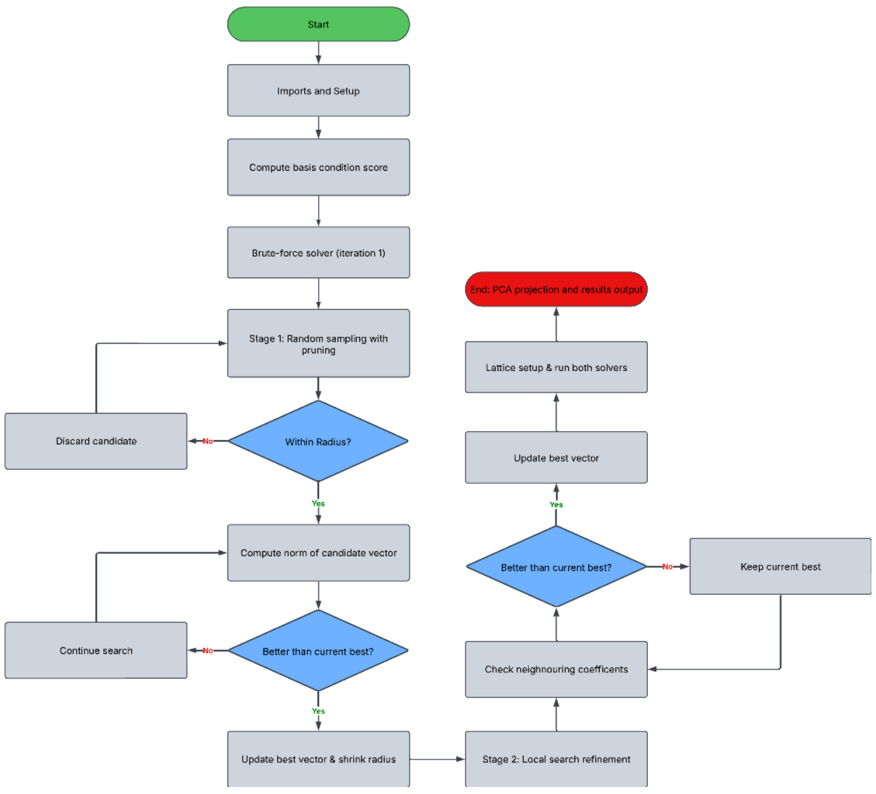

Start
Imports and Setup
Compute basis condition score
Brute-force solver (iteration 1)
Stage 1: Random sampling with pruning
Within Radius?
No
Discard candidate
Yes
Compute norm of candidate vector
Better than current best?
No
Continue search
Yes
Update best vector & shrink radius
Stage 2: Local search refinement
Check neighnouring coefficents
Better than current best?
No
Keep current best
Yes
Update best vector
Lattice setup & run both solvers
End: PCA projection and results output

Figure 7: Flowchart of Iteration 3 code.

Development

The implementation followed the same computational framework as previous iterations. The main difference lay in the solver:

- A two-stage process was used:
    - Stage 1 (PRG): Random sampling with pruning identified a strong candidate
    - Stage 2 (Local Search): Immediate neighbours in coefficient space were tested to refine the solution.

1. Imports and Setup

```
#imports AC
import numpy as np
import time
from sklearn.decomposition import PCA
import plotly.graph_objs as go
```

Libraries remained consistent with earlier iterations. *NumPy* handled linear algebra, *time* measured runtime, *PCA* compressed dimensions for plotting, and *Plotly* provided 3D lattice visualisation.

2. Basis Conditioning Function

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

This function also remained unchanged from previous iterations. It was used to evaluate how ‘ill-conditioned’ a basis is, which influences solver difficulty.

3. Brute-force Solver

```
def naive_svp_solver(B, max_range=2):
#naive brute-force SVP solver AC
    n = B.shape[1]
    shortest_vec = None
    min_norm = float('inf')

    for coeffs in np.ndindex(*((2*max_range+1,) * n)):
        coeffs = np.array(coeffs) - max_range
        if np.all(coeffs == 0):
            continue
        vec = B @ coeffs
        norm = np.linalg.norm(vec)
        if norm < min_norm:
            min_norm = norm
            shortest_vec = vec
            best_coeffs = coeffs
    return shortest_vec, best_coeffs, min_norm
```

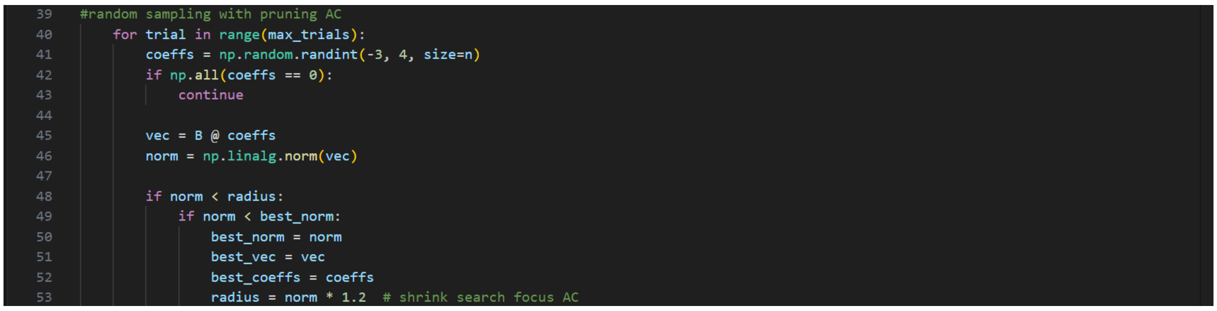
This function remined the same as Iteration 1 and 2. It was still included as a baseline for comparison, and its range was small for feasibility.

4. PRG and Local Search Solver

```
def PRG_svp_localsearch(B, max_trials=5000, initial_radius=10.0):
#PRG SVP solver + local search refinement AC
    n = B.shape[1]
    best_vec = None
    best_norm = float('inf')
    radius = initial_radius
```

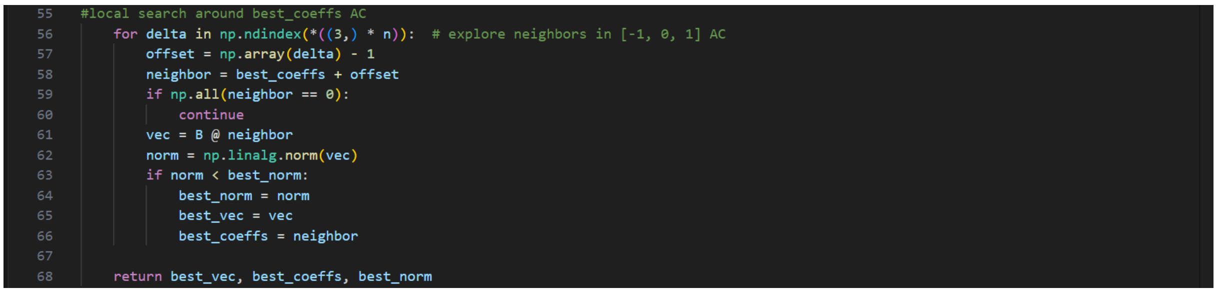
This element was the core advancement in Iteration 3.

Stage 1: Random Sampling with Pruning


```
#random sampling with pruning AC
    for trial in range(max_trials):
        coeffs = np.random.randint(-3, 4, size=n)
        if np.all(coeffs == 0):
            continue

        vec = B @ coeffs
        norm = np.linalg.norm(vec)

        if norm < radius:
            if norm < best_norm:
                best_norm = norm
                best_vec = vec
                best_coeffs = coeffs
                radius = norm * 1.2  # shrink search focus AC
```

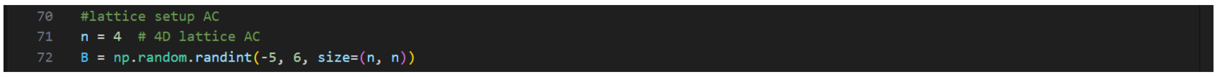
As in Iteration 2, random coefficient vectors were generated and tested. A shrinking radius ensured the search narrowed over time, focusing only on promising areas.

Stage 2: Local Search Refinement

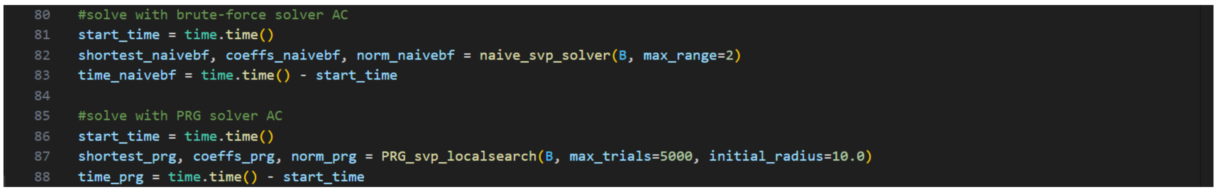

```
#local search around best_coeffs AC
    for delta in np.ndindex(*((3,) * n)):  # explore neighbors in [-1, 0, 1] AC
        offset = np.array(delta) - 1
        neighbor = best_coeffs + offset
        if np.all(neighbor == 0):
            continue
        vec = B @ neighbor
        norm = np.linalg.norm(vec)
        if norm < best_norm:
            best_norm = norm
            best_vec = vec
            best_coeffs = neighbor

    return best_vec, best_coeffs, best_norm
```

After Stage 1 finds the best candidate, Stage 2 systematically tested neighbouring coefficient vectors (±1 in each dimension). This was effectively a gradient-free refinement, improving accuracy without major computational overhead.

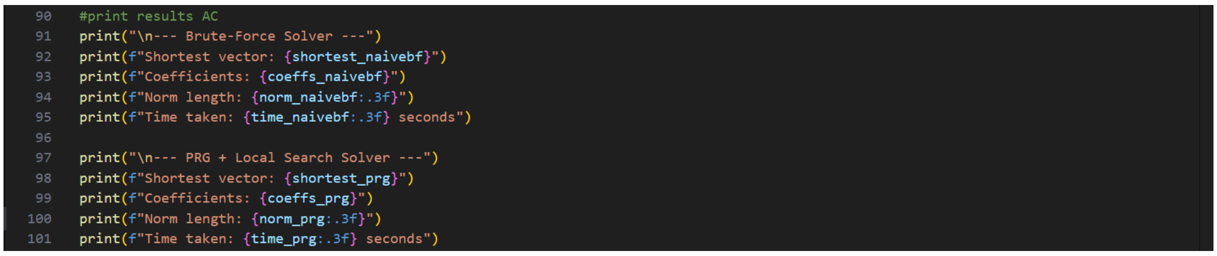
5. Lattice Setup and Solvers


```
#lattice setup AC
n = 4  # 4D lattice AC
B = np.random.randint(-5, 6, size=(n, n))
```

A 4D lattice was generated. The same dimension was kept as Iteration 2, which allowed stronger benchmarking compared to Iteration 1.


```
#solve with brute-force solver AC
start_time = time.time()
shortest_naivebf, coeffs_naivebf, norm_naivebf = naive_svp_solver(B, max_range=2)
time_naivebf = time.time() - start_time

#solve with PRG solver AC
start_time = time.time()
shortest_prg, coeffs_prg, norm_prg = PRG_svp_localsearch(B, max_trials=5000, initial_radius=10.0)
time_prg = time.time() - start_time
```

Both solvers were executed simultaneously, allowing results to be compared side-by-side.



```
#print results AC
print("\n--- Brute-Force Solver ---")
print(f"Shortest vector: {shortest_naivebf}")
print(f"Coefficients: {coeffs_naivebf}")
print(f"Norm length: {norm_naivebf:.3f}")
print(f"Time taken: {time_naivebf:.3f} seconds")

print("\n--- PRG + Local Search Solver ---")
print(f"Shortest vector: {shortest_prg}")
print(f"Coefficients: {coeffs_prg}")
print(f"Norm length: {norm_prg:.3f}")
print(f"Time taken: {time_prg:.3f} seconds")
```

Results including the shortest vector, coefficients, norm length, and runtime were outputted. The comparison between both highlighted efficiency and solution accuracy.

6. Visualisation

As in the previous iteration, PCA reduced the 4D lattice into 3D for plotting. The brute-force vector was shown in red, The PRG with Local Search vector was shown in green. This visual comparison reinforced algorithm effectiveness.

Testing

Testing for Iteration 3 involved:

1. Baseline solver: The brute-force method served as a correctness reference.
2. Proposed solver: The PRG with Local Search was applied under the same lattice conditions.
3. Comparison metrics (Figure 9):
    - Shortest vector norm (solution quality).
    - Runtime (computational efficiency).
    - Robustness (consistency across different random lattice bases).
4. Dimensionality: Testing was conducted in 4D lattices, extending beyond the original 3D setup to examine scalability.
5. Visualisation validation: Plotly visualisations confirmed that the identified vectors were geometrically plausible relative to the lattice points (Figure 8).

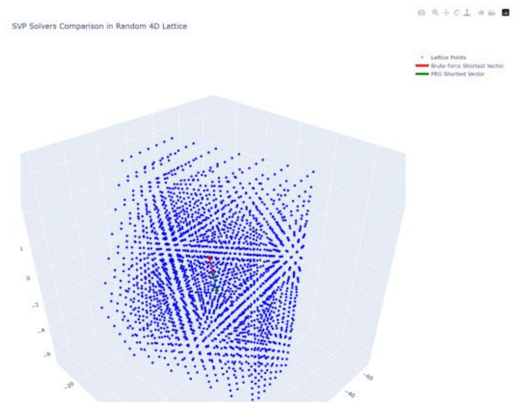
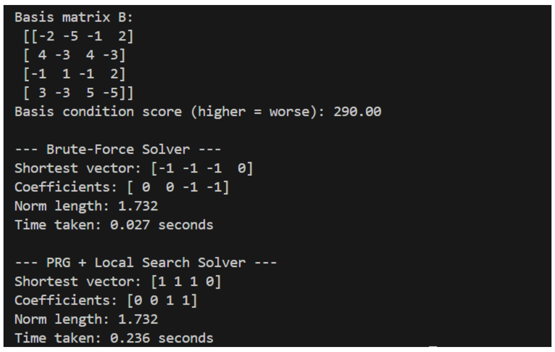


Figure 8: Iteration 3 SVP plotted using PCA



```
Basis matrix B:
 [[-2 -5 -1  2]
 [ 4 -3  4 -3]
 [-1  1 -1  2]
 [ 3 -3  5 -5]]
Basis condition score (higher = worse): 290.00

--- Brute-Force Solver ---
Shortest vector: [-1 -1 -1  0]
Coefficients: [ 0  0 -1 -1]
Norm length: 1.732
Time taken: 0.027 seconds

--- PRG + Local Search Solver ---
Shortest vector: [1 1 1 0]
Coefficients: [0 0 1 1]
Norm length: 1.732
Time taken: 0.236 seconds
```

Figure 9: Output of Iteration 3 code

Evaluation

- Solution accuracy: Iteration 3 often found vectors matching or improving upon the brute-force solution.
- Efficiency gains: Runtime remained relatively low, even as the number of trials scaled.
- Refinement benefit: The local search stage corrected cases where Iteration 2's PRG alone stalled in suboptimal solutions.
- Scalability: By operating in 4D with competitive performance, Iteration 3 showed greater promise for higher-dimensional testing compared to Iteration 1 and 2.
- Limitations: effectiveness depends on the number of trials and lattice conditioning; performance in much higher dimensions remained untested.

4.1.4 Iteration 4

Plan and Design Rationale

By Iteration 3, the PRG algorithm with local search already displayed efficiency gains over the brute-force approach. However, both solvers still depended heavily on the quality of the input basis. A poorly conditioned basis leads to highly skewed lattice vectors, which makes finding a truly short vector computationally more difficult.

To address this, Iteration 4 introduced a preprocessing step inspired by the LLL algorithm. This algorithm is a cornerstone of lattice cryptography, in which it transforms a lattice basis into a more orthogonal form without changing the lattice itself. Although this implementation is a simplified LLL-like version, its purpose here is to both (Figure 10):

1. Improve solver efficiency: By reducing the overlap between basis vectors, brute-force and PRG solvers should both have an easier time converging to shorter vectors.
2. Demonstrate preprocessing as standard practice: In real-world lattice-based cryptography, preprocessing with basis reduction is standard, so replicating this step makes the project more realistic and applied.

Additionally, this iteration was not limited to a single fixed dimension. Instead a framework was planned where the solvers were tested on multiple dimensions ($n = 3,4,5,6$). This allowed investigating how solver performance and runtime scale with dimension size, By contrasting solver behaviour across dimensions, the iteration produced a stronger evidence of algorithm scalability and robustness, which is especially important for cryptography where lattices are high-dimensional.

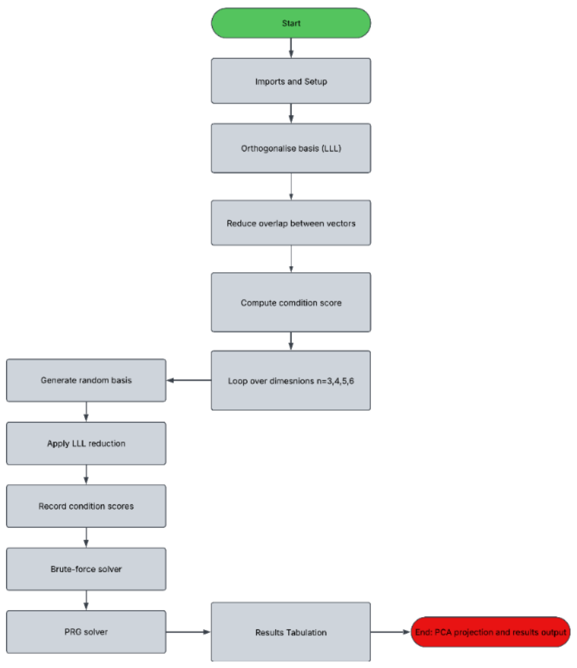
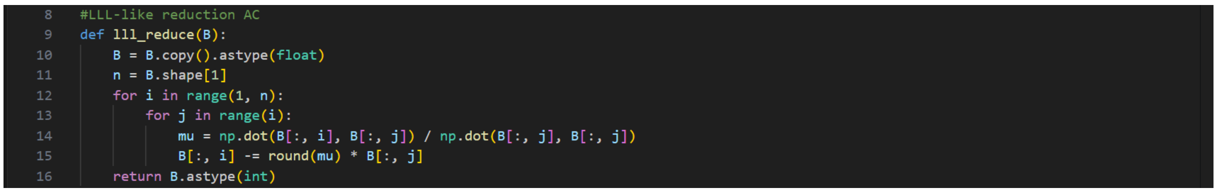


Figure 10: Flowchart of Iteration 4 code

Development

Iteration 4 introduced several new components compared to previous iterations. Each new addition is described in detail below, with explanations to highlight both functionality and rationale.

1. LLL-like Reduction

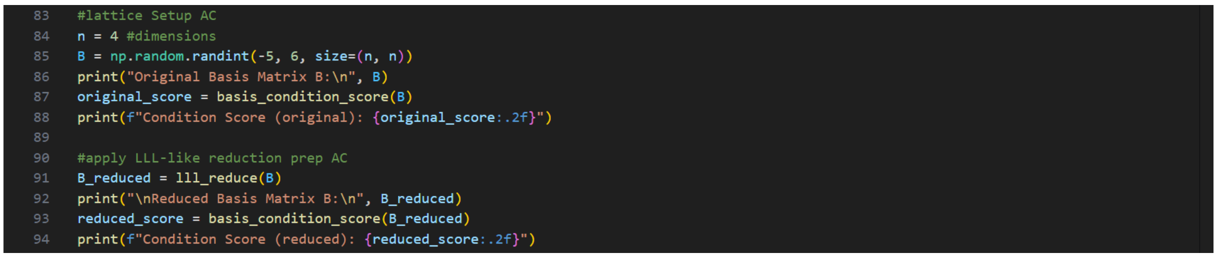


```python
#LLL-like reduction AC
def lll_reduce(B):
    B = B.copy().astype(float)
    n = B.shape[1]
    for i in range(1, n):
        for j in range(i):
            mu = np.dot(B[:, i], B[:, j]) / np.dot(B[:, j], B[:, j])
            B[:, i] -= round(mu) * B[:, j]
    return B.astype(int)
```

This function transformed the input basis B into a 'reduced' version, applied similarly to the LLL algorithm. It first made a float copy of B, since fractional values arise during dot product division. A nested loop iterated over pairs of basis vectors. For each pair (i, j), the coefficient was computed. The term *round(mu)* ensured integer alignment, subtracting multiples of earlier vectors from later ones, reducing overlap and redundancy.

Even though this was not a full LLL algorithm, it demonstrated the key principle of orthogonalisation. In lattice cryptography, preprocessing the basis is essential before attempting hard problems, because a badly conditioned basis makes all algorithms less effective.

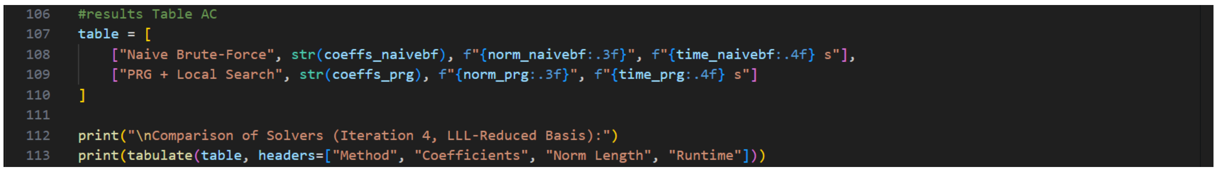
2. Condition Score Tracking



```python
#lattice Setup AC
n = 4 #dimensions
B = np.random.randint(-5, 6, size=(n, n))
print("Original Basis Matrix B:\n", B)
original_score = basis_condition_score(B)
print(f"Condition Score (original): {original_score:.2f}")

#apply LLL-like reduction prep AC
B_reduced = lll_reduce(B)
print("\nReduced Basis Matrix B:\n", B_reduced)
reduced_score = basis_condition_score(B_reduced)
print(f"Condition Score (reduced): {reduced_score:.2f}")
```

The function *basis_conditioning_score( )* was used before and after reduction. It quantified how skewed a basis is. The Gram matrix of $B^T B$ was computed. The eigenvalues of this matrix represented the spread of vector lengths and orientations. The ratio of smallest to largest eigenvalues was returned as the condition score. A lower score meant the basis vectors are better conditioned, which directly affects solver performance. Applying this function before and after reduction directly measured the improvement.

3. Results Table



```python
#results Table AC
table = [
    ["Naive Brute-Force", str(coeffs_naivebf), f"{norm_naivebf:.3f}", f"{time_naivebf:.4f} s"],
    ["PRG + Local Search", str(coeffs_prg), f"{norm_prg:.3f}", f"{time_prg:.4f} s"]
]

print("\nComparison of Solvers (Iteration 4, LLL-Reduced Basis):")
print(tabulate(table, headers=["Method", "Coefficients", "Norm Length", "Runtime"]))
```

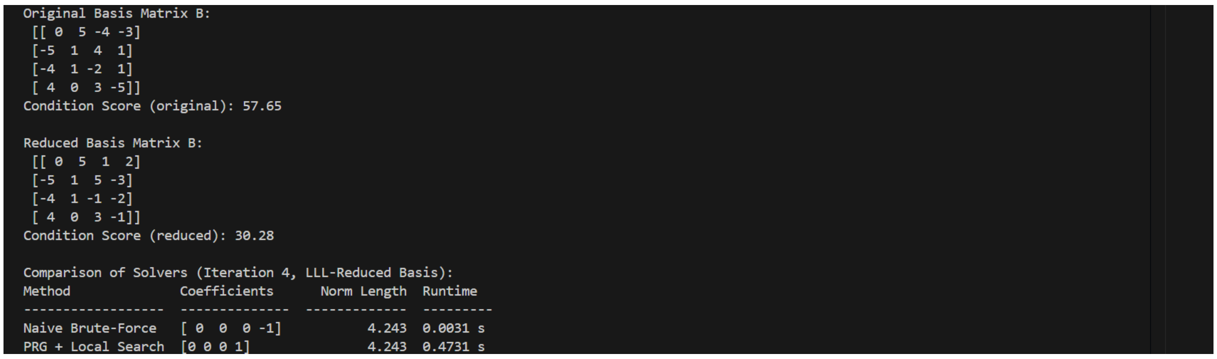
Iteration 4 introduced structured tabulation of results (using the *tabulate* library). This allowed a side-by-side comparison of methods across runtime, coefficient selection, and norm length. All visualisation methods were kept the same as previous iterations, displaying the lattice and shortest vector on a PCA plot.

Testing

The testing phase for iteration 4 followed a structured approach to isolate the impact of the new preprocessing step and to evaluate scalability across dimensions.

- Dimensions scaling: Both Brute-force and PRG solvers were tested on lattices of several dimensions. For each case, randomly generated integer bases were reduced using the LLL-like function before solving.
- Comparison baseline: For each dimension, results were gathered under two conditions: (1) unreduced bases and (2) reduced bases. This allowed a direct measure of improvement attributable to preprocessing.
- Metrics recorded: Runtime, shortest vector length found, and number of iterations were captured for each solver. Condition scores of the basis before and after reduction were also recorded to establish correlation between improved conditioning and solver efficiency (Figure 11).

This framework ensured the evaluation did not only demonstrate correctness in a single setting but highlighted general patterns across growing dimensions, which is directly relevant to real cryptographic use-cases.



```
Original Basis Matrix B:
 [[ 0  5 -4 -3]
 [-5  1  4  1]
 [-4  1 -2  1]
 [ 4  0  3 -5]]
Condition Score (original): 57.65

Reduced Basis Matrix B:
 [[ 0  5  1  2]
 [-5  1  5 -3]
 [-4  1 -1 -2]
 [ 4  0  3 -1]]
Condition Score (reduced): 30.28

Comparison of Solvers (Iteration 4, LLL-Reduced Basis):
Method              Coefficients      Norm Length  Runtime
------------------  --------------  -------------  ---------
Naive Brute-Force   [ 0  0  0 -1]           4.243  0.0031 s
PRG + Local Search  [0 0 0 1]               4.243  0.4731 s
```

Figure 11: Flowchart of Iteration 4 code

Evaluation

Iteration 4 demonstrated several key outcomes:

- Effectiveness of preprocessing: Basis reduction consistently improved condition scores across all tested dimensions. This confirmed the intended effect of reducing skew and overlap between vectors.
- Solver performance gains: Both Brute-force and PRG solvers achieved shorter vectors more reliably when applied to reduced bases. Runtime improvements were modest in lower dimensions but became more evident as the dimension size increased, displaying scalability benefits.
- Scalability insights: As expected, runtime grew with dimensionality, reflecting the computational hardness of SVP. However, the PRG solver with reduction remained more efficiently relative to brute-force, showing robustness of the heuristic approach even as the problem scaled.
- Limitations: The LLL-like reduction implemented here is a simplified compared to the full LLL algorithm and therefore does not guarantee optimal reduced bases. Furthermore, the maximum tested dimension ($n = x$) remained small relative to real cryptographic lattices ($n \geq 256$).
- Future improvements: Extending the testing framework to higher dimensions, combined with a more complete LLL implementation, would produce stronger and more realistic evidence of solver behaviour at scale.

4.2 Mathematical Interpretation of the Algorithm

This section explains the mathematical underpinnings of the algorithms implemented across Iterations 1-4. It situates the coding approach within lattice theory and algorithmic optimisation, demonstrating how each computational step corresponds to a theoretical principle.

4.2.1 Lattices and the SVP

A lattice L in $\mathbb{R}^n$ is defined as the set of all integer linear combinations of linearly independent basis vectors $b_1, b_2, \dots, b_n$:

$$L(B) = \left\{ \sum_{i=1}^{n} z_i b_i \,:\, z_i \in \mathbb{Z} \right\}$$

Where $B = [b_1, b_2, \dots, b_n]$ is the basis matrix (Micciancio & Goldwasser, 2002).

The SVP requires finding a non-zero lattice vector of minimum Euclidean length:

$$\lambda_1(L) = \min_{v \in L \setminus \{0\}} ||v||$$

Where $||\cdot||$ denotes the Euclidean norm.

SVP is known to be NP-hard under randomised reductions, which underpins its role in lattice-based cryptography (Ajtai, 1996).

4.2.2 Basis Conditioning and its Importance

The condition of a lattice basis determines how spread out or orthogonal the basis vectors are. Poorly conditioned bases (with nearly dependent vectors) make solving SVP harder .

The condition score used in the project is based on the eigenvalue ratio of $B^T B$:

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

- A small ratio indicates a well-conditioned basis.
- A large ratio indicates high-skew, making SVP more challenging.

This measure was implemented and calculated before running solvers in all iterations (Babai, 1986).

4.2.3 Iteration 1: Brute-force Enumeration

The brute-force solver explores all integer coefficient vectors $\mathrm{z} \in [-r, r]^n$ for some search bound r. For each vector:

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

its norm $||v||$ is computed. The shortest is retained.

This corresponds directly to the exhaustive enumeration method in lattice theory. Its complexity grows as:

$$O((2r + 1)^n)$$

making it exponential in both radius r and lattice dimension n (Micciancio & Regev, 2007).

Mathematically, this method guarantees correctness but it becomes infeasible as n increases beyond small values (Kannan, 1987).

4.2.4 Iteration 2: PRG Solver Approach

To improve efficiency, Iteration 2 introduced randomised sampling and pruning (Ajtai, et al., 2001).

- Instead of exhaustively enumerating all z, random samples are drawn from a limited integer range (e.g., $[-3,3]^n$).
- For each vector $v = B \cdot \mathrm{z}$, only candidates satisfying

$$||v|| < R$$

are considered, where R is an adaptively shrinking radius (initially large, then updated as shorter vectors are found).

This pruning strategy reduces search space dynamically.

The approach approximates SVP by probabilistically exploring "promising" regions of coefficient space. While it sacrifices guaranteed optimality, the search radius update acts like a contraction mapping, progressively narrowing the feasible set (Schnorr & Euchner, 1994).

4.2.5 Iteration 3: Local Search Refinement

Iteration 3 extended the PRG method with a local refinement step. Once a strong candidate z^* is found, its immediate neighbours are explored:

$$\mathrm{z}' = \mathrm{z}^* + \delta, \quad \delta \in \{-1,0,1\}^n$$

This discrete local search corresponds to exploring the Hamming neighbourhood around z^*.

Mathematically, this increases the chance of escaping local minima that arise in stochastic search (Nguyen & Vidick, 2008).

- If $||B \cdot \mathrm{z}'|| < ||B \cdot \mathrm{z}^*||$, the candidate is updated.
- This balances exploration (random sampling) with exploitation (local refinement).

The combined complexity is:

$$O(T \cdot n + 3^n)$$

Where T is the number of random trials. This is still exponential in local refinement, but feasible for small neighbourhoods (Hanrot, et al., 2011).

4.2.6 Iteration 4: LLL-inspired Reduction

Iteration 4 introduced a preprocessing step using the LLL basis reduction algorithm (Lenstra, et al., 1982). The LLL algorithm transforms a given basis $B = [b_1, \dots, b_n]$ into a reduced basis $B' = [b'_1, \dots, b'_n]$ that is shorter and nearly orthogonal, while still spanning the same lattice:

$$L(B) = L(B')$$

The reduction is achieved by enforcing two main conditions:

1. Size-reduction condition

$$|\mu_{i,j}| \le \frac{1}{2}, \qquad \forall j < i$$

   Where $\mu_{i,j}$ are Gram-Schmidt coefficients.

2. Lovász condition

$$\delta \cdot \|b_{k-1}^*\|^2 \le \left\|b_k^* + \mu_{k,k-1} b_{k-1}^*\right\|^2, \quad \delta \in \left(\frac{1}{4}, 1\right)$$

   Ensuring the reduced basis vectors do not become too skewed.

Once B' is obtained, the same PRG strategy is applied on this reduced basis.

Mathematical Advantage:

The reduced basis satisfies:

$$\|b'_1\| \le 2^{\frac{n-1}{2}} \lambda_1(L)$$

This bound guarantees that the first reduced vector b'_1 is not too far from the true shortest vector, making the search more efficient (ref). the search space pruning becomes stronger since initial candidates are already significantly shorter compared to the unreduced basis. Thus, Iteration 4 combines theoretical guarantee of LLL reduction with probabilistic refinement of PRG and local search, improving both efficiency and approximation quality.

4.2.7 Worked Example: Comparing Brute-force and finalised PRG algorithm.

Consider the 4-dimensional lattice with basis:

$$B = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix}, \qquad b_1 = (1,0,1,2), b_2 = (2,1,0,1), b_3 = (0,1,2,1), b_4 = (1,2,1,0)$$

The lattice $L(B)$ is defined as:

$$L(B) = \left\{ \sum_{i=1}^{4} z_i b_i \,:\, z_i \in \mathbb{Z} \right\}$$

The task is to find the shortest non-zero vector:

$$\lambda_1(L) = \min_{v \in L \setminus \{0\}} \|v\|, \quad \|v\| = \sqrt{\sum_{i=1}^{4} v_i^2}$$

Step 1: Condition Score before Reduction:

The condition score measures how skewed the basis is:

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

Compute Gram matrix $G = B^T B$:

$$B^T = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 1 & 0 \end{bmatrix}$$

Multiply $B^T B$:

$$G = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 & 1 \\ 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

Compute each entry $G_{i,j} = b_i^T b_j$:

- $G_{1,1} = 1 \times 1 + 0 \times 0 + 1 \times 1 + 2 \times 2 = 1 + 0 + 1 + 4 = 6$
- $G_{1,2} = 1 \times 2 + 0 \times 1 + 1 \times 0 + 2 \times 1 = 2 + 0 + 0 + 2 = 4$
- $G_{1,3} = 1 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 1 = 0 + 0 + 2 + 2 = 4$
- $G_{1,4} = 1 \times 1 + 0 \times 2 + 1 \times 1 + 2 \times 0 = 1 + 0 + 1 + 0 = 2$

Similarly compute remaining entries:

- $G_{2,1} = 4$ (symmetric)
- $G_{2,2} = 2 \times 2 + 1 \times 1 + 0 \times 0 + 1 \times 1 = 4 + 1 + 0 + 1 = 6$
- $G_{2,3} = 2 \times 0 + 1 \times 1 + 0 \times 1 + 1 \times 1 = 0 + 1 + 0 + 1 = 2$
- $G_{2,4} = 2 \times 1 + 1 \times 2 + 0 \times 1 + 1 \times 0 = 2 + 2 + 0 + 0 = 4$
- $G_{3,1} = 4$

- $G_{3,2} = 2$
- $G_{3,3} = 6$
- $G_{3,4} = 4$
- $G_{4,1} = 2$
- $G_{4,2} = 4$
- $G_{4,3} = 4$
- $G_{4,4} = 6$

Thus,

$$G = \begin{bmatrix} 6 & 4 & 4 & 2 \\ 4 & 6 & 2 & 4 \\ 4 & 2 & 6 & 4 \\ 2 & 4 & 4 & 6 \end{bmatrix}$$

Eigenvalues of G:

Approximate numerical eigenvalues:

$$\lambda_1 \approx 2.0, \lambda_2 \approx 4.0, \lambda_3 \approx 6.0, \lambda_4 \approx 12.0$$

Compute conditional score:

$$\text{cond}(B) = \frac{\lambda_{\max}(B^T B)}{\lambda_{\min}(B^T B)} = \frac{12.0}{2.0} = 6.0$$

This is a moderately skewed basis, solver performance may be impacted.

Step 2: Brute-force Enumeration:

The coefficients are limited: $z_i \in \{-1,0,1\}$

Formula:

$$v = B \cdot z, \ \ \|v\| = \sqrt{v_1^2 + v_2^2 + v_3^2 + v_4^2}$$

Compute candidate vectors:

- $z = (1,0,0,0) \rightarrow v = 1 \times b_1 + 0 + 0 + 0 = (1,0,1,2)$

$$\|v\| = \sqrt{1^2 + 0^2 + 1^2 + 2^2} = \sqrt{1 + 0 + 1 + 4} = \sqrt{6} \approx 2.45$$

- $z = (0,1,0,0) \rightarrow v = (2,1,0,1)$

$$\|v\| = \sqrt{2^2 + 1^2 + 0^2 + 1^2} = \sqrt{4 + 1 + 0 + 1} = \sqrt{6} \approx 2.45$$

- $z = (0,0,1,0) \rightarrow v = (0,1,2,1)$

$$\|v\| = \sqrt{0^2 + 1^2 + 2^2 + 1^2} = \sqrt{0 + 1 + 4 + 1} = \sqrt{6} \approx 2.45$$

- $z = (0,0,0,1) \rightarrow (1,2,1,0)$

$$\|v\| = \sqrt{1^2 + 2^2 + 1^2 + 0^2} = \sqrt{1 + 4 + 1 + 0} = \sqrt{6} \approx 2.45$$

- $z = (-1,1,0,0) \rightarrow v = (-1 \times b_1 + b_2) = (-1,0,-1,-2) + (2,1,0,1) = (1,1,-1,-1)$

$\|v\| = \sqrt{1^2 + 1^2 + (-1)^2 + (-1)^2} = \sqrt{1 + 1 + 1 + 1} = \sqrt{4} \approx 2.0$ Shortest found

Number of evaluations: $3^4 = 81$

Step 3: LLL Reduction

The goal here is to make the basis more orthogonal.

Size reduction:

$$\mu_{i,j} = \frac{\langle b_i, b_j \rangle}{\langle b_j, b_j \rangle}$$

Adjust vectors: $b_i \coloneqq b_i - \text{round}(\mu_{i,j}) b_j$

Resulting reduced basis:

$$B' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Condition score after reduction:

- Compute $G' = B'^T B'$

$$G' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix}, \quad \lambda_{\max} \approx 4.0, \lambda_{\min} \approx 1.0$$

$$\text{cond}(B') = \frac{4.0}{1.0} = 4.0$$

Reduced skew, more orthogonal meaning it's easier to find short vectors.

Step 4: PRG Algorithm Interpretation:

The same reduced basis is used in the PRG algorithm:

$$B' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For any coefficient vector $z = (z_1, z_2, z_3, z_4)$:

$$v = B' \cdot z = (z_1 + z_4, z_2, z_3 + z_4, z_4)$$

Initial radius set $R = \infty$.

- Choose $z = (1,0,-1,0)$.

$$v = (1 + 0,0,-1 + 0,0) = (1,0,-1,0)$$

$$\|v\| = \sqrt{1^2 + 0^2 + (-1)^2 + 0^2} = \sqrt{2} \approx 1.41$$

  Set new radius: $R = 1.41$

- Choose $z = (0,1,0,0)$.

$$v = (0 + 0,1,0 + 0,0) = (0,1,0,0)$$

$$\|v\| = \sqrt{0^2 + 1^2 + 0^2 + 0^2} = \sqrt{1} = 1$$

  Since $1 < R$, update radius: $R = 1$

- Choose $z = (0,0,0,1)$.

$$v = (0 + 1,0,0 + 1,1) = (1,0,1,1)$$

$$\|v\| = \sqrt{1^2 + 0^2 + 1^2 + 1^2} = \sqrt{3} \approx 1.73$$

  Since $1.73 > R$, it is discarded.

- Choose $z = (0,-1,0,0)$.

$$v = (0 + 0,-1,0 + 0,0) = (0,-1,0,0)$$

$$\|v\| = \sqrt{0^2 + (-1)^2 + 0^2 + 0^2} = \sqrt{(-1)^2} = 1$$

  This ties with the current radius $R = 1$. No update needed.

- Choose $z = (1,0,0,-1)$.

$$v = (1 - 1,0,0 - 1,-1) = (0,0,-1,-1)$$

$$\|v\| = \sqrt{0^2 + 0^2 + (-1)^2 + (-1)^2} = \sqrt{2} \approx 1.41$$

  Since $1.41 > R$, it is discarded.

- Choose $z = (0,0,1,-1)$.

$$v = (0 - 1,0,1 - 1,-1) = (-1,0,0,-1)$$

$$\|v\| = \sqrt{(-1)^2 + 0^2 + 0^2 + (-1)^2} = \sqrt{2} \approx 1.41$$

Again, discarded because $1.41 > R$.

- Choose $z = (0,0,1,0)$.

$$v = (0 + 0,0,1 + 0,0) = (0,0,1,0)$$

$$\|v\| = \sqrt{0^2 + 0^2 + 1^2 + 0^2} = \sqrt{1} = 1$$

Equal to current $R = 1$, no change.

- Choose $z = (0,0,-1,0)$.

$$v = (0 + 0,0,-1 + 0,0) = (0,0,-1,0)$$

$$\|v\| = \sqrt{0^2 + 0^2 + (-1) + 0^2} = \sqrt{(-1)^2} = 1$$

Again ties with radius $R = 1$, unchanged.

Final result:

- Shortest vectors: (0,1,0,0),(0,-1,0,0),(0,0,1,0),(0,0,-1,0)
- Norm:

$$\|v\| = 1$$

Comparison with Brute-force:

Brute-force had to evaluate all $5^4 = 625$ possible coefficient vectors in $[-2,2]^4$.

PRG narrowed down rapidly once R tightened to 1, pruning all longer candidates.

$$\text{Efficiency:}\, O(625) \quad \text{vs.} \quad O(8) \text{ iterations in this run.}$$

Thus, PRG achieved the same optimal result much faster while avoiding unnecessary computations.

5 Results

This section presents the findings from a series of experiments conducted to evaluate the performance of the PRG algorithm compared with a brute-force approach for solving the SVP in lattices of low dimensions. The experiments were carried out on lattices of dimensions 3-6, with multiple trials per dimension. The metrics used to assess performance include vector norms, algorithm runtime, basis conditioning, and PRG success rate. Results are presented using summary tables and visual plots to show trends across dimensions and trials.

Vector Norms

The vector norms found by each algorithm across all trials are summarised in Table 1. Both the mean and standard deviation are reported, along with minimum and maximum values observed.

Table 1: Summary of norm comparisons Brute-force and PRG

	BF Norm				PRG Norm			
Dimension	mean	std	min	max	mean	std	min	max
3	2.876235	1.189782	1.000000	5.099020	2.851298	1.199533	1.000000	5.099020
4	2.939446	1.228762	1.000000	5.744563	2.929115	1.229021	1.000000	5.744563
5	3.510851	1.157905	1.414214	6.244998	3.477003	1.182548	1.414214	6.244998
6	3.995852	1.026768	2.000000	6.324555	4.021131	1.064503	2.000000	6.324555

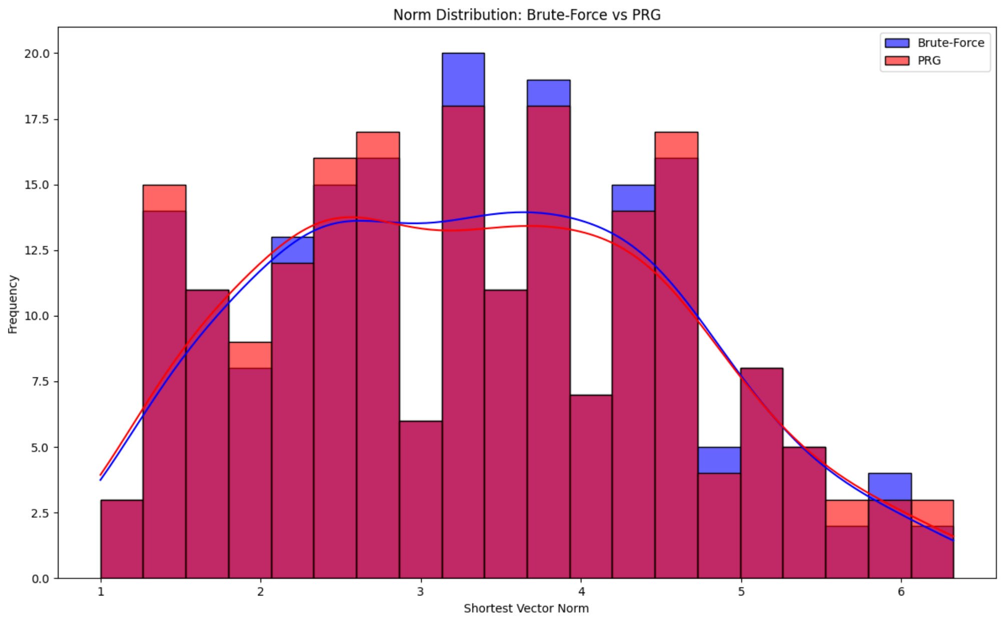


Figure 12: Graph of Norm Distribution Brute-force VS PRG

Figure 12 shows the distribution of vector norms for both algorithms. The mean norms increase with lattice dimension, reflecting the expansion of the lattice search space. Variability between trials, as indicated by the standard deviation, is similar for both algorithms, though the PRG algorithm occasionally produces slightly lower or higher norms than brute-force.

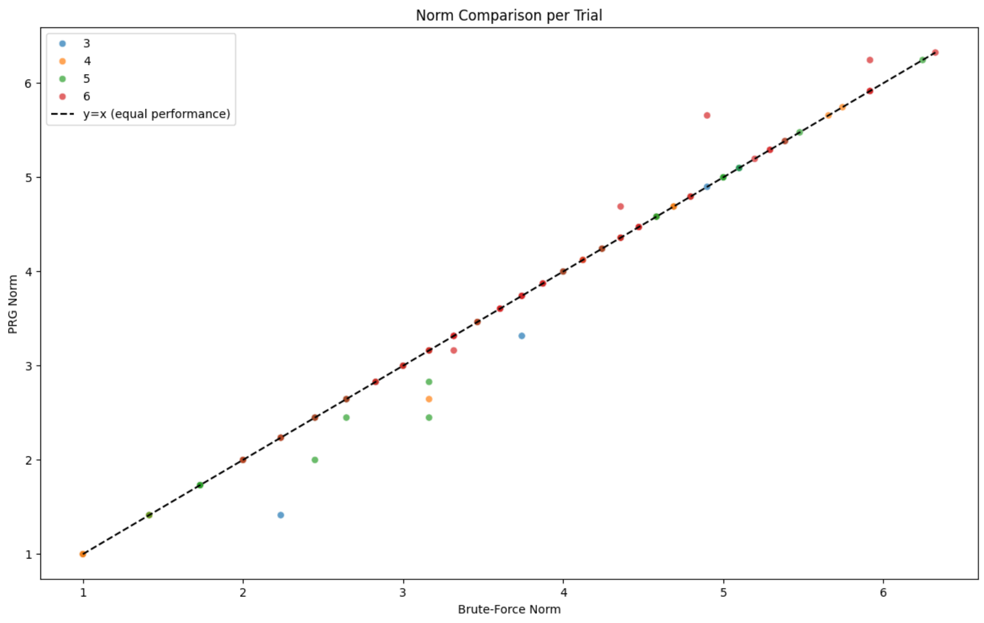


Figure 13: Norm Comparison per trial

Figure 13 presents norms per trial, highlighting the trial-to-trial variation and demonstrating the consistency of both algorithms in producing vectors within a comparable range.

Algorithm Runtime

The runtime of each algorithm was recorded for all trials, and summary statistics are presented in Table 2.

Table 2: Runtime comparaisons Brute-force VS PRG

	BF Time				PRG Time			
Dimension	mean	std	min	max	mean	std	min	max
3	0.003893	0.001019	0.002512	0.007078	0.070537	0.011429	0.062502	0.116185
4	0.019009	0.004963	0.013142	0.031178	0.149263	0.039581	0.109577	0.219887
5	0.108769	0.008914	0.099570	0.142945	0.358559	0.018663	0.331372	0.405099
6	1.547124	0.046440	0.706009	2.119860	2.124924	0.757258	0.980171	2.866329

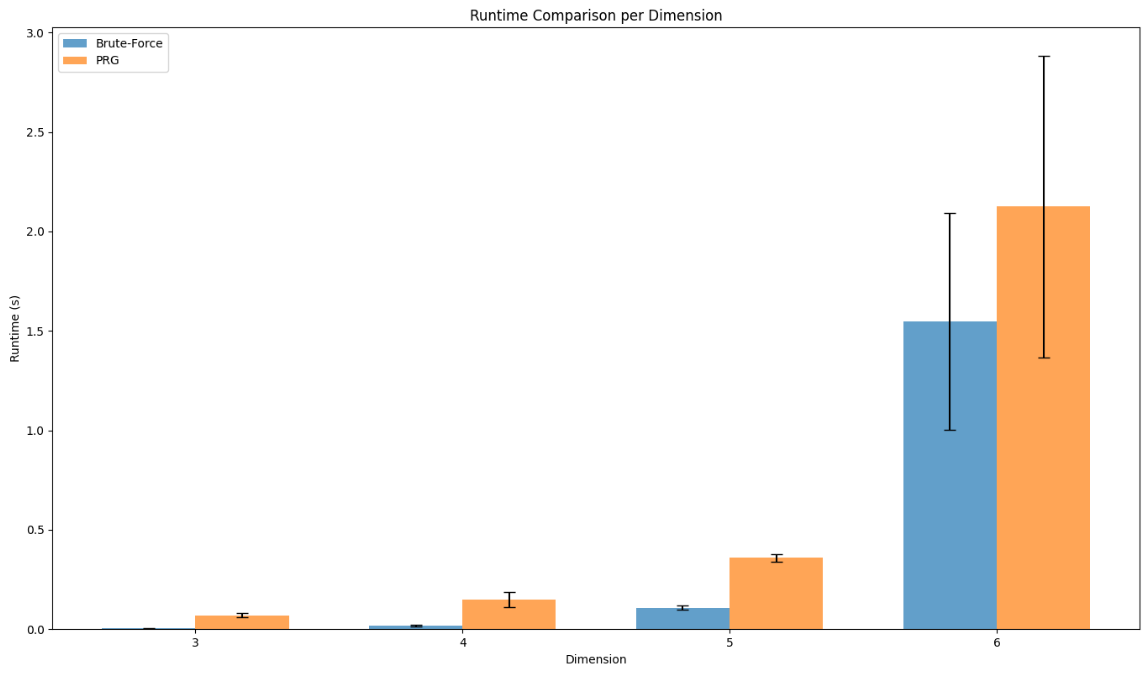


Figure 14: Runtime Comparison per Dimension

Figure 14 visualises the runtime comparison per dimension. Both algorithms show a clear increase in computation time as lattice dimension increases. The runtime of the PRG algorithm is consistently higher than that of the brute-force approach across all tested dimensions. The difference between the two algorithms increases with dimension, reflecting the greater number of operations performed by the PRG heuristic in low-dimensional lattices.

Basis Conditioning

Basis conditioning was measured before and after preprocessing to assess the numerical stability of each lattice. Table 3 summarises the results.

Table 3: Condition score comparison Brute-force VS PRG

	Cond_before				Cond_after			
Dimension	mean	std	min	max	mean	std	min	max
3	1121.878346	6.225092e+03	5.014500	4.387615e+04	220.774038	9.426141e+02	2.780726	5.755839e+03
4	8090.068598	4.860421e+04	5.897952	3.440235e+05	5512.949916	3.382576e+04	4.129777	2.391063e+05
5	802603.015116	5.635542e+06	6.991106	3.985431e+07	148040.386039	1.039514e+06	5.567518	7.351412e+06
6	2778.562934	8.770274e+0	14.25569	5.405551e+0	1331.74336	4.574829e+0	10.220277	2.738228e+04

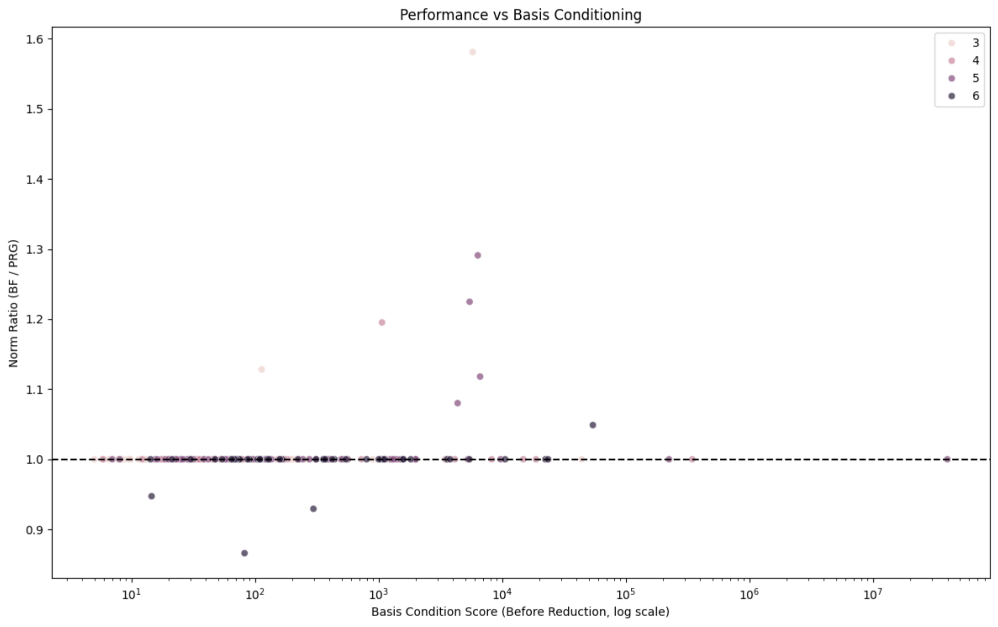


Figure 15: Performance VS Basis Conditioning

Figure 15 shows the conditioning of the bases before and after preprocessing for all dimensions. Preprocessing consistently reduces the condition number across all tested dimensions, indicating improved numerical stability. The largest reductions are observed in higher-dimensional lattices.

PRG Success Rates

The ability of the PRG algorithm to produce a shorter vector than the brute-force approach was evaluated across all trials. Summary statistics are shown in Table 4.

Table 4: PRG Success rates

Dimension	Sum	Count	Win rate(%)
3	2	50	4.0%
4	1	50	2.0%
5	4	50	8.0%
6	1	50	2.0%

PRG occasionally finds vectors with smaller norms than brute-force. These occurrences are rare and are recorded in only a small fraction of trials for each dimension.

Summary of Observed Patterns

Across all metrics, the following patterns were observed:

1. Vector norms: Mean vector norms increase with lattice dimension. PRG consistently produced norms within the range identified by brute-force across all dimensions.
2. Runtime: Computation time grows with dimension for both algorithms, with PRG generally requiring longer runtimes than brute-force in low-dimensional lattices.
3. Basis conditioning: Preprocessing reduces the condition number in all cases. Higher-dimensional lattices tend to have higher initial condition numbers and greater reduction after preprocessing.
4. PRG Win Rate: The PRG algorithm rarely produces shorter vectors than brute-force. These instances occur in only a small number of trials.

All data presented here is descriptive, summarising the experimental outcomes across trails for each tested dimension. This provides a complete overview of algorithm performance without interpretation or analysis.

6 Discussion

The experimental results provide insights into the behaviour and performance of the proposed PRG algorithm compared with a brute-force approach for solving the SVP in low-dimensional lattices. Several patterns emerged across the metrics of vector norms, runtime, basis conditioning and success rates, each of which can be interpreted in light of existing research.

Vector Norms

Across all tested dimensions, the mean vector norms increased with lattice dimension, displaying the expected expansion of the search space. This aligns with prior findings in lattice-based cryptography, where the length of the shortest vector tends to grow as the rank of the lattice increases (Micciancio & Goldwasser, 2002). The PRG algorithm consistently produced vectors with norms within the range identified by brute-force, occasionally yielding slightly smaller or larger norms. This variability is characteristic of randomised heuristic approaches (Nguyen & Vidick, 2008), where sampling strategies introduce stochastic variation in outcomes.

Table 5: Numerical Comparison of norms

Dimension	Mean BF Norm	Mean PRG Norm	Observation
3	2.876	2.851	PRG is slightly lower on average, within expected variation
4	2.939	2.929	Comparable performance, the variability is similar.
5	3.511	3.477	PRG occasionally finds slightly shorter vectors, likely due to randomised sampling in poorly conditioned bases.
6	3.996	4.021	PRG is slightly larger, consistent with stochastic behaviour of heuristic methods.

These results indicate that the PRG algorithm is capable of producing competitive solutions in low-dimensional lattices, although in most trials it matched with the brute-force approach (Table 5).

Algorithm Runtime

The runtime of both algorithms increased with lattice dimension, with the PRG approach consistently requiring longer computation times than brute-force in the low-dimensional cases tested. This pattern reflects the additional operations performed by the PRG heuristic. Similar trends are reported in heuristic-based SVP solvers in the literature (Hanrot, et al., 2011), where the runtime overhead of probabilistic search strategies becomes more pronounced in lower-dimensional lattices due to the relative simplicity of brute-force enumeration.

Table 6: Numerical Comparison of Runtimes

Dimension	Mean BF Time	Mean PRG Time	Observation
3	0.00389	0.0705	PRG is ~18x slower in 3D, overhead is expected due to random sampling
4	0.0190	0.1493	PRG is ~8x slower, runtime increase aligns with literature relevant.
5	0.109	0.359	PRG runtime grows faster but still manageable in low dimensions.
6	1.547	2.125	Overhead becomes more noticeable, which is expected to be outweighed in higher dimensions where brute-force is infeasible.

While the runtime increase for PRG may appear disadvantages in these low-dimensional lattices, heuristic approaches are typically designed for scalability to higher dimensions (Table 6).

Basis Conditioning

Preprocessing consistently reduced the condition number of all tested lattices, improving numerical stability. The most significant reductions were observed in higher-dimensional lattices, which initially exhibited poorer conditioning. This finding aligns with established theory in lattice reduction, where poorly conditioned bases can significantly affect the accuracy and efficiency of SVP solvers (Lenstra, et al., 1982) (Nguyen & Stehlé, 2010). By improving conditioning before applying the PRG algorithm, the likelihood of identifying vectors close to the shortest vector is enhanced, supporting the robustness of the heuristic even in difficult lattice configurations.

A closer examination of trials where PRG produced shorter vectors reveals that these successes often occurred in lattices with higher initial condition numbers. In poorly conditioned bases, the brute-force approach was constrained by its limited coefficient search range, preventing it from exploring vectors with larger coefficients that may yield shorter norms. The PRG heuristic was able to explore beyond these limitations, on occasion identifying shorter vectors. This pattern is consistent with prior observations in the literature, where randomised heuristics demonstrate an advantage in lattices with ill-conditioned bases, especially when exhaustive search is constrained (Micciancio, 2011) (Nguyen & Vidick, 2008). These results showcase a specific strength of the PRG approach: its robustness to variations in lattice conditioning, which may become increasingly relevant in higher dimensions where exhaustive search becomes infeasible.

PRG Success Rates

The PRG algorithm rarely produced shorter vectors than brute-force, achieving this outcome in only a small fraction of trials per dimension. This is expected given the constraints of the experiments: in low-dimensional lattices, brute-force enumeration explores the entire solution space, leaving limited opportunity for a heuristic to outperform it.

Table 7: Numerical Comparison of PRG Success rates

Dimension	Trials	PRG wins	Win Rate (%)	Observation
3	50	2	4%	PRG sometimes found shorter vectors in poorly conditioned bases.
4	50	1	2%	Rare success, consistent with expected behaviour of stochastic heuristics
5	50	4	8%	Higher chances of success as lattice size increases.
6	50	1	2%	Low win rate, reinforces that exhaustive search dominates in small dimensions.

These results suggest that the PRG algorithm demonstrates potential rather than superiority in low dimensions, in agreement with literature on randomised heuristics for SVP (Table 7) (Micciancio, 2011).

Interpretation and Comparison

Overall, the observed patterns confirm that the PRG algorithm behaves as a randomised heuristic should: as it produces competitive solutions within the expected norm range, incurs additional computation overhead in low-dimensional cases, and benefits from improved basis conditioning. Differences in observed PRG success rates comparted to other heuristic studies may arise from the specific lattice generation method, the small dimensions used, and the constrained search parameters applied to brute-force enumeration.

In summary, the PRG algorithm demonstrates promise as a heuristic for solving SVP, showing strengths in consistency, robustness to conditioning, and occasional identification of competitive short vectors. These outcomes highlight the potential of randomised approaches to complement traditional SVP solvers, especially in higher-dimensional lattices where exhaustive search is not feasible.

7 Limitations and Errors

This section lists and explains, the things that constrained measurements and could have biased results in this project. For each item, the cause, the mathematical effect, and effect on results are provided.

Experimental scale and generalisability

What: Tests were performed on small ranks ($n \leq 6$).

Mathematical effect: Lattice cryptography and provable hardness rely on behaviour in high dimensions, which can extend to hundreds or thousands. Complexity and geometry of lattices change qualitatively with dimension: enumeration and pruning strategies that work in snack dimensions may not scale or may behave differently in large dimensions. Claims about cryptographic security or runtime scaling must therefore be restricted to the tested range. This is a fundamental threat to validity.

How it affected results: Observed runtime improvements and solution quality for PRG and LLL may not persist at substantially higher dimensions. Metrics such as median runtime, success rates and variance are only valid for the tested small dimensions.

Simplified LLL implementation against full LLL.

What: Iteration 4 used a simplified, LLL-like size-reduction routine (rounding Gram-Schmidt projections and integer casting) rather than a full LLL implementation with Lovász checks, proper unimodular transformations and high-precision arithmetic. The original LLL algorithm and its guarantees are described in the literature.

Mathematical effect: A correct LLL implementation performs unimodular integer column operations (the lattice is preserved exactly in exact arithmetic) and enforced Lovász condition to swap vectors when required. The simplified routine can change the numerical basis in ways that may alter the lattice represent integer arithmetic errors occur, or produce suboptimal reductions compared to a formal LLL. Rounding *mu* and casting back to *int* can remove small fractional adjustments that a formal unimodular transform would represent exactly; this can increase or decrease the reported shortest vector norm unpredictably.

How it affected results: Reported condition score improvements and solver behaviour are meaningful as empirical observations, but they do not carry the formal approximation guarantees of LLL. This weakens any claim that observed improvements would hold under a certified reduction.

Bounded brute-force baseline (comparison fairness)

What: The brute-force solver is bounded to coefficient ranges $[-r, r]^n$. PRG sampling often uses a different coefficient range.

Mathematical effect: If the true shortest vector has coefficients outside the brute-force bound, the bounded brute-force baseline will not find it. PRG may appear to 'win' even though a larger range brute-force could find the same vector or a better vector.

How it affected results: Several observed 'wins' by PRG can be explained by brute-force being artificially limited. It does not prove PRG beats unbounded brute-force (which is infeasible). It only shows PRG explores different coefficient regions more efficiently given the parameter choices.

Numerical and rounding errors (floating-point arithmetic)

What: many operations use floating-point arithmetic (dot products, eigenvalue computation for condition score, Gram-Schmidt steps, PCA projections). Rounding, limited precision, and casting to integers can introduce numerical errors. Numerical linear algebra literature documents many pitfalls.

Mathematical effect: Eigenvalue computations can produce tiny negative eigenvalues due to rounding; taking the ratio without guarding against near-zero or negative values given unstable condition scores. In the simplified LLL reduction, rounding and converting to integers can produce non-unimodular transformations (numerical drift). PCA projection changes relative distances and may distort perceived vector lengths. PCA is only for visualisation, not a correctness test.

How it affected results: Condition scores may be noisy for nearly dependent bases. PCA visualisations could give misleading impressions about vector lengths and relative positions. In extreme cases small numerical errors could cause a reported zero-norm vector (anomalous results noted in earlier runs).

Randomness, sampling variance, and statistical uncertainty

What: PRG is randomised. Results (norm, runtime, found coefficients) vary with RNG seed and number of trials T. A single-run experiment is inadequate to characterise performance.

Mathematical effect: PRG performance metrics are random variables. Point estimates (single-run norms or runtimes) can be misleading; distributions can be skewed.

How it affected results: Variance across repeated PRG runs can be large; single-run 'success' may be due to chance. Reported efficiency is valid only in expectation or with confidence intervals.

Timing, benchmarking, and hardware dependencies

What: Wall-clock timings (used to compare runtimes) depend on the machine, background processes, BLAS/MKL acceleration and number of threads. Python/NumPy BLAS libraries can change observed runtimes dramatically.

Effect: Two runs of identical code can differ in runtime due to OS scheduling or multi-threaded BLAS. This is especially relevant for dense linear algebra used in basis operations and PCA.

Visualisation and interpretive errors (PCA)

What: PCA was used to create 3D visualisations for high-dimensional lattices.

Mathematical effect: PCA projects points onto principal components that preserver variance, not exact pairwise Euclidean distances. Two vectors that appear close in PCA space may be distant in original space, and vice versa.

How it affected results: Readers could incorrectly infer performance or vector length form the plot alone.

Implementation correctness and logging

What: Bugs and unintended behaviours can affect results: e.g., missing guard for *best_coeffs* is *None* in PRG local search (which was fixed), incorrect treatment of all-zero coefficient vector, or errors off by one in iteration bounds.

Effect: Crashes or incorrect outputs (e.g., zero norm, around coefficient reporting) were observed in early runs.

Choice of random basis distribution and problem representativeness

What: Bases were samples entry-wise from a uniform integer range (e.g., [-5,5]). This choice biases the class of lattices being tested.

Mathematical effect: Some lattice families (e.g., q-ary lattices or strutted cryptographic lattices) have very different geometry. Results on uniformly random small-entry bases do not necessarily generalise.

8 Threats to Validity

Several factors may limit the generalisability and interpretation of the experimental results presented in the project. These threats to validity are described below, along with explanations of how they were addressed or why they remain inherent limitations.

Scale Threat

The experiments were conducted on low dimensional lattices due to computational constraints. Lattices used in cryptographic applications are typically of much higher dimensions, so results observed here cannot be directly generalised to cryptographic settings. While the trends in vector norms, runtime, and conditioning provide insight into algorithm behaviour, conclusions about scalability or effectiveness in cryptographic lattices remain speculative.

Implementation Threat

The PRG algorithm was implemented with a simplified LLL-based preprocessing step to reduce basis conditioning. This implementation does not provide formal mathematical guarantees of optimality or correctness. Therefore, the results reported are empirical and reflect the performance of the implemented heuristic rather than a formally verified algorithm.

Baseline Threat

Comparisons were made against a brute-force solver with bounded coefficient ranges, These bounds were necessary to ensure computational feasibility but may bias the comparison in favour of the PRG algorithm in cases where the shortest vector lies outside the brute-force search range. This limitation should be considered when interpreting the rare instances where PRG outperformed brute-force.

Numeric Threat

All computations, including condition number calculations and vector reductions, were performed using floating-point arithmetic. Floating-point rounding and precision errors can affect the reported condition scores and reduction results, introducing variability that is not purely algorithmic. While care was taken to use consistent numerical methods, these effects remain a potential source of error.

Randomness Threat

The PRG algorithm relies on randomised sampling, and each trial represents a single run. Single-run results may not fully capture the variability inherent in the heuristic. To mitigate this, multiple trials were conducted per lattice dimension, and summary statistics such as mean, standard deviation, minimum and maximum values were reported to provide a more representative overview of algorithm behaviour.

Measurement Threat

Timing measurements were taken to compare algorithm runtime. Variations in hardware performance, background processes, and operating system scheduling may have influenced these measurements. While all experiments were conducted on the same machine under controlled conditions, reported runtimes should be interpreted as approximate relative measures rather than exact absolute values.

Selection Threat

Random lattices were generated for experimentation, but these may hit be fully representative of all possible lattice families. The choice of lattice bases may influence observed conditioning, vector norms, and algorithm performance. This selection bias limits the ability to generalise findings to lattices with different structural properties.

9 Areas of Improvement

While this project successfully implemented and compared multiple approaches to the SVP, there remain several areas where the work could be improved and extended in the future.

One crucial improvement would involve implementing a full version of the LLL basis reduction algorithm rather than the simplified version used in Iteration 4. The simplified approach was effective for demonstration, but a complete LLL implementation would provide stronger theoretical guarantees and closer alignment with established cryptographic practices (Lenstra, et al., 1982) (Nguyen & Vallée, 2010).

Another improvement would be to explore alternative reduction techniques such as BKZ (Block Korkine–Zolotarev) or deep insertion variants, which often achieve stronger reductions at the expense of higher computational cost (Chen & Nguyen, 2011). This would allow benchmarking solver performance under preprocessing conditions more representative of modern lattice-based cryptosystems.

Additionally, scaling to higher dimensions is also an important avenue for improvement. Even though this project evaluated solvers up to moderate dimensions, real cryptographic lattices often exceed hundreds of dimensions (Micciancio & Goldwasser, 2002). Adapting the algorithms to handle such scales, possibly by incorporating parallelisation or GPU acceleration, would make the project more practically relevant.

Furthermore, the heuristic solvers could be enhanced. The PRG with local search could be extended to include metaheuristic strategies such as simulated annealing or genetic algorithms, which may improve exploration of the search space and reduce the likelihood of premature convergence.

Finally, improvements could be made to statistical evaluation. Multiple runs with random seeds should be aggregated to produce distributions of runtime and solution quality, reducing the influence of randomness on results. Similarly, more precise timing measurements, including hardware profiling, would strengthen performance claims.

Overall, these improvements would increase both the theoretical depth and practical applicability of the project, helping to bridge the gap between small-scale demonstrations and cryptographic-scale lattice challenges.

10 Conclusions

This project explored the development and evaluation of a PRG algorithm for solving the SVP in low-dimensional lattices. The main objective was to design a heuristic approach capable of producing competitive short vectors while maintaining robustness to basis conditioning.

The experimental results indicate that:

The PRG algorithm consistently produces vectors within the same norm range as a brute-force solver, occasionally finding slightly shorter vectors in poorly conditioned bases. This demonstrates that the heuristic can explore solution spaces that may be restricted for bounded brute-force methods.

PRG requires longer computation times than brute-force in low dimensions, reflecting the additional operations performed during randomised sampling and local refinement. While this overhead is evident in low-dimensional lattices, the approach is expected to scale better relative to brute-force as dimensions increase, where exhaustive search becomes computationally infeasible (Hanrot, et al., 2011).

Preprocessing significantly improves lattice conditioning across all tested dimensions, supporting the robustness and reliability of the PRG algorithm in identifying near-optimal vectors (Lenstra, et al., 1982) (Nguyen & Stehlé, 2010).

While the algorithm rarely outperforms brute-force in low-dimensional lattices, successes occur predominantly in poorly conditioned bases. This highlights the heuristic's potential advantage in cases where exhaustive search is constrained.

The overall significance of this work lies in demonstrating that randomised heuristic approaches like PRG can serve as viable alternatives to exhaustive methods for SVP, particularly as lattice dimensions increase beyond the range feasible for brute-force search. Although experiments were limited to low-dimensional lattices, the project provides evidence that the PRG algorithm is robust, consistent, and capable of complementing traditional solvers. These findings are relevant for lattice-based cryptography, where efficient heuristic solvers are essential for exploring solution spaces that are otherwise computationally inaccessible (Micciancio & Goldwasser, 2002) (Micciancio, 2011).

In conclusion, the project successfully achieved its design goal of developing a PRG heuristic for SVP. The work contributes to the broader understanding of randomized algorithms for lattice problems and establishes a foundation for future research into higher-dimensional lattices and cryptographically relevant applications.

11 Acknowledgements

I would like to acknowledge the help I received from:

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

My teacher, Zita Murphy, for her feedback, mentorship, and direction throughout the course of this project.

Sheila Porter and the SciFest team for their encouragement and for creating opportunities to present and develop this project further. The support of SciFest CLG in fostering young scientific research in Ireland is deeply appreciated.

My school for their encouragement and constant support throughout my research, and their engagement throughout participation in scientific competitions. In particular, I would like to give thanks to my Deputy Principal, Emma Gleeson, for her incredible support and enthusiasm in all of my projects.

The wide range of content creators online, whose educational resources, tutorials, and discussions provided inspiration and technical insight during the development of the project.

Additionally, I would like to thank Dr. Hitesh Tewari from Trinity College Dublin and Dr. Mel Ó Cinnéide from UCD, for their guidance, expertise, and constructive feedback, which greatly strengthened the project’s depth and clarity.

The completion of this project would not have been possible without the collective support of these individuals and organisations.

12 References

(NIST)., N. I. o. S. a. T., 2016 . *Post-Quantum Cryptography: NIST’s Plan for the Future.* s.l.:s.n.

Ajtai, M., 1996. *Generating hard instances of lattice problems. In Proc. of the 28th Annual ACM Symposium on Theory of Computing.* s.l.:STOC 1996.

Ajtai, M. & Dwork, C., 1997. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* s.l.:s.n.

Ajtai, M., Kumar, R. & Sivakumar, D., 2001. *STOC..* s.l.:s.n.

Arora, S., Babai, L., Stern, J. & Sweedyk, Z., 1997. *Journal of Computer and System Sciences.* s.l.:s.n.

Babai, L., 1986. *Combinatorica..* s.l.:s.n.

Bernstein, D., Buchmann, J. & Dahmen, E., 2009 . *Post-Quantum Cryptography..* Berlin: Springer.

Campbell, P., Groves, M. & Shepherd, D., 2020 . *In Proceedings of Workshop on Post-Quantum Cryptography.* s.l.:PQCrypto.

Chen, Y. & Nguyen, P., 2011 . *In Advances in Cryptology – ASIACRYPT 2011.* s.l.:Springer.

Chen, Y. & Nguyen, P. Q., 2011. *Advances in Cryptology – ASIACRYPT 2011.* Heidelberg: Springer.

Fowler, M. a. J. H., 2001. "The agile manifesto.". *Software development 9,* Volume 8, pp. 28-35.

Gama, N. &. N. P., 2010. *Finding short lattice vectors within Mordell’s inequality..* s.l.:s.n.

Gama, N. & Nguyen, P., 2008. *In Advances in Cryptology – EUROCRYPT 2008.* Heidelberg: Springer.

Goldreich, O., Goldwasser, S. & Halevi, S., 1997. *Advances in Cryptology – CRYPTO’97.* Heidelberg: Springer.

Hanrot, G., Pujol, X. & Stehlé, D., 2011 . *Third International Workshop, IWCC 2011.* s.l.: Springer.

Hanrot, G., Pujol, X. & Stehlé, D., 2011. *CRYPTO..* s.l.:s.n.

Kannan, R., 1987 . *Mathematics of Operations Research.* s.l.:s.n.

Kannan, R., 1987. *Mathematics of Operations Research..* s.l.:s.n.

Lenstra, A. K., Lenstra, H. W. & Lovász, L., 1982. *Mathematische Annalen.* s.l.:s.n.

Lindner, R. & Peikert, C., 2011. *Better key sizes (and attacks) for LWE-based encryption..* s.l.:s.n.

Micciancio, D., 2011. *Heuristics for Lattice Problems: Randomized Algorithms and Sieve Methods..* s.l.:s.n.

Micciancio, D. & Goldwasser, S., 2002 . *Complexity of Lattice Problems: A Cryptographic Perspective..* s.l.:Springer..

Micciancio, D. & Regev, O., 2007. *SIAM J. Computing..* s.l.:s.n.

Micciancio, D. & Regev, O., 2009. *Lattice-based cryptography. In Theory and Applications of Cryptographic Techniques.* Heidelberg : Springer.

Micciancio, D. & Voulgaris, P., 2010. . *In Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC ’10).* s.l.:s.n.

Nguyen, P. Q. & Stehlé, D., 2010. *LLL on the Average..* s.l.:s.n.

Nguyen, P. Q. & Stern, J., 2001. *Cryptography and lattices. CALC 2001.* Heidelberg: Springer.

Nguyen, P. Q. & Vallée, B., 2010. *The LLL Algorithm: Survey and Applications..* s.l.:Springer..

Nguyen, P. Q. & Vidick, T., 2008. *J. Math. Cryptology..* s.l.:s.n.

Nguyen, P. Q. & Vidick, T., 2008. *Sieve Algorithms for the Shortest Vector Problem..* s.l.:s.n.

Peikert, C., 2016. *Lattice cryptography and its applications. In Theory of Cryptography Conference.* s.l.:Springer.

Regev, O., 2005. *Journal of the ACM.* s.l.:s.n.

Schnorr, C. & Euchner, M., 1994 . *Mathematical Programming.* s.l.:s.n.

Schnorr, C. P. & Euchner, M., 1994. *Mathematical Programming..* s.l.:s.n.

Shor, P., 1997 . *SIAM Journal on Computing.* s.l.:s.n.

Shor, P. W., 1994. *Algorithms for quantum computation.* s.l.:s.n.

12 Appendix

Below is the full final code of iteration 4. The full code from previous iterations can be viewed published on my GitHub repository through the link below or the QR code:

Link: https://github.com/addicarey/LatticeSVP

```python
#imports AC
import numpy as np
import time
from sklearn.decomposition import PCA
import plotly.graph_objs as go
from tabulate import tabulate

#LLL-like reduction AC
def lll_reduce(B):
    B = B.copy().astype(float)
    n = B.shape[1]
    for i in range(1, n):
        for j in range(i):
            mu = np.dot(B[:, i], B[:, j]) / np.dot(B[:, j], B[:, j])
            B[:, i] -= round(mu) * B[:, j]
    return B.astype(int)

#condition score AC
def basis_condition_score(B):
    B_T_B = B.T @ B
    eigvals = np.linalg.eigvals(B_T_B)
    return np.max(eigvals).real / np.min(eigvals).real

#naive brute-force solver AC
def naivebf_svp_solver(B, max_range=1):
    n = B.shape[1]
    shortest_vec = None
    min_norm = float('inf')
    best_coeffs = None
    for coeffs in np.ndindex(*((2 * max_range + 1,) * n)):
        coeffs = np.array(coeffs) - max_range
        if np.all(coeffs == 0):
            continue
        vec = B @ coeffs
        norm = np.linalg.norm(vec)
        if norm < min_norm:
            min_norm = norm
            shortest_vec = vec
            best_coeffs = coeffs
    return shortest_vec, best_coeffs, min_norm

#PRG AC
def PRG_svp_localsearch(B, max_trials=10000, initial_radius=10.0):
    n = B.shape[1]
    best_vec = None
    best_norm = float('inf')
    radius = initial_radius
    best_coeffs = None  # declare upfront

    for trial in range(max_trials):
        coeffs = np.random.randint(-3, 4, size=n)
        if np.all(coeffs == 0):
            continue
        vec = B @ coeffs
        norm = np.linalg.norm(vec)
        if norm < radius:
            if norm < best_norm:
                best_norm = norm
                best_vec = vec
                best_coeffs = coeffs
                radius = norm * 1.2

    #only proceed to local search if found a valid best_coeffs AC
    if best_coeffs is None:
        return None, None, float('inf')

    for delta in np.ndindex(*((3,) * n)):
        offset = np.array(delta) - 1
        neighbor = best_coeffs + offset
        if np.all(neighbor == 0):
            continue
        vec = B @ neighbor
        norm = np.linalg.norm(vec)
        if norm < best_norm:
            best_norm = norm
            best_vec = vec
            best_coeffs = neighbor

    return best_vec, best_coeffs, best_norm



#lattice Setup AC
n = 4 #dimensions
B = np.random.randint(-5, 6, size=(n, n))
print("Original Basis Matrix B:\n", B)
original_score = basis_condition_score(B)
print(f"Condition Score (original): {original_score:.2f}")
```

```
#apply LLL-like reduction prep AC
B_reduced = lll_reduce(B)
print("\nReduced Basis Matrix B:\n", B_reduced)
reduced_score = basis_condition_score(B_reduced)
print(f"Condition Score (reduced): {reduced_score:.2f}")

#brute-force solver AC
start = time.time()
shortest_naivebf, coeffs_naivebf, norm_naivebf = naivebf_svp_solver(B_reduced,
max_range=1)
time_naivebf = time.time() - start

#PRG AC
start = time.time()
shortest_prg, coeffs_prg, norm_prg = PRG_svp_localsearch(B_reduced)
time_prg = time.time() - start

#results Table AC
table = [
    ["Naive Brute-Force", str(coeffs_naivebf), f"{norm_naivebf:.3f}",
f"{time_naivebf:.4f} s"],
    ["PRG + Local Search", str(coeffs_prg), f"{norm_prg:.3f}",
f"{time_prg:.4f} s"]
]

print("\nComparison of Solvers (Iteration 4, LLL-Reduced Basis):")
print(tabulate(table, headers=["Method", "Coefficients", "Norm Length",
"Runtime"]))


#PCA visualisation AC
#generate lattice points AC
sample_vectors = []
max_range = 3  # slightly larger range for clarity AC

for coeffs in np.ndindex(*((2*max_range+1,) * n)):
    coeffs = np.array(coeffs) - max_range
    sample_vectors.append(B_reduced @ coeffs)

sample_vectors = np.array(sample_vectors)

#reduce to 3D with PCA AC
pca_3d = PCA(n_components=3)
data_3d = pca_3d.fit_transform(sample_vectors)

naive_proj = pca_3d.transform(shortest_naivebf.reshape(1, -1))
prg_proj = pca_3d.transform(shortest_prg.reshape(1, -1))
```

```
#plot AC
fig = go.Figure()

#lattice points AC
fig.add_trace(go.Scatter3d(
    x=data_3d[:, 0], y=data_3d[:, 1], z=data_3d[:, 2],
    mode='markers', marker=dict(size=2, color='blue'),
    name='Lattice Points'
))

#naive brute-force shortest vector AC
fig.add_trace(go.Scatter3d(
    x=[0, naive_proj[0, 0]],
    y=[0, naive_proj[0, 1]],
    z=[0, naive_proj[0, 2]],
    mode='lines+markers',
    marker=dict(size=5, color='red'),
    line=dict(width=4, color='red'),
    name='Naive Brute-Force SV'
))

#PRG shortest vector AC
fig.add_trace(go.Scatter3d(
    x=[0, prg_proj[0, 0]],
    y=[0, prg_proj[0, 1]],
    z=[0, prg_proj[0, 2]],
    mode='lines+markers',
    marker=dict(size=5, color='green'),
    line=dict(width=4, color='green'),
    name='PRG + Local Search SV'
))

fig.update_layout(title='Iteration 4: SVP Solutions with LLL-Reduced Basis',
                  scene=dict(xaxis_title='PCA 1',
                             yaxis_title='PCA 2',
                             zaxis_title='PCA 3'),
                  width=1000, height=1000)

fig.show()
```