

University of Oklahoma

GRADUATE COLLEGE

# Evolutionary Lattice Sieves

*A dissertation submitted to the graduate faculty  
in partial fulfilment of the requirements for the  
degree of  
Doctor of Philosophy*

By

أَحْمَدُ تَاشْفِينُ

AHMAD TASHFEEN

Norman, Oklahoma

2026

EVOLUTIONARY LATTICE SIEVES

A DISSERTATION APPROVED FOR THE  
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Qi Cheng—Chair

Dr. Dimitrios Diochnos

Dr. Dean Frederick Hougen

Dr. Ameya Pitale



Put up in a place  
where it's easy to see  
the cryptic admonishment

T. T. T.

When you feel how depressingly  
slowly you climb,  
it's well to remember that

Things Take Time

—*Hein* (1970)

# ACKNOWLEDGEMENTS

Custom holds that a dissertation has one author; custom, in this case, is mistaken. The evidence lies in the stalwart support of my committee members. Dr. Cheng (advisor), Dr. Diochnos and Dr. Hougen have never failed to make time for me, nor to be patient when I needed it. Dr. Pitale also graciously agreed to be the external committee member.

My thanks go to Dr. Radhakrishnan, who is the most powerful director the C.S. department has ever seen; Dr. Fili for showing me Cryptography and  $\text{\LaTeX}$ ; Dr. Wright for teaching me Linear and Abstract Algebra *without compromise*; Dr. Crick for carrying me throughout my undergrad; Prof. Horton for still allowing me in his lectures after all these years; Sir Sajjad for teaching me Physics; and Dr. Enchassi for being my Imam. I also want to thank the OU C.S. staff, Jaylin and Courtney; my OSU academic advisor, Cara Brun; and the student government advisor, Janelle Hanson.

To friends and family, thanks to Sam for talking me into a bachelor's in mathematics, Malka (Fareeha) for making me feel accomplished, Liza for being my archnemesis and Khadija for showing me how to solder, and for the quieter repairs no iron can fix.

I am grateful to my father, Ch. Mehmood, who still finds my ability to fix his phone the prime litmus test of my computer science education, and to my mother, Irsa Ghazal, who taught me right from wrong and is *my first teacher*.

Lastly, I remember my uncle (*mamū*) Khalid and grandma (*naano*) Anwar Bibi, who have passed away but always had faith in me, often more than I had in myself.

# ABSTRACT

Traditional cryptography, rooted in problems, e.g., integer factorisation or discrete log, is inevitably vulnerable to a fully operational quantum computer. Although it remains an engineering frontier, the looming threat extends to encrypted data stored today, which could be decrypted in the future with quantum capabilities. To safeguard against this eventuality, the backbone of modern quantum-safe cryptography is the *Shortest Vector Problem* (SVP). We enhance [Laarhoven](#)'s treatment of [Ajtai et al.](#)'s *sieving* as a genetic algorithm (GA) for the SVP by incorporating domain-informed SVP representation and crossover while extending application to the module lattices.

# CONTENTS

|                                                              |             |
|--------------------------------------------------------------|-------------|
| <b>Acknowledgements</b>                                      | <b>v</b>    |
| <b>Abstract</b>                                              | <b>vi</b>   |
| <b>List of Figures</b>                                       | <b>viii</b> |
| <b>List of Tables</b>                                        | <b>ix</b>   |
| <b>List of Algorithms</b>                                    | <b>x</b>    |
| <b>1 Lattices &amp; Quantum Cryptography</b>                 | <b>1</b>    |
| 1.1 Introduction . . . . .                                   | 1           |
| 1.2 Shortest Vector Problem Backed Cryptography . . . . .    | 2           |
| 1.3 Solving the Shortest Vector Problem . . . . .            | 3           |
| 1.4 Our Contributions . . . . .                              | 4           |
| 1.5 Outline . . . . .                                        | 5           |
| <b>2 Preliminaries</b>                                       | <b>6</b>    |
| 2.1 Lattice Notation . . . . .                               | 6           |
| 2.2 Integral Lattices . . . . .                              | 8           |
| 2.2.1 Good vs. Bad Lattice Bases . . . . .                   | 8           |
| 2.2.2 Hadamard Ratio . . . . .                               | 8           |
| 2.2.3 Lattice Problems . . . . .                             | 9           |
| 2.2.4 Gaussian Heuristic for the Shortest Vector . . . . .   | 10          |
| 2.3 Module Lattices . . . . .                                | 10          |
| 2.4 Gram–Schmidt Orthogonalisation Process . . . . .         | 10          |
| 2.5 Genetic Algorithms . . . . .                             | 14          |
| <b>3 Related Work</b>                                        | <b>16</b>   |
| 3.1 Lattice Reduction Algorithms . . . . .                   | 16          |
| 3.1.1 Gaussian Lattice Reduction Algorithm . . . . .         | 17          |
| 3.1.2 Lenstra, Lenstra, and Lovász (LLL) Algorithm . . . . . | 21          |
| 3.1.3 LLL Variations . . . . .                               | 23          |
| 3.2 Laarhoven’s Sieve . . . . .                              | 23          |

|          |                                                             |           |
|----------|-------------------------------------------------------------|-----------|
| 3.3      | The Knapsack Problem . . . . .                              | 24        |
| 3.3.1    | Collision Algorithm to Solve the Knapsack Problem . . . . . | 25        |
| 3.3.2    | Merkle–Hellman Knapsack Cryptosystem . . . . .              | 27        |
| 3.4      | Cryptanalysis via Lattices . . . . .                        | 29        |
| 3.4.1    | Knapsack Problem’s Reduction to the SVP . . . . .           | 30        |
| 3.4.2    | Key Sizes . . . . .                                         | 33        |
| <b>4</b> | <b>Methodology</b>                                          | <b>34</b> |
| 4.1      | A Naïve Sieve . . . . .                                     | 34        |
| 4.2      | Our Genetic Algorithm for Sieving . . . . .                 | 42        |
| 4.2.1    | Genotype . . . . .                                          | 42        |
| 4.2.2    | Crossover . . . . .                                         | 43        |
| 4.2.3    | Fitness Function . . . . .                                  | 44        |
| 4.2.4    | Mutation . . . . .                                          | 44        |
| 4.2.5    | Initial Population . . . . .                                | 45        |
| 4.2.6    | Selection Strategy . . . . .                                | 46        |
| 4.3      | Analysis of the Genetic Algorithms for Sieving . . . . .    | 47        |
| <b>5</b> | <b>Results</b>                                              | <b>48</b> |
| 5.1      | Choosing Data and Hyperparameters . . . . .                 | 48        |
| 5.2      | Reduction Results . . . . .                                 | 51        |
| 5.2.1    | Reducing SVP Challenge Lattices . . . . .                   | 51        |
| 5.2.2    | Reducing Random Integral Lattices . . . . .                 | 52        |
| 5.2.3    | Reducing Module Lattices . . . . .                          | 52        |
| 5.3      | Interpreting Results . . . . .                              | 52        |
| <b>6</b> | <b>Conclusion and Future Work</b>                           | <b>54</b> |
| 6.1      | Conclusion . . . . .                                        | 54        |
| 6.2      | Future Work . . . . .                                       | 55        |
|          | <b>Bibliography</b>                                         | <b>56</b> |
| <b>A</b> | <b>Figures</b>                                              | <b>59</b> |
| <b>B</b> | <b>Code Listings</b>                                        | <b>69</b> |
| <b>C</b> | <b>Console Outputs</b>                                      | <b>73</b> |

# LIST OF FIGURES

|     |                                                                                 |    |
|-----|---------------------------------------------------------------------------------|----|
| 2.1 | A Two-Dimensional Lattice . . . . .                                             | 7  |
| 2.2 | Vector Projection . . . . .                                                     | 11 |
| 3.1 | Gaussian Lattice Reduction Algorithm Example Steps One & Two . .                | 20 |
| 3.1 | Gaussian Lattice Reduction Algorithm Example Step Three . . . . .               | 21 |
| 4.1 | A Sample from the Lattice in Equation 2.1 & Figure 2.1 . . . . .                | 35 |
| 4.2 | Iterations 2 & 3 of Algorithm 4 Illustration . . . . .                          | 39 |
| 4.2 | Iterations 4 & 5 of Algorithm 4 Illustration . . . . .                          | 40 |
| 4.2 | Iterations 6 & 7 of Algorithm 4 Illustration . . . . .                          | 41 |
| 4.3 | Illustration of Our Crossover Technique . . . . .                               | 43 |
| 4.4 | Possible Offspring Mutations . . . . .                                          | 45 |
| 5.1 | Lattice Reduction Strength Comparison . . . . .                                 | 50 |
| A.1 | SVP Challenge Lattice ( $d = 40$ ) Reduction Curve . . . . .                    | 59 |
| A.1 | SVP Challenge Lattices' ( $d \in \{50, 60\}$ ) Reduction Curve . . . . .        | 60 |
| A.1 | SVP Challenge Lattices' ( $d \in \{70, 80\}$ ) Reduction Curve . . . . .        | 61 |
| A.1 | SVP Challenge Lattices' ( $d \in \{90, 100\}$ ) Reduction Curve . . . . .       | 62 |
| A.2 | SVP Random Integral Lattice ( $d = 40$ ) Reduction Curve . . . . .              | 63 |
| A.2 | SVP Random Integral Lattices' ( $d \in \{50, 60\}$ ) Reduction Curve . . . . .  | 64 |
| A.2 | SVP Random Integral Lattices' ( $d \in \{70, 80\}$ ) Reduction Curve . . . . .  | 65 |
| A.2 | SVP Random Integral Lattices' ( $d \in \{90, 100\}$ ) Reduction Curve . . . . . | 66 |
| A.3 | SVP Module Lattices' ( $d \in \{20, 30\}$ ) Reduction Curve . . . . .           | 67 |
| A.3 | SVP Module Lattices' ( $d \in \{40, 50\}$ ) Reduction Curve . . . . .           | 68 |

# LIST OF TABLES

|     |                                                              |    |
|-----|--------------------------------------------------------------|----|
| 3.1 | Euclid’s Greatest Common Divisor Algorithm Example . . . . . | 18 |
| 3.2 | Gaussian Lattice Reduction Algorithm Example . . . . .       | 19 |
| 3.3 | Knapsack Cryptosystem . . . . .                              | 28 |
| 3.4 | Encoding Table . . . . .                                     | 31 |
| 5.1 | Vector Length Comparison Table . . . . .                     | 49 |

# LIST OF ALGORITHMS

|   |                                                            |    |
|---|------------------------------------------------------------|----|
| 1 | Generic Genetic Algorithm . . . . .                        | 15 |
| 2 | Lenstra, Lenstra, and Lovász (LLL) Algorithm . . . . .     | 22 |
| 3 | Knapsack Problem Solver for Superincreasing Sets . . . . . | 27 |
| 4 | Naïve Lattice Sieving Algorithm . . . . .                  | 34 |
| 5 | Main Evolutionary Lattice Sieve . . . . .                  | 42 |
| 6 | Selection Algorithm . . . . .                              | 47 |

# LATTICES & QUANTUM CRYPTOGRAPHY

In this chapter, we introduce lattices and their various applications by highlighting the current relevance of the Shortest Vector Problem (SVP) in post-quantum cryptography and the [United States Department of Commerce](#)’s National Institute of Standards and Technology’s (NIST) reliance on it for modern cybersecurity standards. We also list our contributions towards a possible SVP solution via a nouvelle interdisciplinary (number theory and artificial intelligence) approach.

## 1.1 Introduction

Across disciplines as distant as civil engineering and cybersecurity, a single mathematical framework has unexpected prominence ([Lidl and Pilz, 1984](#); [Abramkina, Yakubova, Serikov, Begimbayeva, and Yakubov, 2024](#)). Lattices, once a number-theoretic abstraction, now stand at the heart of efforts to model, optimise, and secure modern complex systems. Among the vast applications of lattices, of more recent importance is *post-quantum cryptography*. This is mathematical (mostly asymmetric or public-key) cryptography post the creation or under the assumption of a fully functional quantum

computer.

Traditional encryption standards, for example, Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC), are vulnerable against quantum computers running Shor's algorithm. Asymmetric or public-key cryptography is as hard to break as the underlying maths problem, e.g., RSA relies on integer factorisation into prime numbers and ECC relies on calculation of discrete logarithm within finite fields and elliptic curves. Such problems remain hard on classical computers but can be easily solved with the assistance of a quantum computer. While a complete quantum computer may still be far in the future, current correspondences may still be stored for later decryption.

## 1.2 Shortest Vector Problem Backed Cryptography

On the 20<sup>th</sup> of December 2016, NIST launched an initiative to select and standardise quantum-resistant algorithms. Cryptographers found themselves seeking maths problems so formidable that even the whispers of a theoretical quantum machine could not unravel them. Since 2017, NIST has evaluated 69 of the 82 worldwide submissions, selecting the top 15. These were split into 7 (4 encryption, 3 signature) finalist and 8 (5 encryption, 3 signature) alternative categories, with draft standards published in 2023 ([Alagic et al., 2019, 2020, 2022, 2025](#)). Cybersecurity experts are now encouraged to incorporate these new standards to enhance protection against quantum threats.

Three of four final encryption algorithms chosen by NIST (CRYSTALS-KYBER, Saber and NTRU) are based on problems from lattice theory. CRYSTALS-KYBER and Saber use module lattices and NTRU is based on the SVP. Along with the knapsack<sup>1</sup> (subset-sum) problem's reduction to the SVP, the SVP has attracted the attention of

---

<sup>1</sup>A *rucksack* for the less north-atlantically inclined.

many prominent mathematicians over time, most notably, Carl Friedrich Gauss.

### 1.3 Solving the Shortest Vector Problem

Today, one of the most practical approaches to the SVP is sieving for the shortest vector as compared to the more traditional basis reduction algorithms (de Boer and van Woerden, 2025; Zhao et al., 2025). These include the Lenstra, Lenstra, and Lovász (LLL) and the block Korkin–Zolotarev (BKZ) algorithms (Section 3.1). These algorithms may be seen as the integral variants of the more general *Gram–Schmidt orthogonalisation process* described in Section 2.4.

More than half of the solutions for Darmstadt’s SVP challenge problems, posted on their website (Nguyen, 2025), are done via some sieving technique (Sun et al., 2020). Laarhoven’s (2019) treatment of sieving as a genetic algorithm enabled the application of the evolutionary framework to lattices. In this work, we enhance and further test this framework for lattice sieves by incorporating field-specific knowledge, while also extending it to the increasingly significant module lattices.

## 1.4 Our Contributions

The primary aim of this work is to focus on demonstrating the improvements achieved in a *simple genetic algorithm* only by integrating insights from lattice theory. Optimisations through the latest machine learning techniques remain an excellent opportunity for future research. This allows our results to be directly comparable to Laarhoven’s (2019) sieve and lets us attribute any improvements solely to the domain-informed modelling. Our contributions summarise as follows:

- 1) We define an improved SVP representation for genetic algorithms enabling a versatile crossover operator using the insights from traditional algorithms like LLL. Our crossover is compatible with module lattices, thereby extending a sieving algorithm to module lattices for the first time.
- 2) We solve all the challenge (Nguyen, 2025) and randomly generated problems up to 100 dimensions, improving application and scalability of Laarhoven’s (2019) approach.

## 1.5 Outline

The contents of each chapter are outlined as follows:

Chapter 1 introduces lattices and their various applications. It highlights the current relevance of the SVP in post-quantum cryptography and NIST’s reliance on it for modern cybersecurity standards. We also list our contributions towards a possible SVP solution via sieving and evolutionary computation in this chapter.

Chapter 2 presents key (lattice theory and other) concepts, problems, intuitions, heuristics, and notations utilised throughout this work. If the reader finds themselves in notationally unfamiliar territory or requires a short introduction to the fundamentals of lattice theory, this chapter explains both in sufficient detail.

Chapter 3 reviews related work, focusing on the traditional and modern approaches to the SVP and related problems, as they pertain to our work.

Chapter 4 lays out and analyses each of the building blocks of our primary algorithm. We start by building the reader’s intuition through a naïve version of the primary algorithm and a hand-traceable example is given. The example is followed by a detailed description of the primary genetic algorithm along with its pseudocode.

Chapter 5 sets up the experiment campaign and presents the observed outcomes. The chapter includes the raw data from the results and its interpretation.

Chapter 6 provides possible avenues of future work and finishes with the concluding remarks.

# PRELIMINARIES

In this chapter, we will introduce lattice mathematics: problems, intuitions, heuristics and notation used in this work.

A *lattice* is a discrete additive subgroup of a  $d$ -dimensional Euclidean space, i.e., a subset closed under addition where each element has no other element within a certain radius. Intuitively, lattice points are to Euclidean spaces what multiples of  $d$  are to the real number line. In two dimensions, a lattice may be colloquially referred to as a grid of points.

See Figure 2.1 for an example in two-dimensions.

## 2.1 Lattice Notation

Let  $\mathbb{N}, \mathbb{Z}, \mathbb{R}$  and  $\mathbb{C}$  be the sets of natural, integer, real and complex numbers respectively. Consider a basis matrix in a Euclidean space  $\mathcal{B} \in \mathbb{R}^{d \times d}$  with  $d$  linearly independent column vectors  $\{b_i \in \mathbb{R}^d : 1 \leq i \leq d\}$  such that  $b_i$  is the  $i^{\text{th}}$  column of  $\mathcal{B}$ . We recall a possibly transformed  $d$ -dimensional Euclidean space as  $\{\mathcal{B}x : x \in \mathbb{R}^d\}$ . If we restrict  $x \in \mathbb{Z}^d$ , i.e.,  $\mathcal{B}x$  to only the integral linear combinations, we obtain a discrete structure known as a lattice and notate it as  $\mathcal{L} = \mathcal{L}(\mathcal{B}) = \{\mathcal{B}x : x \in \mathbb{Z}^d\}$ .

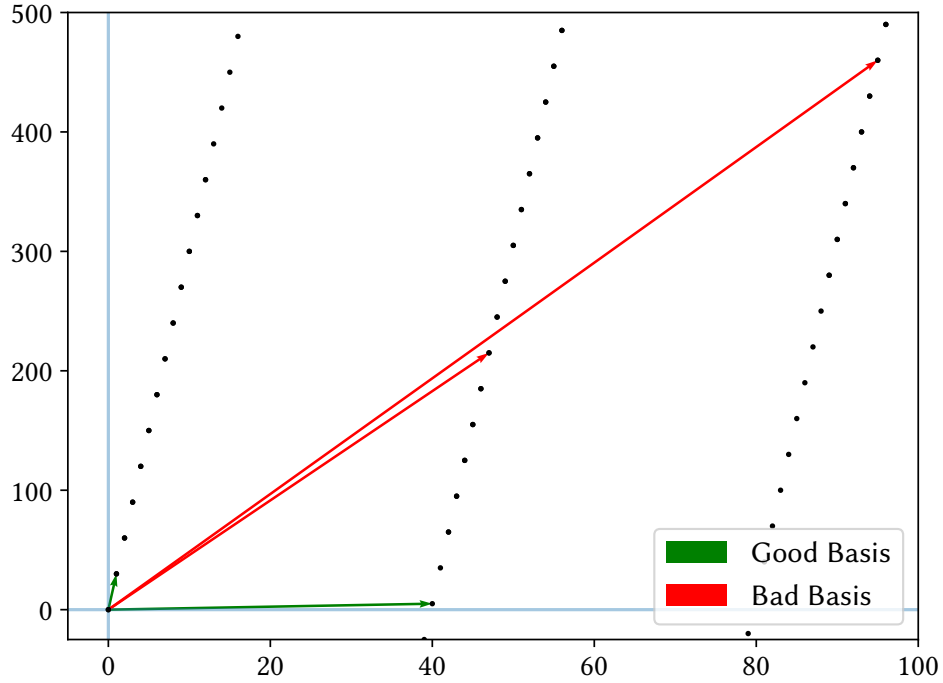
You may also see it written as,

$$\mathcal{L}(b_1, \dots, b_d) = \{x_1 b_1 + x_2 b_2 + \dots + x_d b_d : x_i \in \mathbb{Z}\}.$$

Or, to be a bit more notationally compact,

$$\mathcal{L}(\mathcal{B}) = \left\{ \sum_{i=1}^d x_i b_i : x_i \in \mathbb{Z} \right\}.$$

The *dimension* of a lattice is  $\dim(\mathcal{L}) = d$ , i.e., the number of column-vectors in the basis matrix  $\mathcal{B}$  and the determinant similarly  $\det \mathcal{L}(\mathcal{B}) = |\det \mathcal{B}|$ . If  $\mathcal{B}$  also has  $d$  rows, we call  $\mathcal{L}$  a *full rank lattice*. The set  $\mathbb{R}^d$  is now the *ambient space* for the reduced set of vectors that are in the lattice  $\mathcal{L}$ . The *group operation* of a lattice is the vector sum, i.e., if  $u \in \mathcal{L} \ni v$  then  $(u + v) \in \mathcal{L}$ .



**FIGURE 2.1**—An example of a two-dimensional lattice spanned by both a good basis (short and almost orthogonal) and a bad basis (long and narrow). See Section 2.2.1 or Equation 2.1.

## 2.2 Integral Lattices

If we further restrict  $\mathcal{B} \in \mathbb{Z}^{d \times d}$ , the resulting lattice is known as an *integral lattice*. You might imagine a trivial integral lattice  $\mathbb{Z}^2$ , spanned by the identity matrix. We show the plot of a two-dimensional lattice ( $d = 2$ ) in Figure 2.1 spanned by both of the bases (integral linear combinations of each other) stated in Equation 2.1.

$$\mathcal{B}_{\text{bad}} = \begin{bmatrix} 95 & 47 \\ 460 & 215 \end{bmatrix}, \quad \mathcal{B}_{\text{good}} = \begin{bmatrix} 1 & 40 \\ 30 & 5 \end{bmatrix} \quad (2.1)$$

### 2.2.1 Good vs. Bad Lattice Bases

Figure 2.1 shows two bases given in Equation 2.1 that are integral linear combinations of each other and span the same lattice<sup>1</sup>. One is labelled as *good* and the other as *bad*. A basis gets better as its vectors get shorter and more pairwise orthogonal. Hence, a good basis has short orthogonal basis vectors and vice versa. The process of turning a bad basis into a good basis is also sometimes referred to as performing *lattice reduction*. We care about reduced or good bases since certain lattice problems (Section 2.2.3) are easier with a good basis as compared to a bad basis. See Section 3.1 for how this reduction is performed.

### 2.2.2 Hadamard Ratio

The Hadamard ratio is used as a measure of orthogonality in-between the basis vectors  $b_i \in \mathcal{B}$ . Put simply, the closer this ratio is to 1 for a basis  $\mathcal{B}$ , the more it is orthogonal.

$$\mathcal{H}(\mathcal{B}) = \left( \frac{\det \mathcal{L}(\mathcal{B})}{\|b_1\| \|b_2\| \|b_3\| \cdots \|b_d\|} \right)^{\frac{1}{d}}$$

---

<sup>1</sup>“What’s in a name? That which we call a rose by any other name would smell as sweet.”—Shakespeare

### 2.2.3 Lattice Problems

The scale of theoretical complexity from the easiest to the hardest is arranged as,

$$\mathcal{NP}\text{-complete} \subseteq \mathcal{NP}\text{-hard} \subset \text{☠}.$$

Both the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP) are  $\mathcal{NP}$ -hard. However, SVP is  $\mathcal{NP}$ -hard under a randomised reduction, i.e., it is reduced to be  $\mathcal{NP}$ -hard via a probabilistic polynomial time algorithm instead of a deterministic polynomial time algorithm.

Let  $\|v\|$  be the  $\ell^2$  norm of a vector  $v$ .

**Definition 2.1** (Shortest Vector Problem (SVP)). Find the shortest non-zero vector  $v \in \mathcal{L}$  such that for all  $u \in \mathcal{L}$  where  $v \neq u$ , we have  $\|v\| \leq \|u\|$ .

**Definition 2.2** (Approximate Shortest Vector Problem (apprSVP)). Let  $\alpha \geq 1$  be an approximation factor and find a  $v \in \mathcal{L}$  such that  $\|v\|$  is no bigger than  $\alpha$  times the length of the shortest non-zero vector, i.e.,  $\|v\| \leq \alpha \|v_{\text{shortest}}\|$ .

**Definition 2.3** (Closest Vector Problem (CVP)). For  $w \in \mathbb{R}^d$  where  $w \notin \mathcal{L}$  find a vector  $v \in \mathcal{L}$  that is closest to  $w$  and  $\|w - v\| > 0$ , i.e.,  $\min_{v \in \mathcal{L}} \|w - v\|$ .

**Definition 2.4** (Shortest Basis Problem (SBP)). Find a  $b_i \in \mathcal{B}_{\text{good}}$  from/using  $\mathcal{B}_{\text{bad}}$  where  $b_i$  are short in some sense, e.g., minimise  $\max_{1 \leq i \leq d} \|b_i\|$  or  $\sum_{i=1}^d \|b_i\|^2$ .

*Remark 2.5.* SBP may be solved via various lattice reduction algorithms (Section 3.1). Any solution to the SBP often also uncovers a comparable solution to the apprSVP. Essentially all lattice-based cryptography (as talked about in the introduction) relies on the inability of lattice reduction algorithms to solve the apprSVP with an approximation of  $\alpha = \text{poly}(d)$  (Micciancio, 2025).

### 2.2.4 Gaussian Heuristic for the Shortest Vector

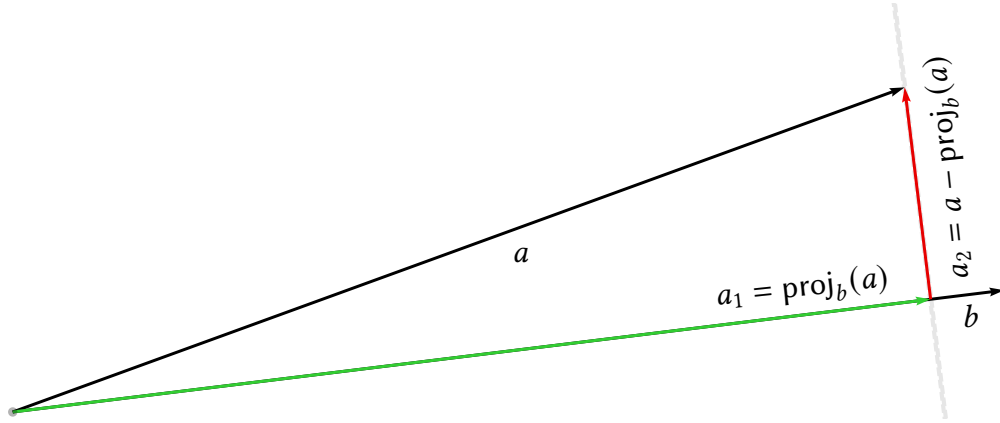
It is difficult to verify a solution for the apprSVP and SVP since the length of the shortest vector is unknown in the general case. As the dimension  $d$  of a lattice increases, we may rely on the Gaussian expected shortest length, also known as the Gaussian heuristic. For a small  $\varepsilon > 0$  and a sufficiently large dimension  $d$ , the shortest vector in a random lattice satisfies the relation,  $(1 - \varepsilon)\sigma(\mathcal{L}) \leq \|v_{\text{shortest}}\| \leq (1 + \varepsilon)\sigma(\mathcal{L})$  or rearranged as,  $\|v_{\text{shortest}}\| \approx \sigma(\mathcal{L}) = \sqrt{d/2\pi e}(\det \mathcal{L})^{\frac{1}{d}}$  (Hoffstein et al., 2008).

## 2.3 Module Lattices

In this work, we will also discuss the *module* lattices formed over the set of *Gaussian integers* notated as  $\mathbb{Z}[i] = \{\alpha + \beta i : (\alpha, \beta) \in \mathbb{Z}^2\}$  where  $i^2 = -1$ . Despite the name, Gaussian integers are complex numbers  $\alpha + \beta i$  where both the real  $\Re(\alpha + \beta i) = \alpha$  and the imaginary  $\Im(\alpha + \beta i) = \beta$  parts are limited to the integers. We can similarly form a lattice, but this time in the ambient space of complex numbers  $\mathbb{C}$ . This is done by letting  $\mathcal{B} \in \mathbb{Z}[i]^{d \times d}$  for the lattice  $\mathcal{L} = \{\mathcal{B}x : x \in \mathbb{Z}[i]^d\}$ . The norm of a  $v \in \mathbb{Z}[i]^d$  is given as,  $\|v\|_2^2 = v^H v = |v_1|^2 + |v_2|^2 + \dots + |v_d|^2$  where  $|v_j|^2 = \alpha^2 + \beta^2$  for  $v_j = \alpha + \beta i$ . The  $v^H$  is known as the *conjugate transpose* of  $v$ . For more on module lattices and their current usage, see de Boer and van Woerden (2025).

## 2.4 Gram–Schmidt Orthogonalisation Process

The following sets up the Gram–Schmidt orthogonalisation process in a manner tailored towards the subsequent notational requirements of Section 3.1. We start by illustrating a simple *vector projection* of  $a$  onto  $b$  notated as  $\text{proj}_b(a)$ . In Figure 2.2, we have two linearly independent vectors  $a$  and  $b$  from  $\mathbb{R}^2$  where  $a$  is projected onto  $b$ .



**FIGURE 2.2**—Projection of  $a$  onto  $b$ , i.e.,  $a_1 = \text{proj}_b(a) = \left( \frac{b \cdot a}{b \cdot b} \right) b$ .

Note that  $(b \cdot a)/(b \cdot b)$  is a scalar since it is a ratio of two dot (inner) products. Although well known, it may be helpful to restate that the dot product of two vectors  $a$  and  $b$  is defined as the sum of the products of the corresponding coordinates (or coefficients),

$$a \cdot b = \sum_{i=1}^d a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 + \cdots + a_d b_d.$$

While  $a$  and  $b$  form a basis of  $\mathbb{R}^2$  on the account of their linear independence, they do not form the desired orthogonal basis. On the other hand,  $a_2$  and  $b$  not only form the basis of  $\mathbb{R}^2$ , but they are also orthogonal. Therefore, we say that  $a_2$  and  $b$  form an orthogonal basis of  $\mathbb{R}^2$ . Since we already have  $b$ , observe that,

$$a_1 + a_2 = a$$

Head-to-Tail Rule.

$$a_2 = a - a_1$$

Subtracting  $a_1$  on both sides.

$$= a - \text{proj}_b(a)$$

By Figure 2.2.

$$= a - \left( \frac{b \cdot a}{b \cdot b} \right) b$$

By the definition of  $\text{proj}_b(a)$ .

$$= a - \mu b$$

Let  $\mu = \left( \frac{b \cdot a}{b \cdot b} \right)$

The Gram–Schmidt orthogonalisation process takes a linearly independent collection of vectors and repeatedly removes each vector’s components that lie in the directions already processed, leaving a new set of mutually orthogonal vectors.

Let  $b_1, b_2, b_3, \dots, b_d$  be a set of linearly independent vectors forming a basis of  $\mathbb{R}^d$ , then  $b_1^*, b_2^*, b_3^*, \dots, b_d^*$  is defined as a sequence of vector projections.

$$b_1^* = b_1$$

$$b_2^* = b_2 - \text{proj}_{b_1^*}(b_2)$$

$$b_3^* = b_3 - \text{proj}_{b_1^*}(b_3) - \text{proj}_{b_2^*}(b_3)$$

$\ddots$

$$b_k^* = b_k - \text{proj}_{b_1^*}(b_k) - \text{proj}_{b_2^*}(b_k) - \dots - \text{proj}_{b_j^*}(b_k) - \dots - \text{proj}_{b_{k-1}^*}(b_k)$$

$\ddots$

$$b_d^* = b_d - \text{proj}_{b_1^*}(b_d) - \text{proj}_{b_2^*}(b_d) - \text{proj}_{b_3^*}(b_d) - \dots - \text{proj}_{b_j^*}(b_d) - \dots - \text{proj}_{b_{d-1}^*}(b_d)$$

We explicitly write  $b_1^*, b_2^*, b_3^*, \dots, b_d^*$  to demonstrate the coefficients  $(b_j^* \cdot b_k)/(b_j^* \cdot b_j^*)$  that appear during the Gram–Schmidt process.

$$b_1^* = b_1$$

$$b_2^* = b_2 - \left( \frac{b_1^* \cdot b_2}{b_1^* \cdot b_1^*} \right) b_1^*$$

$$b_3^* = b_3 - \left( \frac{b_1^* \cdot b_3}{b_1^* \cdot b_1^*} \right) b_1^* - \left( \frac{b_2^* \cdot b_3}{b_2^* \cdot b_2^*} \right) b_2^*$$

$\ddots$

$$b_k^* = b_k - \left( \frac{b_1^* \cdot b_k}{b_1^* \cdot b_1^*} \right) b_1^* - \left( \frac{b_2^* \cdot b_k}{b_2^* \cdot b_2^*} \right) b_2^* - \dots - \left( \frac{b_j^* \cdot b_k}{b_j^* \cdot b_j^*} \right) b_j^* - \dots - \left( \frac{b_{k-1}^* \cdot b_k}{b_{k-1}^* \cdot b_{k-1}^*} \right) b_{k-1}^*$$

$\ddots$

$$b_d^* = b_d - \left( \frac{b_1^* \cdot b_d}{b_1^* \cdot b_1^*} \right) b_1^* - \left( \frac{b_2^* \cdot b_d}{b_2^* \cdot b_2^*} \right) b_2^* - \dots - \left( \frac{b_j^* \cdot b_d}{b_j^* \cdot b_j^*} \right) b_j^* - \dots - \left( \frac{b_{d-1}^* \cdot b_d}{b_{d-1}^* \cdot b_{d-1}^*} \right) b_{d-1}^*$$

Let  $\mu_{k,j} = \left( \frac{b_j^* \cdot b_k}{b_j^* \cdot b_j^*} \right)$  for  $1 \leq j < k \leq d$  then,

$$b_1^* = b_1$$

$$b_2^* = b_2 - \mu_{2,1}b_1^*$$

$$b_3^* = b_3 - \mu_{3,1}b_1^* - \mu_{3,2}b_2^*$$

$\ddots$

$$b_k^* = b_k - \mu_{k,1}b_1^* - \mu_{k,2}b_2^* - \cdots - \mu_{k,j}b_j^* - \cdots - \mu_{k,k-1}b_{k-1}^*$$

$\ddots$

$$b_d^* = b_d - \mu_{d,1}b_1^* - \mu_{d,2}b_2^* - \mu_{d,3}b_3^* - \cdots - \mu_{d,j}b_j^* - \cdots - \mu_{d,d-1}b_{d-1}^*.$$

From the setup of the  $b_1^*, b_2^*, b_3^*, \dots, b_d^*$  in this section, we will extract a  $d$  by  $d$  lower-triangular Gram–Schmidt coefficients matrix  $\mu$  for later utilisation in the lattice reduction algorithms, e.g., the LLL algorithm given in Section 3.1.2. Each entry  $\mu_{ij}$  records the projection coefficient of the basis vector  $b_i$  onto the orthogonalised vector  $b_j^*$ , and therefore captures precisely how far  $b_i$  leans in the direction of the earlier vectors. In this sense,  $\mu$  serves as a compact numerical summary of the quality of a basis: entries that are large in magnitude indicate basis vectors that are far from orthogonal. Lattice reduction algorithms exploit this summary directly, inspecting the entries of  $\mu$  to decide which basis vectors to adjust and by how much.

$$\mu = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ \mu_{2,1} & 0 & 0 & \cdots & 0 & 0 \\ \mu_{3,1} & \mu_{3,2} & 0 & \cdots & 0 & 0 \\ \mu_{4,1} & \mu_{4,2} & \mu_{4,3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mu_{d,1} & \mu_{d,2} & \mu_{d,3} & \cdots & \mu_{d,d-1} & 0 \end{bmatrix}$$

Code Listing B.2 in Appendix B gives a Python implementation of the vector projections and the Gram–Schmidt orthogonalisation as we defined it.

## 2.5 Genetic Algorithms

Genetic algorithms simulate the theory of biological evolution and natural selection to solve a range of mathematical-optimisation problems ([Mitchell, 2009](#)). Just like in naturally occurring evolution where individuals in a population evolve to gain desirable features (or, rather individuals with undesirable features go extinct), machine evolution starts with an initial population set  $P$  and filters individuals to be optimised after a certain number of generations of the same population.

To solve an optimisation problem with a genetic algorithm, we start by designing a *genotype* that encodes a possible solution. An instance of this genotype is referred to as an *individual*. A set of such individuals is a *population*. Populations of individuals evolve through *generations* by the means of reproduction. Using a *selection* process based on a *fitness-function*, two parent individuals are selected for *crossover* to produce a child individual in the next generation. The child individual may go through a *mutation*. We describe a general genetic algorithm for optimisation in [Algorithm 1](#).

[Algorithm 1](#) has a potential pitfall: it may lose the best individuals from one generation to another during crossover or mutation due to the probabilistic selection. This can cause divergence or slow convergence to a local optimum. *Elitism* is used to mitigate this and similar issues. Elitism ensures that the fittest  $|P|$  individuals are carried over to the next generation from the pool  $P \cup R$ . As a result, the next generation will consist of the best individuals out of the offspring and the parent set. Note that this is the same as global selection from [Laarhoven \(2019\)](#).

**Input:** Initial population:  $P$ , fitness-function:  $\text{Fitness} : P \rightarrow [0, 1]$ .

**Output:** The best individual in population, according to the fitness.

```
1: do
2:    $R \leftarrow \emptyset$ 
3:   for  $(u, v) \leftarrow \text{Select}(P, \text{Fitness})$ 
4:      $t \leftarrow \text{Cross}(u, v)$ 
5:     if (small random probability) then  $t \leftarrow \text{Mutate}(t)$ 
6:      $R \leftarrow R \cup \{t\}$ 
7:    $P \leftarrow R$ 
8: while  $(\forall v \in P, \text{Fitness}(v) < 1 - \varepsilon)$ 
```

**ALGORITHM 1**—Generic genetic algorithm from [Russell and Norvig \(2016, pg. 129\)](#).

The process starts with an initial population  $P$  of possible solutions encoded as individuals. Some individuals are then selected as a function of their fitness for reproduction.  $\text{Select}(P, \text{Fitness})$  selects reproduction candidates. The fitness function for any  $x \in P$  is  $0 \leq \text{Fitness}(x) \leq 1$  and gives a measure of how good is a solution. The  $\text{Cross}(u, v)$  operator takes two parent individuals and recombines them for a new possibly more fit offspring.  $\text{Mutate}(x)$  randomly mutates an individual. Most often this means adding a small amount of random noise to the individual  $x$ . The algorithm continues evolving generations until either we hit the maximum number of generations or an individual with fitness  $1 - \varepsilon$  is found.

## RELATED WORK

In this chapter, we review existing literature, focusing on the traditional and modern approaches to the SVP and related problems.

### 3.1 Lattice Reduction Algorithms

If a lattice is expressed in terms of its good basis then solving the SVP becomes fairly easy. This means that if we first solve the SBP then the SVP is easy. For example, assume for a certain  $\mathcal{B}_{\text{good}}$  that all column vectors  $b_i$  are pairwise orthogonal, i.e., for  $i \neq j$  we know that  $b_i \cdot b_j = 0$ . Therefore, for any  $x \in \mathbb{Z}^d$ ,

$$\|x_1 b_1 + x_2 b_2 + \cdots + x_d b_d\|^2 = x_1^2 \|b_1\|^2 + x_2^2 \|b_2\|^2 + \cdots + x_d^2 \|b_d\|^2.$$

And, the shortest non-zero vector(s) can be found in  $\{\pm b_1, \dots, \pm b_d\}$ . For an approximate solution of an instance of the CVP using a good basis, see Babai's nearest hyperplane algorithm ([Babai, 1986](#)) or [Hoffstein et al. \(2008, Theorem 7.34, pg. 405\)](#). In practice, however, we are seldom given such an orthogonal basis; the cryptographer publishes a long and skewed  $\mathcal{B}_{\text{bad}}$  precisely so that the geometry of the lattice remains obscured. Lattice reduction algorithms address this by transforming a given  $\mathcal{B}_{\text{bad}}$  basis into  $\mathcal{B}_{\text{good}}$ .

### 3.1.1 Gaussian Lattice Reduction Algorithm

In his famous book, *The Art of Computer Programming*, [Knuth \(2014\)](#) remarks on the Euclidean algorithm for the *Greatest Common Divisor* (GCD), also known as the *Highest Common Factor* (HCF).

We might call Euclid's method the granddaddy of all algorithms, because it is the oldest non-trivial algorithm that has survived to the present day.

The remarkability of Euclid's algorithm, given in Equation 3.1, cannot be overstated.

$$G(a, b) = \begin{cases} G(b, a) & \text{if } \|a\| > \|b\| \\ (a, b) & \text{if } f(a, b) = 0 \\ G(a, b - f(a, b)a) & \end{cases} \quad (3.1)$$

For two non-negative integers  $a$  and  $b$ ,  $G(a, b) = \gcd(a, b)$  where,

$$f(a, b) = \left\lfloor \frac{b}{a} \right\rfloor \quad \text{if } a > 0 \text{ else } 0.$$

Here the function  $f$  serves as the floored-quotient function. The expression  $b - f(a, b)a$  is effectively evaluating the remainder  $b \bmod a$ .

Euclid's GCD method is known as the *Gaussian Lattice Reduction Algorithm* ([Hoffstein et al., 2008](#), Section 7.13.1, pg. 436) if we treat  $a = b_1$  and  $b = b_2$  for the two-dimensional lattice basis  $\mathcal{B} = \{b_1, b_2\}$ . The function  $f$  in the lattice case is defined as  $f(b_1, b_2) = \lfloor \mu_{2,1} \rfloor$  from Section 2.4. Just as we know that,

$$\gcd(a, b) = G(a, b) = G(a, b - f(a, b)a)$$

for a two-dimensional lattice  $\mathcal{L}(\mathcal{B})$  we know that,

$$\text{SBP}(\mathcal{L}(\mathcal{B})) = G(b_1, b_2) = G(b_1, b_2 - f(b_1, b_2)b_1).$$

Example 1 for  $G(748, 2024)$  is summarised without the swapping steps in Table 3.1.

*Example 1* (Demonstrating the greatest common divisor between integers 748 and 2024 via  $G(748, 2024)$  from Equation 3.1).

|                                        |                                                                        |
|----------------------------------------|------------------------------------------------------------------------|
| $G(748, 2024) = G(748, 2024 - (2)748)$ | Since $f(2024, 748) = \left\lfloor \frac{2024}{748} \right\rfloor = 2$ |
| $G(748, 528) = G(528, 748)$            | Since $\ 748\  > \ 528\ $                                              |
| $G(528, 748) = G(528, 748 - (1)528)$   | Since $f(748, 528) = \left\lfloor \frac{748}{528} \right\rfloor = 1$   |
| $G(528, 220) = G(220, 528)$            | Since $\ 528\  > \ 220\ $                                              |
| $G(220, 528) = G(220, 528 - (2)220)$   | Since $f(528, 220) = \left\lfloor \frac{528}{220} \right\rfloor = 2$   |
| $G(220, 88) = G(88, 220)$              | Since $\ 220\  > \ 88\ $                                               |
| $G(88, 220) = G(88, 220 - (2)88)$      | Since $f(220, 88) = \left\lfloor \frac{220}{88} \right\rfloor = 2$     |
| $G(88, 44) = G(44, 88)$                | Since $\ 88\  > \ 44\ $                                                |
| $G(44, 88) = G(44, 88 - (2)44)$        | Since $f(88, 44) = \left\lfloor \frac{88}{44} \right\rfloor = 2$       |
| $G(44, 0) = G(0, 44)$                  | Since $\ 44\  > \ 0\ $                                                 |
| $G(0, 44) = (0, 44)$                   | Since $a = 0$                                                          |

**TABLE 3.1**—Euclid’s GCD algorithm example run for  $G(748, 2024)$  from Equation 3.1.

| Step | $a$ | $b$  | $f(a, b)$ |
|------|-----|------|-----------|
| 1    | 748 | 2024 | 2         |
| 2    | 528 | 748  | 1         |
| 3    | 220 | 528  | 2         |
| 4    | 88  | 220  | 2         |
| 5    | 44  | 88   | 2         |
| 6    | 0   | 44   | 0         |

A similar example of the Gaussian reduction is to reduce Equation 2.1  $\mathcal{B}_{\text{bad}}$  to  $\mathcal{B}_{\text{good}}$ .

$$\mathcal{B}_{\text{bad}} = \mathcal{B}_{\text{good}} \begin{bmatrix} 15 & 7 \\ 2 & 1 \end{bmatrix}$$

To be certain that  $\mathcal{B}_{\text{bad}}$  spans the same lattice as  $\mathcal{B}_{\text{good}}$  and not just a sub-lattice, we can verify the determinant of the coefficient matrix is 1.

$$\begin{vmatrix} 15 & 7 \\ 2 & 1 \end{vmatrix} = (15 \times 1) - (7 \times 2) = 15 - 14 = 1$$

*Example 2* (Demonstrating the Gaussian lattice reduction of  $\mathcal{B}_{\text{bad}}$  from Equation 2.1 using  $G(\mathcal{B}_{\text{bad}})$  from Equation 3.1).

$$G(\langle 47, 215 \rangle, \langle 95, 460 \rangle) = G(\langle 47, 215 \rangle, \langle 95, 460 \rangle - (2) \langle 47, 215 \rangle) \quad \text{See Figure 3.1a}$$

$$f(\langle 47, 215 \rangle, \langle 95, 460 \rangle) = \left\lfloor \frac{47 \cdot 95 + 215 \cdot 460}{47^2 + 215^2} \right\rfloor = 2$$

$$G(\langle 47, 215 \rangle, \langle 1, 30 \rangle) = G(\langle 1, 30 \rangle, \langle 47, 215 \rangle) \quad 47^2 + 215^2 > 30^2$$

$$G(\langle 1, 30 \rangle, \langle 47, 215 \rangle) = G(\langle 1, 30 \rangle, \langle 47, 215 \rangle - (7) \langle 1, 30 \rangle) \quad \text{See Figure 3.1b}$$

$$f(\langle 47, 215 \rangle, \langle 1, 30 \rangle) = \left\lfloor \frac{47 \cdot 1 + 215 \cdot 30}{1^2 + 30^2} \right\rfloor = 7$$

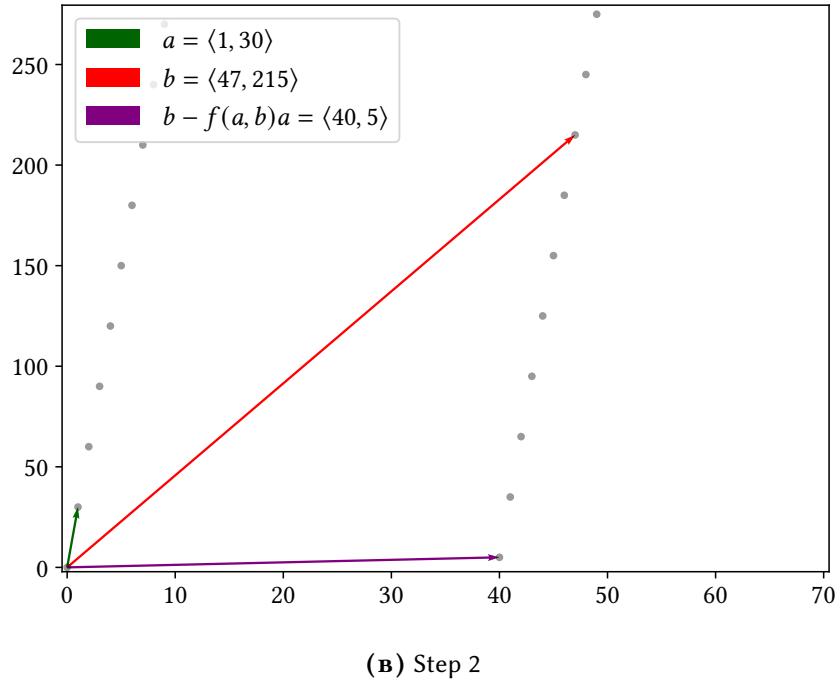
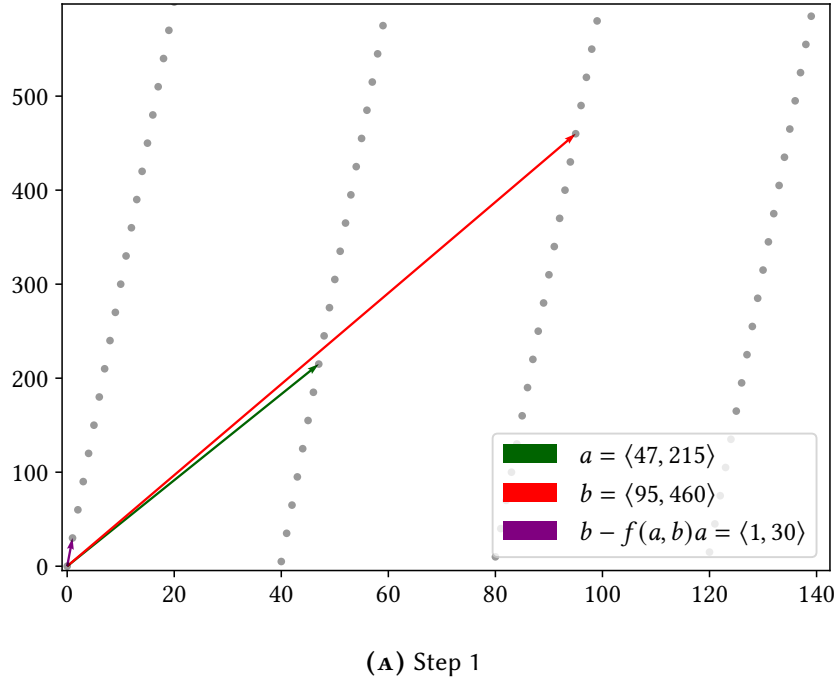
$$G(\langle 1, 30 \rangle, \langle 40, 5 \rangle) = (\langle 1, 30 \rangle, \langle 40, 5 \rangle)$$

$$f(\langle 1, 30 \rangle, \langle 40, 5 \rangle) = \left\lfloor \frac{1 \cdot 40 + 30 \cdot 5}{1^2 + 30^2} \right\rfloor = 0 \quad \text{See Figure 3.1c}$$

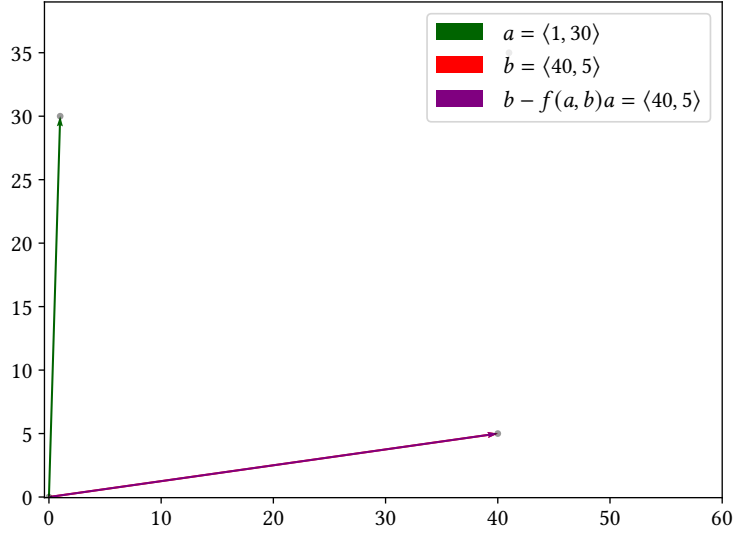
**TABLE 3.2**—Gaussian lattice reduction example run for Equations 3.1 and 2.1 with  $G(\mathcal{B}_{\text{bad}}) = G(\langle 47, 215 \rangle, \langle 95, 460 \rangle)$ .

| Step | $a$                       | $b$                       | $f(a, b)$ |
|------|---------------------------|---------------------------|-----------|
| 1    | $\langle 47, 215 \rangle$ | $\langle 95, 460 \rangle$ | 2         |
| 2    | $\langle 1, 30 \rangle$   | $\langle 47, 215 \rangle$ | 7         |
| 3    | $\langle 1, 30 \rangle$   | $\langle 40, 5 \rangle$   | 0         |

Table 3.2 summarises Example 2 for  $G(\mathcal{B}_{\text{bad}}) = G(\langle 47, 215 \rangle, \langle 95, 460 \rangle)$  behaving as the Gaussian lattice reduction. Furthermore, Figure 3.1 illustrates each of the steps. Code Listing B.1 gives a Python implementation for Eq. 3.1 and Examples 1 and 2.



**FIGURE 3.1**—Steps 1 and 2 of reducing  $\mathcal{B}_{\text{bad}}$  to  $\mathcal{B}_{\text{good}}$  from Equation 2.1 using Equation 3.1.



(c) Step 3

**FIGURE 3.1**—Steps 3 of reducing  $\mathcal{B}_{\text{bad}}$  to  $\mathcal{B}_{\text{good}}$  from Equation 2.1 using Equation 3.1.

### 3.1.2 Lenstra, Lenstra, and Lovász (LLL) Algorithm

The LLL algorithm generalises Gaussian lattice reduction from two to  $d$ -dimensions. Just like Gaussian lattice reduction, it subtracts an integral multiple of a shorter basis vector from a larger basis vector until some size condition is fulfilled. For each  $1 \leq j < k \leq d$ , we reduce  $b_k$  as  $b_k = b_k - \lfloor \mu_{k,j} \rfloor b_j$ . The vector  $b_j^*$  is the  $j^{\text{th}}$  basis vector in the Gram–Schmidt orthogonalisation of  $\mathcal{B}$ . The reduction of  $b_k$  is carried out using the Gram–Schmidt orthogonalisations of all the already reduced  $b_j$  for every index  $1 \leq j < k - 1$  where the *size condition*  $|\mu_{k,j}| > 0.5$  is met. Read Section 2.4 for more on the  $b_j^*$  and  $\mu_{k,j}$  appearing during the Gram–Schmidt orthogonalisation process.

If LLL terminates after only this recursive size reduction then the goodness of the reduced basis depends on the order of the original basis vectors in the basis matrix. Therefore, after the size reduction of  $b_k$ , the popular *Lovász condition* is checked.

$$\|b_k^*\|^2 \geq \left( \frac{3}{4} - \left( \frac{b_{k-1}^* \cdot b_k}{b_{k-1}^* \cdot b_{k-1}^*} \right)^2 \right) \|b_{k-1}^*\|^2.$$

For  $\delta = 3/4$  we check the condition,

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

If the Lovász condition is met then  $b_k$  is considered reduced and  $k$  will be incremented. Otherwise, for the optimal ordering, we swap  $b_k$  and  $b_{k-1}$  and decrement  $k$ <sup>1</sup>. LLL is a polynomial time lattice reduction algorithm for all  $0 < \delta < 1$ . The outer loop of Algorithm 2 runs at most in  $O(d^2 \log(d) + d^2 \log(\max\|b_i\|))$  and solves the apprSVP with an approximation factor of  $\alpha = 2^{(d-1)/2}$ . It is also an open problem whether LLL terminates in polynomial time for  $\delta = 1$ . For a more in-depth analysis, see [Kalbach and Chinburg \(2024\)](#).

**Input:** Lovász condition constant:  $0 < \delta < 1$ .

**Input:** Bad basis:  $\mathcal{B} = \{b_1, b_2, \dots, b_d\}$ .

**Output:** Good/Reduced basis:  $\mathcal{B} = \{b_1, b_2, \dots, b_d\}$ .

```

1:  $k \leftarrow 2$ 
2:  $(\mathcal{B}^*, \mu) \leftarrow \text{GramSchmidt}(\mathcal{B})$ 
3:   while  $k \leq d$ 
4:     for  $j \in \{k-1, k-2, k-3, \dots, 1\}$ 
5:       if  $|\mu_{k,j}| > 0.5$ 
6:          $b_k \leftarrow b_k - \lfloor \mu_{k,j} \rfloor b_j$ 
7:        $(\mathcal{B}^*, \mu) \leftarrow \text{GramSchmidt}(\mathcal{B})$ 
8:     if  $\|b_k^*\|^2 \geq \left(\delta - \mu_{k,k-1}^2\right) \|b_{k-1}^*\|^2$ 
9:        $k \leftarrow k + 1$ 
10:    else
11:       $\text{Swap}(b_{k-1}, b_k)$ 
12:       $(\mathcal{B}^*, \mu) \leftarrow \text{GramSchmidt}(\mathcal{B})$ 
13:       $k \leftarrow \max(k-1, 2)$ 
```

**ALGORITHM 2**—The Lenstra, Lenstra, and Lovász (LLL) algorithm. Pseudocode details are in Section 3.1.2.  $\text{GramSchmidt}(\mathcal{B})$  returns the Gram–Schmidt orthogonalisation of  $\mathcal{B}$  as well as the projection scalars  $\mu_{k,j}$  evaluated during the process of orthogonalising  $b_k$  via  $b_j$ .  $\text{Swap}(b_j, b_k)$  simply swaps the values of  $b_j$  and  $b_k$ . A Python implementation is given in Code Listing B.3.

---

<sup>1</sup>More precisely  $k = \max(k-1, 2)$

### 3.1.3 LLL Variations

While LLL is a polynomial time algorithm, many of its generalisations and extensions tend to perform just as fast during empirical analysis and yield a further reduced basis (Balny et al., 2025). Gama and Nguyen (2008) argue by their extensive empirical analysis that there is a gap between what theory is able to prove and what is the true power of the reduction algorithms.

The two possible exponential time variations of LLL are DEEP (deep insertion method) and BKZ (block Korkin–Zolotarev). DEEP differs from the standard LLL when the Lovász condition fails. Standard LLL simply swaps the  $b_k$  with  $b_{k-1}$  whereas in DEEP we insert  $b_k$  at an optimal place before the  $k^{\text{th}}$  basis vector. While in BKZ, instead of reducing  $b_k$  with only one  $b_j$ , the same is done with a block of basis vectors,  $b_j, b_{j+1}, b_{j+2}, \dots, b_{j+\beta-1}$  where  $\beta$  is the block size. Note that if we let  $\beta = d$  then the shortest vector in the output of BKZ solves the SVP and for any  $\beta < d$  we solve some version of the apprSVP.

## 3.2 Laarhoven’s Sieve

Sieving for the shortest vector first surfaced due to the works of Ajtai, Kumar, and Sivakumar (2001). Today, sieving algorithms have become very practical (Sun et al., 2020). At the time of this writing, more than half (437/847) of the solutions posted on the *Darmstadt Shortest Vector Problem Challenges* website (Nguyen, 2025) are done via sieving. The rest are solved using a combination of reduction and enumeration algorithms.

The idea behind sieving is quite simple: if we have two lattice vectors  $u, v \in \mathcal{L}$ , and we want a shorter one, we might try  $v - u$ . More specifically, generate a fixed subset  $P$  of  $\mathcal{L}$  and repeatedly check if there exists a pair of vectors  $u, v \in P$  such that

$\|v - u\| < \|v\|$  then  $v$  is replaced with  $v - u$ . When  $\|v - u\| < \|v\|$  is not true for any  $u, v \in P$ , we hope to solve some version of the apprSVP with the shortest vector in  $P$ .

Laarhoven (2019) set sieving up as a genetic algorithm. The foremost concern here arises about a possible method of mutation if lattice vectors  $v \in \mathcal{L}(\mathcal{B})$  are treated as individuals in a genetic algorithm. This is due to the fact that mutating a gene  $v_i \in v$  may cause  $v$  to step outside the lattice. For this reason, they encode the individual for  $v = \mathcal{B}x$  as  $x \in \mathbb{Z}^d$ . This is possible because of the definition of a lattice given in Chapter 2. Now  $x$  may be perturbed by adding binary noise to a random  $x_i \in x$ . While encoding a lattice vector  $v = \mathcal{B}x$  as  $x$  gives a way to mutate  $v$  through  $x$ , it requires a matrix multiplication  $\mathcal{B}x$  in order for the fitness (norm) of  $v$  to be calculated.

They compare their final genetic sieving algorithm with other state-of-the-art sieving algorithms to show the improvements accrued by optimising sieving through evolutionary techniques such as global selection of the fittest vectors. Global selection in the context of genetic sieving is done by picking the next generation of vectors per their norm over both the set of parents and offspring. Note that Laarhoven's algorithm's final variant solves the same (and only attempted) 40-dimensional lattice as we will in the results Chapter 5. They reported 29 iterations taking 1.9 seconds starting with a set of 1500 lattice vectors to find the shortest vector of norm 1702 (Laarhoven, 2019).

### 3.3 The Knapsack Problem

If any of the  $\mathcal{NP}$ -complete problems are shown to be in  $\mathcal{P}$  then all  $\mathcal{NP}$ -problems are in  $\mathcal{P}$ . In the same seminal paper (Karp, 1972) where Karp determines a subset of the  $\mathcal{NP}$  problems as  $\mathcal{NP}$ -complete with the aforementioned property, he also gives twenty-one examples of such  $\mathcal{NP}$ -complete problems. Number 18 on the list is the

knapsack or the subset-sum problem.

If your house was on fire, and you could only carry out a kilogram of weight with you, you might want to add the most valuable items up to a kilogram. How could that be done? You can take the power set of your household items, then sieve it to only contain the sets that weigh a kilogram and find the one with the maximum value. But of course, your house is on fire, so hurry up!

More realistically, we solve an instance of the knapsack problem every time we make change. The most common coin denominations in the United States are the twenty-five cents, ten cents, five cents and one cent, named quarters, dimes, nickels and pence.

$$M_{\text{easy}} = \{1, 5, 10, 25\}$$

Let's say we had the above set and wanted to make 31 cents. You may come out with the subset  $\{1, 5, 25\}$ . We can represent this as a vector dot product.

$$M_{\text{easy}} = \langle 1, 5, 10, 25 \rangle, \quad x = \langle 1, 1, 0, 1 \rangle \quad \text{and} \quad M_{\text{easy}} \cdot x = 31$$

**Definition 3.1** (Knapsack Problem). Given a set  $M = \{m_1, m_2, m_3, \dots, m_n\} \in \mathbb{N}^n$  of positive integers and a positive integer sum  $S \in \mathbb{N}$ , find an  $x \in \{0, 1\}^n$  such that the dot product  $Mx = S$ . In other words, find a subset of  $M$ , whose elements sum to  $S$ .

### 3.3.1 Collision Algorithm to Solve the Knapsack Problem

We prove a simple collision-algorithm for a general knapsack problem that runs in  $O(2^{\frac{n}{2}})$  steps, here  $n$  is the cardinality of  $M$ . The idea is to split  $M$  in two halves and search each half against the other till a collision that adds up to  $S$  is found. A hand traceable example execution of the algorithm is given in Example 3 where  $M = \{2, 3, 5, 7, 11, 13\}$  and  $S = 26$ .

*Proof.* Any general knapsack problem  $(M, S)$  can be solved in  $O(2^{\frac{n}{2}})$ .

We start by splitting  $M$  into two halves,

$$\begin{aligned} M_L &= \left\{ 0 \leq i < \left\lfloor \frac{n}{2} \right\rfloor : M_i \right\} && \text{The left half} \\ M_R &= \left\{ \left\lfloor \frac{n}{2} \right\rfloor \leq i < n : M_i \right\} && \text{The right half} \end{aligned}$$

Note that if  $n$  is odd,  $M_L$  has fewer elements. Now we state all the possible half-subsets  $\chi_i$  and their corresponding sums. Let  $b_j(i) = \left\lfloor \frac{i}{2^j} \right\rfloor \bmod 2$  be the  $j^{\text{th}}$  bit in the binary representation of  $i$  from the little end. For example,  $b_0(2) = 0$  and  $b_1(2) = 1$ . We will also define  $\chi_i = \{0 \leq j \leq \lfloor \lg i \rfloor : b_j(i)\}$  as the binary representation of  $i$  and the corresponding sum as,

$$\begin{aligned} L &= \left\{ 0 \leq i < 2^{\lfloor \frac{n}{2} \rfloor} : \sum_{j=0}^{\lfloor \lg i \rfloor} \chi_{i,j} M_{L_j} \right\}, \\ R &= \left\{ 0 \leq i < 2^{\lceil \frac{n}{2} \rceil} : \sum_{j=0}^{\lfloor \lg i \rfloor} \chi_{i,j} M_{R_j} \right\}. \end{aligned}$$

Since  $M_L \in \mathbb{N}^{\lfloor \frac{n}{2} \rfloor}$  and  $M_R \in \mathbb{N}^{\lceil \frac{n}{2} \rceil}$  are both sets of positive integers, we may sort both  $L$  and  $R$  in  $O(2^{n/2})$ . Once they are both sorted, we may find a collision between  $L$  and  $R$  like this:  $L \star R = L_\ell + R_{r'} = S$ ,

$$\ell \leftarrow 2^{\lfloor \frac{n}{2} \rfloor} - 1$$

$$r' \leftarrow 0$$

**while**  $(L_\ell + R_{r'} \neq S)$  # while no collision

**if**  $(L_\ell + R_{r'} < S)$  **then**  $r' \leftarrow r' + 1$

**if**  $(L_\ell + R_{r'} > S)$  **then**  $\ell \leftarrow \ell - 1$

The above loop finds a collision in at most  $2^{n/2}$  iterations since  $\ell$  and  $r'$  will at most meet in the middle. After, we may concatenate  $\chi_\ell$  and  $\chi_{r'}$  for  $x$  in  $Mx = S$ .

QUOD  
ERAT  
DEMONSTRATUM ■

*Example 3* (Solving  $M = \{2, 3, 5, 7, 11, 13\}$  and  $S = 26$  using the collision algorithm shown in Section 3.3.1).

$$\begin{array}{c} \chi \\ \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} M_L \\ \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \end{array} = \begin{array}{c} L \\ \begin{bmatrix} 0 \\ 2 \\ 3 \\ 5 \\ 5 \\ 7 \\ 8 \\ 10 \end{bmatrix} \end{array} \star \begin{array}{c} R \\ \begin{bmatrix} 0 & \leftarrow \cancel{2} \\ 7 & \leftarrow \cancel{3} \\ 11 & \leftarrow \cancel{5} \\ 13 & \leftarrow \cancel{7} \\ 18 & \leftarrow \cancel{11} \\ 20 \\ 24 \\ 31 \end{bmatrix} \end{array} = \begin{array}{c} \chi \\ \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} M_R \\ \begin{bmatrix} 7 \\ 11 \\ 13 \end{bmatrix} \end{array} \quad (3.2)$$

The sum  $L_\ell + R_\ell$  goes through 10, 17, 21, 23, 28, 26 and we recover  $x = \langle 0, 1, 1, 1, 1, 0 \rangle$  in less than  $2^{6/2} = 8$  steps.

**Input:** Superincreasing set  $M_{\text{easy}}$  and integer sum  $S$ .

**Output:** Solution to the knapsack problem  $x$ .

- 1: **for**  $i \in \{|M_{\text{easy}}|, |M_{\text{easy}}| - 1, |M_{\text{easy}}| - 2, \dots, 1\}$
- 2:    $x_i \leftarrow 0$
- 3:   **if**  $S \geq (M_{\text{easy}})_i$
- 4:      $x_i \leftarrow 1$
- 5:      $S \leftarrow S - (M_{\text{easy}})_i$

**ALGORITHM 3**—Linear time algorithm to solve the knapsack problem for superincreasing sets  $M_{\text{easy}}$ .

### 3.3.2 Merkle–Hellman Knapsack Cryptosystem

Revisiting the aforementioned coin denominations set  $M_{\text{easy}} = \{1, 5, 10, 25\}$ , note that  $m'_{i+1} \geq 2m'_i$ . That is not just a coincidence. The sets/sequences of the form  $M_{\text{easy}} = \{1 \leq i \leq n : m'_i\}$  where  $m'_{i+1} \geq 2m'_i$  are known as *superincreasing*. Knapsack problems for such sets can be solved in linear time by Algorithm 3. Furthermore, there is a bijection between the subsets of superincreasing sequences and their sums.

The first cryptosystem to be based on an  $\mathcal{NP}$ -complete problem uses a disguised

knapsack problem and was attempted by [Merkle and Hellman \(2019\)](#). We say a *disguised* knapsack problem since whether a cryptographic system can be as hard to break as to solve an  $\mathcal{NP}$ -complete problem is an open problem in itself ([Pass, 2006](#)). One might disguise an easy knapsack problem, defined with a *superincreasing*  $m'_i \in M_{\text{easy}}$  such that  $m'_{i+1} \geq 2m'_i$  by modding integral multiples of its elements. Note that Algorithm 3 does not work on  $M_{\text{disguised}} = \{Am'_i \bmod B : \gcd(A, B) = 1, m'_i \in M_{\text{easy}}\}$ .

The cryptosystem of [Merkle and Hellman \(2019\)](#) is given in Table 3.3. Even if the general knapsack problem is  $\mathcal{NP}$ -complete, the disguised version  $(M_{\text{disguised}}, S)$  is not since we know that there exists information  $A, B$  which makes it easy. We call such problems—which appear  $\mathcal{NP}$ -hard only in the absence of trapdoor information:  $\mathcal{NP}$ -imposter problems.

**TABLE 3.3**—[Merkle and Hellman \(2019\)](#) knapsack cryptosystem. Alice can disguise a superincreasing set  $M_{\text{easy}}$  as  $M_{\text{disguised}}$  then Bob can send her messages  $x$  (binary vectors) by sending  $S = M_{\text{disguised}} \cdot x$  over a public channel. In such a way, Eve (an eavesdropper) knows  $(M_{\text{disguised}}, S)$  and has to solve an  $\mathcal{NP}$ -imposter problem to obtain  $x$  whereas since Alice knows  $(M_{\text{easy}}, S')$ , she can use the Algorithm 3 to quickly obtain Bob’s original message.

| Alice                                                                                                                                                                           | Eve                                                         | Bob                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|-------------------------------------|
| Pick $M_{\text{easy}} = \langle m'_1, m'_2, \dots, m'_n \rangle$ ,<br>such that $m'_1 > 2^n$ and $m'_{i+1} \geq 2m'_i$ .<br>Pick $A, B$ with $B > 2m'_n$ and $\gcd(A, B) = 1$ . |                                                             | Wants to send Alice a message $x$ . |
| Send $M_{\text{disguised}} = \{Am'_i \bmod B : m'_i \in M_{\text{easy}}\}$                                                                                                      | $M_{\text{disguised}}$                                      | $M_{\text{disguised}}$              |
| Receive $S$                                                                                                                                                                     | $S$                                                         |                                     |
| Let $S' \equiv A^{-1}S \bmod B$ .<br>Solve $(M_{\text{easy}}, S') \rightarrow x$ .<br>We have $M_{\text{easy}}x = S'$ .                                                         | $(M_{\text{disguised}}, S)$<br>is $\mathcal{NP}$ -imposter. | $S = M_{\text{disguised}}x$         |

Towards the end of Table 3.3 where Alice receives Bob’s  $S$  and computes  $S' \equiv A^{-1}S \bmod B$ , we must also be certain that  $M_{\text{easy}}x = S'$  if and only if  $S = M_{\text{disguised}}x$ . In the

derivation below, we refer to  $M_{\text{disguised}}$  simply as  $M$ .

$$\begin{aligned}
S' &\equiv A^{-1}S \pmod{B} \\
&\equiv A^{-1}Mx \pmod{B} && \text{Bob's Encryption } S = Mx \\
&\equiv \sum_{i=1}^n A^{-1}m_i x_i \pmod{B} \\
&\equiv \sum_{i=1}^n A^{-1}(Am'_i)x_i \pmod{B} && \text{Since } m_i \equiv Am'_i \\
&\equiv \sum_{i=1}^n m'_i x_i \pmod{B} \\
&= M_{\text{easy}}x && \text{Since } M_{\text{easy}}x \leq m'_1 + m'_2 + m'_3 + \cdots + m'_n < 2m'_n < B
\end{aligned}$$

### 3.4 Cryptanalysis via Lattices

Generic cryptanalysis constitutes a systematic investigation of techniques for uncovering the structure or secret keys of cryptographic algorithms without detailed knowledge of their implementation. It draws upon a wide spectrum of methods—including statistical inference, differential analysis, and algebraic attacks—to expose vulnerabilities in symmetric ciphers, public-key systems, and hash functions. By modelling adversarial capabilities, the analysis identifies exploitable weaknesses in the mathematical foundations of these primitives. Both theoretical proofs and empirical testing are combined to assess the computational effort and probability of success associated with each attack. Abstracting away protocol-specific details yields a universal framework for comparing the resilience of diverse cryptographic schemes. The resulting insights inform the design of robust algorithms and the establishment of rigorous security standards.

In this section, we reduce the knapsack problem to the SVP and introduce the consequent cryptanalysis that results in successful attacks on cryptosystems such as the one proposed by [Merkle and Hellman \(2019\)](#).

### 3.4.1 Knapsack Problem's Reduction to the SVP

Lagarias and Odlyzko (1985) showed that any knapsack problem can be encoded as an SVP. Take any knapsack problem  $(M, S)$  and the relevant solution  $x$  such that  $Mx = S$ . Now consider the lattice basis  $\mathcal{B}_{\text{bad}}$  of dimension  $d = n + 1$  in Equation 3.3. The lattice spanned by  $\mathcal{B}_{\text{bad}}$  must have a vector  $t \in \mathcal{L}(\mathcal{B}_{\text{bad}})$  that is the result of an integral linear combination due to  $x$ .

$$t = \underbrace{\begin{bmatrix} 2 & 0 & \cdots & 0 & 1 \\ 0 & 2 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 2 & 1 \\ m_1 & m_2 & \cdots & m_n & S \end{bmatrix}}_{\mathcal{B}_{\text{bad}}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ -1 \end{bmatrix} = \begin{bmatrix} 2x_1 - 1 \\ 2x_2 - 1 \\ \vdots \\ 2x_n - 1 \\ M \cdot x - S \end{bmatrix} = \begin{bmatrix} 2x_1 - 1 \\ 2x_2 - 1 \\ \vdots \\ 2x_n - 1 \\ 0 \end{bmatrix} \quad (3.3)$$

Since  $x$  is a binary vector, any  $(2x_i - 1) \in t$  must be either  $2(1) - 1 = 1$  or  $2(0) - 1 = -1$ . Therefore,  $\|t\| = \sqrt{n}$  and  $t$  is very likely to be the shortest vector in the lattice  $\mathcal{L}(\mathcal{B}_{\text{bad}})$  because for all other lattice vectors  $v \in \mathcal{L}(\mathcal{B}_{\text{bad}})$  the length  $\|v\| \gg \sqrt{n}$  due to the non-unit squares. This is especially true if the original  $m_i \in M$  were chosen to be on the order of  $\Theta(2^{2n})$  due to requirements specified in Table 3.3.

$$\Theta(2^{2n}) \ni 2^{2n} = \underbrace{(2 \cdot 2^{n-1} \cdot 2^n)}_B > \underbrace{(2^{n-1} \cdot 2^n)}_{m'_n} \geq \cdots \geq \underbrace{(2 \cdot 2 \cdot 2^n)}_{m'_3} \geq \underbrace{(2 \cdot 2^n)}_{m'_2} \geq m'_1 > 2^n$$

The short vector  $t$  in the lattice spanned by the basis  $\mathcal{B}_{\text{bad}}$  will reveal the solution  $x$  to the knapsack problem  $(M, S)$ , i.e.,  $x_k = 1$  if  $t_k > 0$  else  $x_k = 0$ . Since,  $\mathcal{L}(\mathcal{B}_{\text{bad}}) = \mathcal{L}(\mathcal{B}_{\text{good}})$  as discussed in Section 2.2.1, if we can find the corresponding  $\mathcal{B}_{\text{good}}$  by solving the SBP (Sections 2.2.3 & 3.1) then we may easily solve the knapsack problem encoded therein.

*Example 4* (Lattice-based attack on Merkle and Hellman (2019) cryptosystem). One of Tashfeen's friends, Sam, uses the Merkle and Hellman (2019) cryptosystem, with the following public key,

$$M = \langle m_1, m_2, m_3, \dots, m_{25} \rangle$$

$$= \langle 67108861, 134217725, 268435453, 536870909, 1073741821, 2147483645, \\ 4294967293, 8589934589, 17179869181, 34359738365, 68719476733, 137438953469, \\ 274877906941, 549755813885, 1099511627773, 2199023255549, 4398046511101, \\ 8796093022205, 17592186044413, 35184372088829, 70368744177661, \\ 140737488355325, 281474976710653, 562949953421309, 1125899906842621 \rangle.$$

And, the encoding given in Table 3.4. Tashfeen sends Sam the name of his favourite

**TABLE 3.4**—Encoding table.

| $\perp$ | A     | B     | C     | $\dots$ | X     | Y     | Z     |
|---------|-------|-------|-------|---------|-------|-------|-------|
| 0       | 1     | 2     | 3     | $\dots$ | 24    | 25    | 26    |
| 00000   | 00001 | 00010 | 00011 | $\dots$ | 11000 | 11001 | 11010 |

Advisory Conference Report (ACR) committee member via  $S = 2002491457667039$ .

Tashfeen’s advisor decides to figure out the plaintext  $x$ . He first constructs the bad

basis  $\mathcal{B}_{\text{bad}}$ .

[illegible]

Then, he uses a lattice reduction algorithm to obtain the reduced  $\mathcal{B}_{\text{good}}$ .

|    |    |    |    |    |    |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |         |
|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|---------|
| -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | -1 | 0  | 0  | 0  | 0  | 1174258 |
| 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | 3  | -1 | 0  | 0  | 0  | 0  | 1174260 |
| 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | 1  | -1 | 0  | 0  | 0  | 0  | 1174260 |
| 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | -1 | 1  | 0  | 0  | 0  | 0  | 1174258 |
| 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | 1  | -3 | 0  | 0  | 0  | 0  | 1174262 |
| 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 1  | 0  | 0  | 0  | 0  | 1174264 |
| 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1174266 |
| 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | -1 | 0  | 0  | 0  | 0  | 1174276 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 3  | 0  | 0  | 0  | 0  | 1174296 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 1  | 0  | 0  | 0  | 0  | 1174328 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 1  | 0  | 0  | 0  | 0  | 1174396 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 1  | 0  | 0  | 0  | 0  | 1174534 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 2  | -4 | 0 | -1 | 0  | 0  | 0  | 0  | 0  | 1  | -1 | 0  | 0  | 0  | 0  | 1174810 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 2  | -4 | 1  | 0  | 0  | 0  | -1 | -1 | 0  | 0  | 0  | 0  | 1175364 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 2  | -1 | -4 | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1176470 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 0  | 1  | 2  | -4 | 0  | 0  | -1 | -1 | 0  | 0  | 0  | 1178676 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 2  | -4 | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1183098 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 2  | -4 | -3 | -3 | 0  | 0  | 0  | 0  | 1191934 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 2  | -1 | -1 | 0  | 0  | 0  | 0  | 1209608 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | 1  | -3 | 0  | 0  | 0  | 0  | 1244958 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | 1  | -4 | 0  | 0  | 0  | 1315654 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | 1  | 0  | 0  | 0  | 0  | -1 | -1 | 2  | -4 | 0  | 0  | 1457052 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 2  | -4 | 0  | 1739842 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 2  | -4 | 2305430 |
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0  | -1 | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 2  | 3436596 |
| 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3 | 3  | 0  | 3  | 3  | 3  | 3  | 0  | 0  | 3  | 3  | 3  | 3  | 782839  |

Sure enough, after some inspection he finds that the fifteenth column in  $\mathcal{B}_{\text{good}}$  is the shortest vector  $t$  in the lattice spanned by  $\mathcal{B}_{\text{bad}}$ . He can further verify that the norm  $\|t\| = \sqrt{n} = \sqrt{25} = 5$ . He then lets,

$$x \equiv t - 1 \equiv [0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1] \pmod{3}.$$

Breaking  $x$  per encoding Table 3.4,

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 00011 | 01000 | 00101 | 01110 | 00111 |
| ↓     | ↓     | ↓     | ↓     | ↓     |
| 3     | 8     | 5     | 14    | 7     |
| ↓     | ↓     | ↓     | ↓     | ↓     |
| C     | H     | E     | N     | G     |

### 3.4.2 Key Sizes

The [Merkle and Hellman \(2019\)](#) knapsack cryptosystem is based on an  $\mathcal{NP}$ -complete problem but as Example 4 shows the key size  $n$  matters. Clearly,  $n = 25$  is not large enough. How large should  $n$  be to be computationally secure? LLL approximates the shortest vector by finding another vector in the lattice  $\mathcal{L}$  which is *no longer than*  $\left(\frac{4}{3}\right)^{\frac{\dim(\mathcal{L})-1}{2}}$  times the length of the shortest vector. We might use this theoretically provable upper bound to judge an  $n = 150$ ,

$$\left(\frac{4}{3}\right)^{\frac{151-1}{2}} < 2346417266$$

That is large. However, [Gama and Nguyen \(2008\)](#) show a gap between theoretically provable and empirically predictable upper-bound.

$$(1.0219)^{\frac{151-1}{2}} < 5.1$$

This means that for a cryptosystem depending upon the hardness of approximating the shortest vector in a lattice to be secure, we need  $\dim(\mathcal{L}) = n + 1 > 150$ . The ideal choice as of this work is  $d = 300$ .

# METHODOLOGY

In this chapter, we intuit, describe and analyse the core components of our domain-informed representation for the evolutionary lattice sieves.

## 4.1 A Naïve Sieve

Algorithm 4 checks the difference of all possible  $(u, v) \in P^2$  for a shorter non-zero vector, not already present in  $P$ . These newer and shorter vectors are collected in  $R$ . At each iteration, we combine  $P$  and  $R$ , selecting out of the combination at most  $|P|$  successive shortest vectors.

**Input:**  $P \subset \mathcal{L}$ . **Output:** Reduced version of subset:  $P$ .

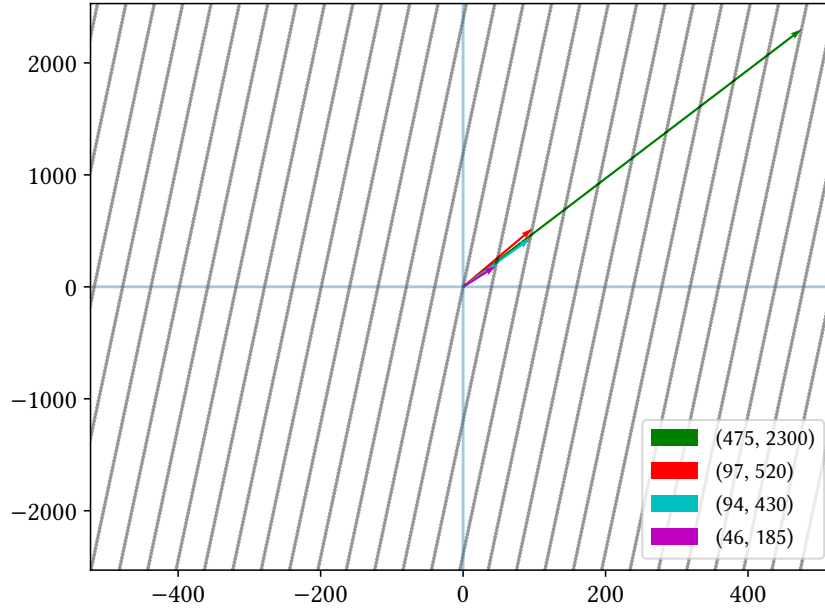
```

1: do
2:    $R \leftarrow \emptyset$ 
3:   for each  $(u, v) \in P^2$ 
4:      $t \leftarrow v - u$ 
5:     if  $(\vec{0} \neq t \notin P) \wedge (\|t\| < \|u\| \vee \|t\| < \|v\|)$  then  $R \leftarrow R \cup \{t\}$ 
6:    $P \leftarrow \text{Elite}(P \cup R)$ 
7: while  $R \neq \emptyset$ 

```

**ALGORITHM 4**—Naïve sieving algorithm with global selection. **Elite** $(P \cup R)$  selects  $|P|$  shortest (fittest) vectors (individuals) among  $P \cup R$ .

*Example 5* (Algorithm 4 reducing the bad basis  $\mathcal{B}_{\text{bad}}$  given in Equation 2.1 of Chapter 2).



**FIGURE 4.1**—A sample given in Eq. 4.1 of lattice vectors spanned by basis given in Eq. 2.1. It is used as initial population  $P$  to demonstrate Alg. 4 in Fig. 4.2.

Iteration 1 (Vectors marked by  $\checkmark$  are included in  $R$ ):

$$P = \begin{bmatrix} 46 & 94 & 97 & 475 \\ 185 & 430 & 520 & 2300 \end{bmatrix} \quad \text{Figure 4.1} \quad (4.1)$$

$$\langle 46, 185 \rangle - \langle 94, 430 \rangle = \langle -48, -245 \rangle \checkmark$$

$$\langle 46, 185 \rangle - \langle 97, 520 \rangle = \langle -51, -335 \rangle \checkmark$$

$$\langle 46, 185 \rangle - \langle 475, 2300 \rangle = \langle -429, -2115 \rangle \checkmark$$

$$\langle 94, 430 \rangle - \langle 97, 520 \rangle = \langle -3, -90 \rangle \checkmark$$

$$\langle 94, 430 \rangle - \langle 475, 2300 \rangle = \langle -381, -1870 \rangle \checkmark$$

$$\langle 97, 520 \rangle - \langle 475, 2300 \rangle = \langle -378, -1780 \rangle \checkmark$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -3 & 46 & -48 & -51 \\ -90 & 185 & -245 & -335 \end{bmatrix}$$

Iteration 2:

$$P = \begin{bmatrix} -3 & 46 & -48 & -51 \\ -90 & 185 & -245 & -335 \end{bmatrix}$$

Figure 4.2a

$$\langle -3, -90 \rangle - \langle 46, 185 \rangle = \langle -49, -275 \rangle$$

$$\langle -3, -90 \rangle - \langle -48, -245 \rangle = \langle 45, 155 \rangle \checkmark$$

$$\langle -3, -90 \rangle - \langle -51, -335 \rangle = \langle 48, 245 \rangle$$

$$\langle 46, 185 \rangle - \langle -48, -245 \rangle = \langle 94, 430 \rangle$$

$$\langle 46, 185 \rangle - \langle -51, -335 \rangle = \langle 97, 520 \rangle$$

$$\langle -48, -245 \rangle - \langle -51, -335 \rangle = \langle 3, 90 \rangle$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -3 & 45 & 46 & -48 \\ -90 & 155 & 185 & -245 \end{bmatrix}$$

Iteration 3:

$$P = \begin{bmatrix} -3 & 45 & 46 & -48 \\ -90 & 155 & 185 & -245 \end{bmatrix}$$

Figure 4.2b

$$\langle -3, -90 \rangle - \langle 45, 155 \rangle = \langle -48, -245 \rangle$$

$$\langle -3, -90 \rangle - \langle 46, 185 \rangle = \langle -49, -275 \rangle$$

$$\langle -3, -90 \rangle - \langle -48, -245 \rangle = \langle 45, 155 \rangle$$

$$\langle 45, 155 \rangle - \langle 46, 185 \rangle = \langle -1, -30 \rangle \checkmark$$

$$\langle 45, 155 \rangle - \langle -48, -245 \rangle = \langle 93, 400 \rangle$$

$$\langle 46, 185 \rangle - \langle -48, -245 \rangle = \langle 94, 430 \rangle$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -1 & -3 & 45 & 46 \\ -30 & -90 & 155 & 185 \end{bmatrix}$$

Iteration 4:

$$P = \begin{bmatrix} -1 & -3 & 45 & 46 \\ -30 & -90 & 155 & 185 \end{bmatrix}$$

Figure 4.2c

$$\langle -1, -30 \rangle - \langle -3, -90 \rangle = \langle 2, 60 \rangle \checkmark$$

$$\langle -1, -30 \rangle - \langle 45, 155 \rangle = \langle -46, -185 \rangle$$

$$\langle -1, -30 \rangle - \langle 46, 185 \rangle = \langle -47, -215 \rangle$$

$$\langle -3, -90 \rangle - \langle 45, 155 \rangle = \langle -48, -245 \rangle$$

$$\langle -3, -90 \rangle - \langle 46, 185 \rangle = \langle -49, -275 \rangle$$

$$\langle 45, 155 \rangle - \langle 46, 185 \rangle = \langle -1, -30 \rangle$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -1 & 2 & -3 & 45 \\ -30 & 60 & -90 & 155 \end{bmatrix}$$

Iteration 5:

$$P = \begin{bmatrix} -1 & 2 & -3 & 45 \\ -30 & 60 & -90 & 155 \end{bmatrix}$$

Figure 4.2d

$$\langle -1, -30 \rangle - \langle 2, 60 \rangle = \langle -3, -90 \rangle$$

$$\langle -1, -30 \rangle - \langle -3, -90 \rangle = \langle 2, 60 \rangle$$

$$\langle -1, -30 \rangle - \langle 45, 155 \rangle = \langle -46, -185 \rangle$$

$$\langle 2, 60 \rangle - \langle -3, -90 \rangle = \langle 5, 150 \rangle$$

$$\langle 2, 60 \rangle - \langle 45, 155 \rangle = \langle -43, -95 \rangle \checkmark$$

$$\langle -3, -90 \rangle - \langle 45, 155 \rangle = \langle -48, -245 \rangle$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -1 & 2 & -3 & -43 \\ -30 & 60 & -90 & -95 \end{bmatrix}$$

Iteration 6:

$$P = \begin{bmatrix} -1 & 2 & -3 & -43 \\ -30 & 60 & -90 & -95 \end{bmatrix}$$

Figure 4.2e

$$\langle -1, -30 \rangle - \langle 2, 60 \rangle = \langle -3, -90 \rangle$$

$$\langle -1, -30 \rangle - \langle -3, -90 \rangle = \langle 2, 60 \rangle$$

$$\langle -1, -30 \rangle - \langle -43, -95 \rangle = \langle 42, 65 \rangle \checkmark$$

$$\langle 2, 60 \rangle - \langle -3, -90 \rangle = \langle 5, 150 \rangle$$

$$\langle 2, 60 \rangle - \langle -43, -95 \rangle = \langle 45, 155 \rangle$$

$$\langle -3, -90 \rangle - \langle -43, -95 \rangle = \langle 40, 5 \rangle \checkmark$$

$$\mathbf{Elite}(P \cup R) = \begin{bmatrix} -1 & 40 & 2 & 42 \\ -30 & 5 & 60 & 65 \end{bmatrix}$$

Iteration 7:

$$P = \begin{bmatrix} -1 & 40 & 2 & 42 \\ -30 & 5 & 60 & 65 \end{bmatrix}$$

Figure 4.2f

$$\langle -1, -30 \rangle - \langle 40, 5 \rangle = \langle -41, -35 \rangle$$

$$\langle -1, -30 \rangle - \langle 2, 60 \rangle = \langle -3, -90 \rangle$$

$$\langle -1, -30 \rangle - \langle 42, 65 \rangle = \langle -43, -95 \rangle$$

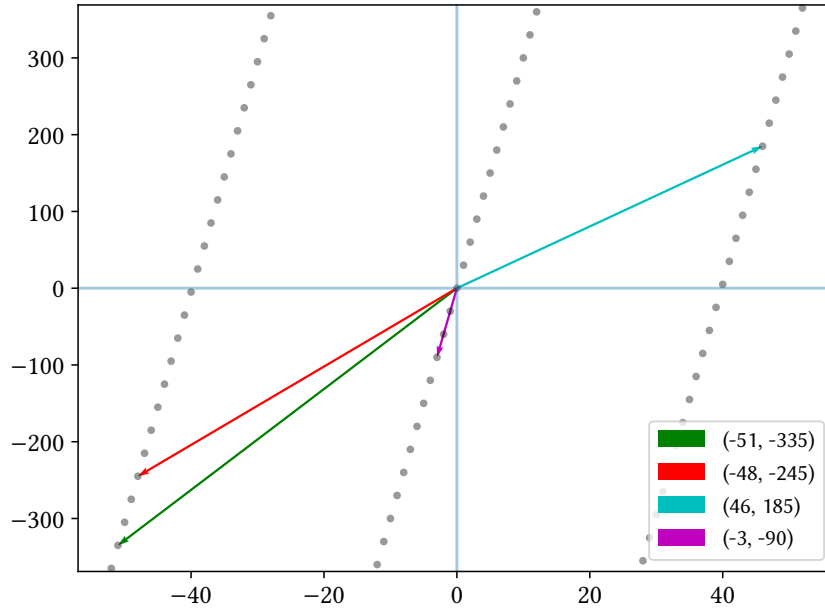
$$\langle 40, 5 \rangle - \langle 2, 60 \rangle = \langle 38, -55 \rangle$$

$$\langle 40, 5 \rangle - \langle 42, 65 \rangle = \langle -2, -60 \rangle$$

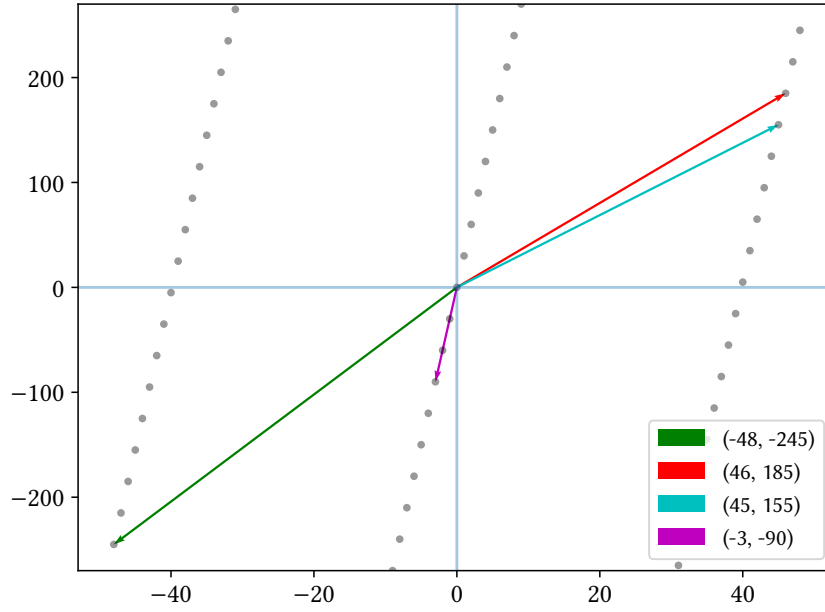
$$\langle 2, 60 \rangle - \langle 42, 65 \rangle = \langle -40, -5 \rangle$$

$$\mathbf{Elite}(P \cup R) = P$$

Figure 4.1 illustrates a plot of this initial population. Figure 4.2 shows the subsequent new  $P$  at each new iteration of a total of seven iterations.

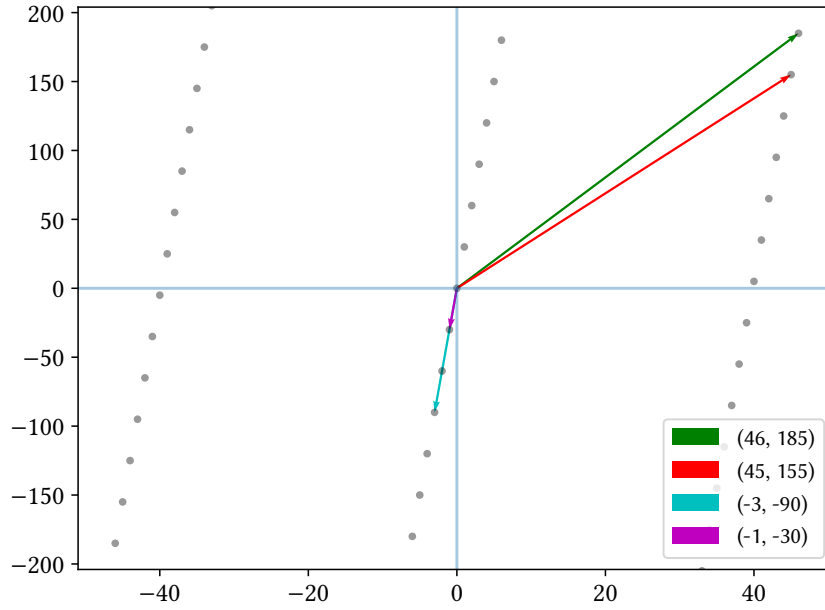


(A) Iteration 2.

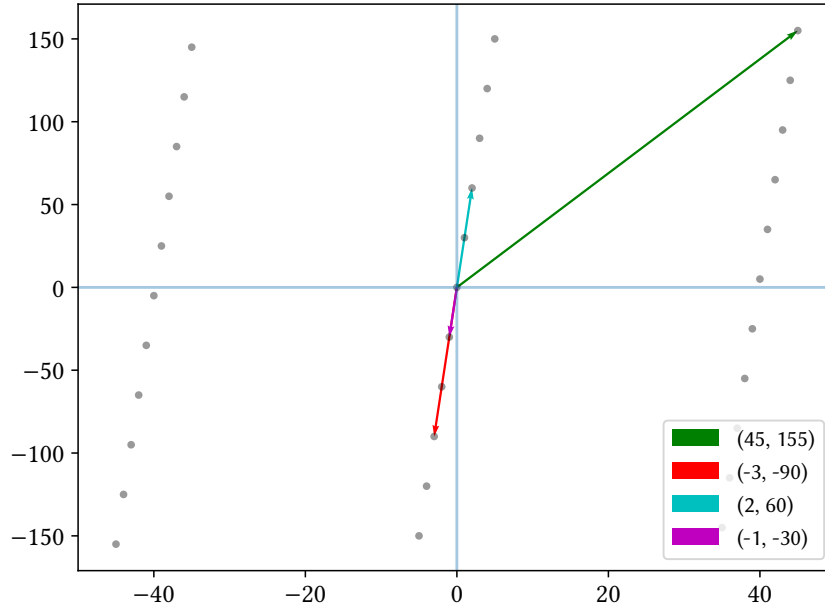


(B) Iteration 3.

**FIGURE 4.2**—Algorithm 4 reducing  $\mathcal{B}_{\text{bad}}$  in Equation 2.1 starting with population  $P$  in Equation 4.1 and Figure 4.1. Note that in this case, we find the shortest vector by solving the SBP exactly. The seventh iteration contains  $\mathcal{B}_{\text{good}}$  of Equation 2.1.

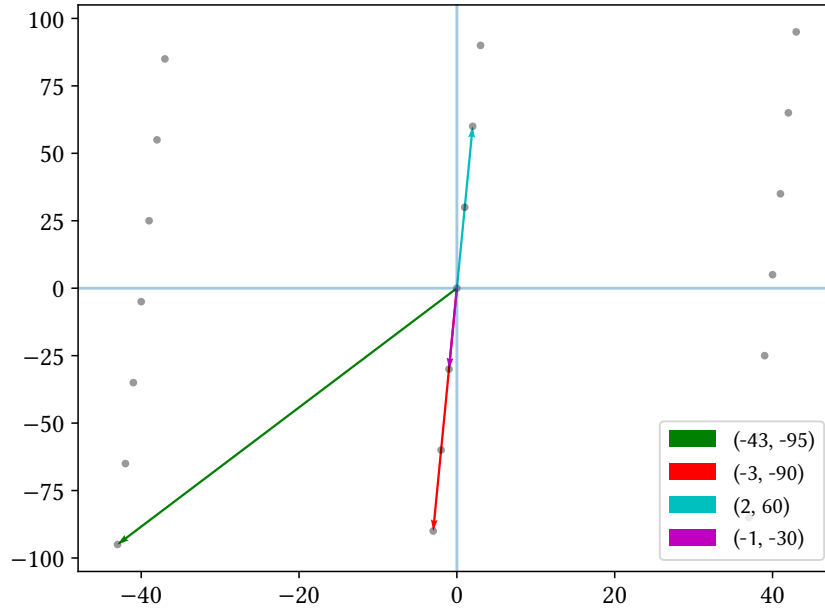


(c) Iteration 4.

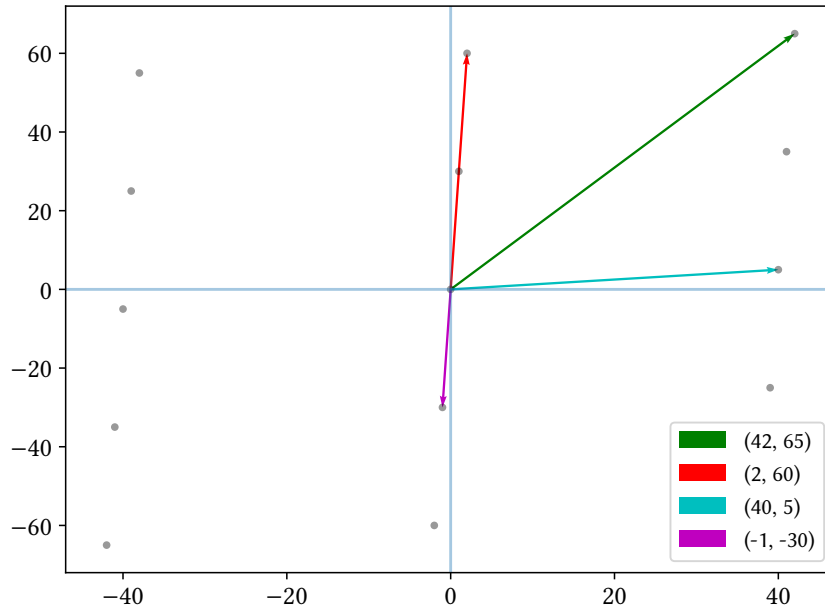


(d) Iteration 5.

**FIGURE 4.2**—Algorithm 4 reducing  $\mathcal{B}_{\text{bad}}$  in Equation 2.1 starting with population  $P$  in Equation 4.1 and Figure 4.1. Note that in this case, we find the shortest vector by solving the SBP exactly. The seventh iteration contains  $\mathcal{B}_{\text{good}}$  of Equation 2.1.



(E) Iteration 6.



(F) Iteration 7.

**FIGURE 4.2**—Algorithm 4 reducing  $\mathcal{B}_{\text{bad}}$  in Equation 2.1 starting with population  $P$  in Equation 4.1 and Figure 4.1. Note that in this case, we find the shortest vector by solving the SBP exactly. The seventh iteration contains  $\mathcal{B}_{\text{good}}$  of Equation 2.1.

## 4.2 Our Genetic Algorithm for Sieving

We specify each of the components of the improved genetic algorithm used in this work to solve the apprSVP. Pseudocode is given in Algorithm 5.

**Input:** (1) Basis:  $\mathcal{B}$ , (2) Population size:  $n$  (3) Sampling density:  $\rho$

**Output:** Approximation of the shortest vector.

```

1:  $C_{d,n} \leftarrow (c_{i,j} \sim \text{Bernoulli}(\rho))$ 
2:  $P \leftarrow \mathcal{B}C$ 
3:  $R \leftarrow \emptyset$ 
4: do
5:    $P \leftarrow \text{Elite}(P \cup R)$ 
6:    $R \leftarrow \emptyset$ 
7:   do
8:      $u, v \leftarrow \text{Select}(P)$ 
9:      $t \leftarrow \text{Cross}(u, v)$ 
10:    if  $(\vec{0} \neq t \notin P) \wedge (\|t\| < \|u\| \vee \|t\| < \|v\|)$ 
11:       $R \leftarrow R \cup \{t\}$ 
12:    while  $|R| < n^{1.5}$ 
13: while  $\forall v \in P, \text{Fitness}(v) < w^{-1}$ 

```

**ALGORITHM 5**—The evolutionary sieving algorithm.  $\text{Elite}(P \cup R)$  returns the  $|P|$  many fittest vectors from  $P \cup R$ .  $\text{Select}(P)$  generates the elements of  $P^2$  one by one as shown in Algorithm 6.  $\text{Cross}(u, v)$  is defined in Section 4.2.2 with a 2-dimensional illustration in Figure 4.3. Lastly,  $\text{Fitness}(v) = \|v\|_2^{-1}$ . The algorithm continues evolving generations until an individual with norm  $w$  or less is found. See Section 4.2. A readable albeit slow Python implementation is given in Code Listing B.4.

### 4.2.1 Genotype

We represent vectors  $v = \langle v_1, v_2, v_3, \dots, v_d \rangle \in \mathcal{L}$  as is. Note that here  $\mathcal{L} \subseteq \mathbb{Z}^d$  or  $\mathcal{L} \subseteq \mathbb{Z}[i]^d$  (see Section 2.3).

*Remark 4.1.* Laarhoven encode their individual for a lattice vector  $v = \mathcal{B}x$  as the integral combination vector  $x$  to be able to mutate  $v$  by adding a small amount of binary noise to  $x$ . A consequence of their encoding is that now a matrix multiplication

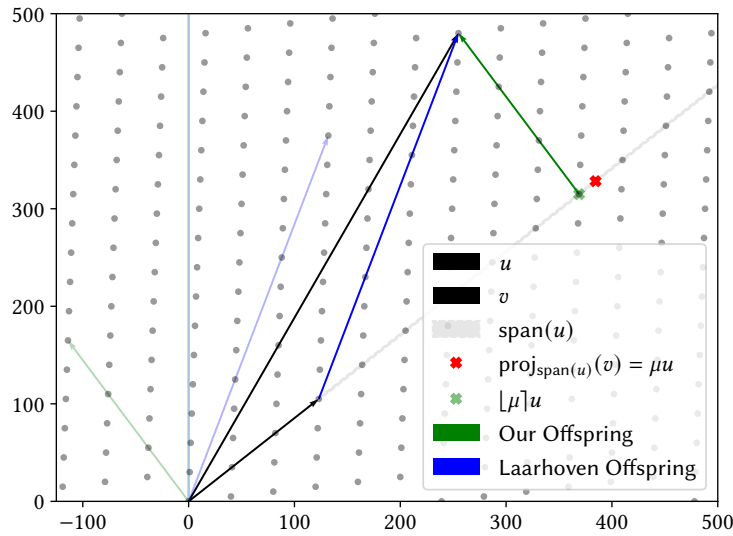
$\mathcal{B}x$  must precede the evaluation of  $v$ 's fitness. Our encoding of  $v$  avoids this issue.

## 4.2.2 Crossover

For parents,  $u, v \in P$ , we may generate the offspring  $t$ ,

$$t = \text{Cross}(u, v) = v - \lfloor \mu \rfloor u \quad \text{where} \quad \mu = \frac{\Re(u \cdot v)}{u \cdot u} + \frac{\Im(u \cdot v)}{u \cdot u}i.$$

If  $u, v \in \mathbb{Z}^d$  then the imaginary part  $\Im(u \cdot v) = 0$ . We round off a complex number  $z = \alpha + \beta i$  where  $(\alpha, \beta) \in \mathbb{R}^2$  by  $\lfloor z \rfloor = \lfloor \alpha \rfloor + \lfloor \beta \rfloor i$ . See Figure 4.3 for a geometric



**FIGURE 4.3**—Illustration of our crossover technique vs. Laarhoven's. Letting  $u$  be the shorter vector we realise that multiple copies  $\lfloor \mu \rfloor$  of  $u$  could be subtracted from  $v$ . This constant  $\mu$  is used in  $\text{proj}_{\text{span}(u)}(v) = \mu u$ . However,  $\text{proj}_{\text{span}(u)}(v)$  is not necessarily in the lattice. Therefore, instead we subtract  $\lfloor \mu \rfloor u$  from  $v$  making our offspring much shorter (fitter) than  $v - u$ .

interpretation of the crossover in 2 dimensions. The idea is that if  $(u, v)$  are close to each other then the offspring  $t$  is of high fitness but in the average case,  $(u, v)$  are not near. Letting  $u$  be the shorter vector we realise that multiple copies  $\lfloor \mu \rfloor$  of  $u$  could be subtracted from  $v$ . This constant  $\mu$  is used to evaluate the projection of  $v$  onto the span of  $u$ , i.e.,  $\text{proj}_{\text{span}(u)}(v) = \mu u$ . However, as seen in Figure 4.3,  $\text{proj}_{\text{span}(u)}(v)$  is not

necessarily in the lattice. Therefore, instead of  $\text{proj}_{\text{span}(u)}(v)$  we subtract  $\lfloor \mu \rfloor u$  from  $v$  getting a much shorter (fitter) offspring than Laarhoven's offspring  $v - u$ .

### 4.2.3 Fitness Function

We evaluate the fitness of a vector  $v$  by the inverse of its  $\ell^2$  norm. Observe that one may also invert the squared norm.

$$\text{Fitness}(v) = \frac{1}{\|v\|_2} \propto \frac{1}{\|v\|_2^2} = \frac{1}{v \cdot v}$$

### 4.2.4 Mutation

The  $j^{\text{th}}$  column of the initial population and the subsequent generations is given as follows.

$$\begin{aligned} P_j &= \begin{bmatrix} b_{1,1}x_{1,j} + b_{1,2}x_{2,j} + \cdots + b_{1,d}x_{d,j} \\ b_{2,1}x_{1,j} + b_{2,2}x_{2,j} + \cdots + b_{2,d}x_{d,j} \\ \vdots \\ b_{d,1}x_{1,j} + b_{d,2}x_{2,j} + \cdots + b_{d,d}x_{d,j} \end{bmatrix} \\ &= \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,d} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ b_{d,1} & b_{d,2} & \cdots & b_{d,d} \end{bmatrix} \begin{bmatrix} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{d,j} \end{bmatrix} \\ &= \mathcal{B}x_j \end{aligned}$$

Laarhoven (2019) represent each individual in the population as  $x_j$  instead of  $\mathcal{B}x_j$ . This enabled them to mutate  $P_j = \mathcal{B}x_j$  by adding a small integral perturbation to  $x_{i,j}$ . However, this representation requires a matrix multiplication before the fitness (inverse  $\ell^2$  norm) of each individual can be calculated. In our representation, if an offspring was to be mutated, we may multiply  $\mu$  (Section 4.2.2) by a normal random variable  $\xi \sim \mathcal{N}(1, 1)$  with mean and standard deviation 1, i.e.,  $t_{\text{mutated}} = v - \lfloor \xi \mu \rfloor u$ . The crossover

step can now be given as  $\text{Cross}(u, v) = v - \lfloor \xi \mu \rfloor u$  where either  $\xi = 1$  or  $\xi \sim \mathcal{N}(1, 1)$  under a small probability. As an example, for each  $\xi \in \{0.1, 0.4, 0.7, 1, 1.3, 1.6, 1.9\}$ ,

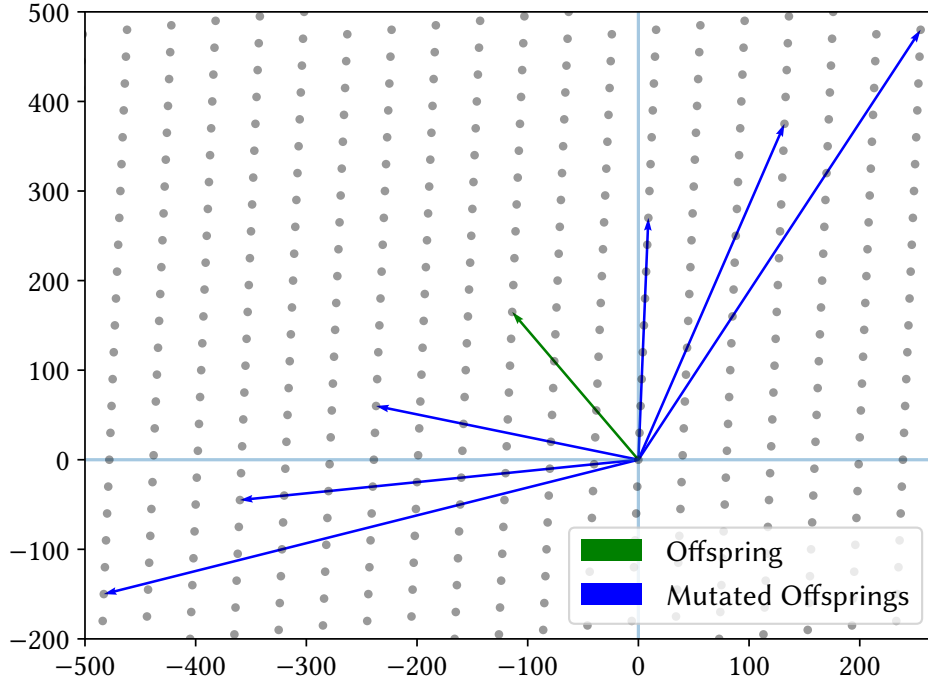


FIGURE 4.4—Mutations of the offspring in Figure 4.3.

Figure 4.4 shows the corresponding six possible mutations ( $\xi = 1$  is not mutated) of our offspring shown in Figure 4.3.

### 4.2.5 Initial Population

Before we proceed with the generation of the initial population using  $\mathcal{B}$ , we may *optionally* compute the *Hermite normal form* of  $\mathcal{B}$  as well as reduce it using the LLL algorithm for some  $\delta$  (Section 3.1.2).

Hermite normal form is a way of arranging a basis so that each  $b_i$  points in a direction that is as “simple” as possible while still spanning the same lattice. In this arrangement, every vector after the first aligns with a coordinate axis, and any ear-

lier vector contributes a whole-number multiple to the later ones, ensuring a clean, stepped structure. The resulting set is unique for a given lattice, providing a canonical, hierarchical representation of the original basis.

*Remark 4.2.* The Hermite normal form is to integral matrices what the reduced echelon form is to the matrices over the reals.

Letting  $n = |P|$ , we generate the initial population via a constant  $d$  by  $n$  binary matrix  $C$  like this:  $P = \mathcal{B}C$ . If  $\mathcal{L}(\mathcal{B}) \subseteq \mathbb{Z}^d$  or  $\mathcal{B} \in \mathbb{Z}^{d \times d}$ ,

$$C \in \{0, 1\}^{d \times n} \quad \text{where} \quad \Pr(C_{i,j} = 1) = \rho.$$

However, if  $\mathcal{L}(\mathcal{B}) \subseteq \mathbb{Z}[i]^d$  or  $\mathcal{B} \in \mathbb{Z}[i]^{d \times d}$  then for  $\alpha, \beta \in \{0, 1\}$ ,

$$C_{i,j} = \alpha + \beta i \quad \text{where} \quad \Pr(\alpha = 1) = \Pr(\beta = 1) = \rho.$$

#### 4.2.6 Selection Strategy

Let  $P$  be the previous generation and  $R$  be the current generation, or the generation produced as a result of the reproduction among  $P$ . The next generation is then produced by picking the  $|P|$  successive shortest vectors from the pool  $P \cup R$  and then assigning them back to  $P \leftarrow \text{Elite}(P \cup R)$ . Observe how this implies that  $P$  will now be sorted by vector length in ascending order. We will use this  $P$  to select the individuals for crossover. Let  $u_i, v_j$  be the  $i^{\text{th}}$  and  $j^{\text{th}}$  vectors in  $P$  then Algorithm 6 states how we can generate pairings of vectors in  $P$ . Note that Algorithm 6 generates at most  $|P|$  choose 2  $= 0.5n(n-1)$  pairs, *yielding* them one by one for possible reproduction in Algorithm 5.

**Input:** Population:  $P$ .

**Output:** Generated pairs for reproduction.

```
1: for  $i \in \{1, 2, 3, \dots, |P| - 1\}$ 
2:   for  $j \in \{i + 1, \dots, |P|\}$ 
3:     yield  $(u_i, v_j)$ 
```

**ALGORITHM 6**—`yield` ( $u_i, v_j$ ) yields each pair to `Select`( $P$ ) in Algorithm 5. This is enumerating a subset of  $P^2$ .

## 4.3 Analysis of the Genetic Algorithms for Sieving

Let  $m$  count the maximum generations,  $n = |P|$  be the size of the initial population and  $d$  be the dimension of the lattice. On average, our algorithm runs in  $O(dmn^{1.5})$ . This is due to the  $O(d)$  cost of each fitness evaluation times the total number of generations  $m$  times the total number of expected  $n^{1.5}$  crossovers.

In the case of [Laarhoven's \(2019\)](#) algorithm, we keep the same  $m$  generations,  $n$  individuals and note that they consider all possible children for a potential insertion in the next generation. Furthermore, they represent the  $j^{\text{th}}$  individual  $P_j = \mathcal{B}x_j$  as  $x_j$  in contrast to  $\mathcal{B}x_j$  to stay compatible with their mutation operation. Their representation of  $P_j$  as  $x_j$  introduces a matrix multiplication (namely  $\mathcal{B}$  multiplied on the left with  $x_j$ ) each time fitness is to be calculated since fitness is a function of  $\|\mathcal{B}x_j\|$ . Therefore, Laarhoven's algorithm can be shown to run in  $O(d^3mn^2)$  on average.

The exponent 1.5 of  $n^{1.5}$  in our algorithm was determined by a combination of empirical analysis and heuristics from the original choices of  $\delta$  in the LLL algorithm as explained in Section [3.1.2](#).

# RESULTS

In this chapter, we explain the experiments and present the observed outcomes, including the raw data and its interpretations. Recall that lattice-based cryptography relies on the inability of lattice reduction algorithms to solve the apprSVP in high dimensions ( $d > 300$ ) with an approximation factor  $\alpha = \text{poly}(d)$  (Section 2.2.3 and Section 3.1).

For dimensions  $d \leq 100$  in integral and  $d \leq 50$  in module lattices, we optimally solve the apprSVP with an  $\alpha < 2.05$ . The results are summarised in Table 5.1.

## 5.1 Choosing Data and Hyperparameters

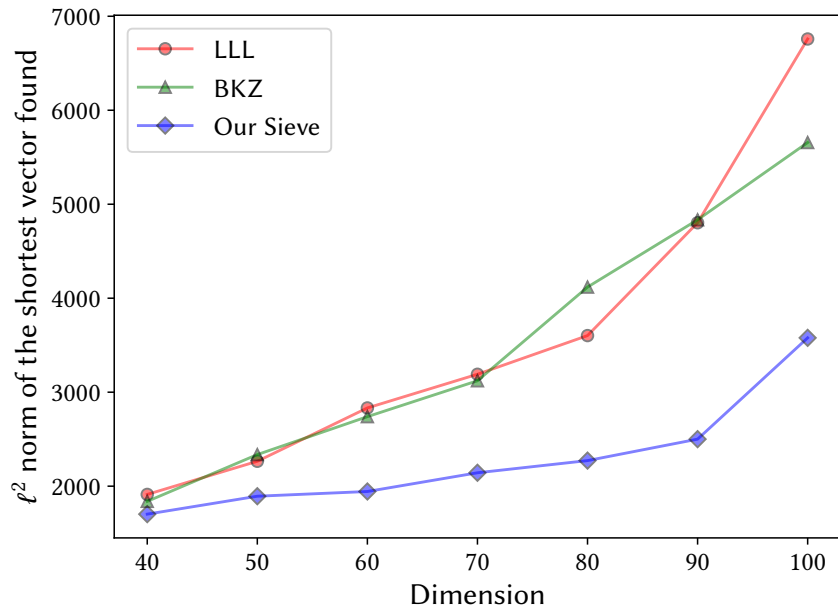
We preprocessed the basis with the state-of-the-art LLL algorithm with a reduction parameter  $\delta = 0.9999999$ , i.e., fairly close to 1. This puts our algorithm in direct comparison with the LLL reduction algorithm. Figure 5.1 further compares the reduction power of LLL as well as BKZ with  $\delta = 0.9999999$  and the standard block-size of 10 against our algorithm. We outperform both LLL and BKZ.

The experiments reducing the SVP challenge and the random integral lattices in dimensions 40, 50, 60, 70 and 80 were carried out twice: once with no mutations and once with 1% chance of mutation. The resulting shortest vectors were the same in either case. However, the experiments with mutations took about twice as long. Afterwards, experiments on lattices of dimensions  $d \in \{90, 100\}$  and module lattices were carried

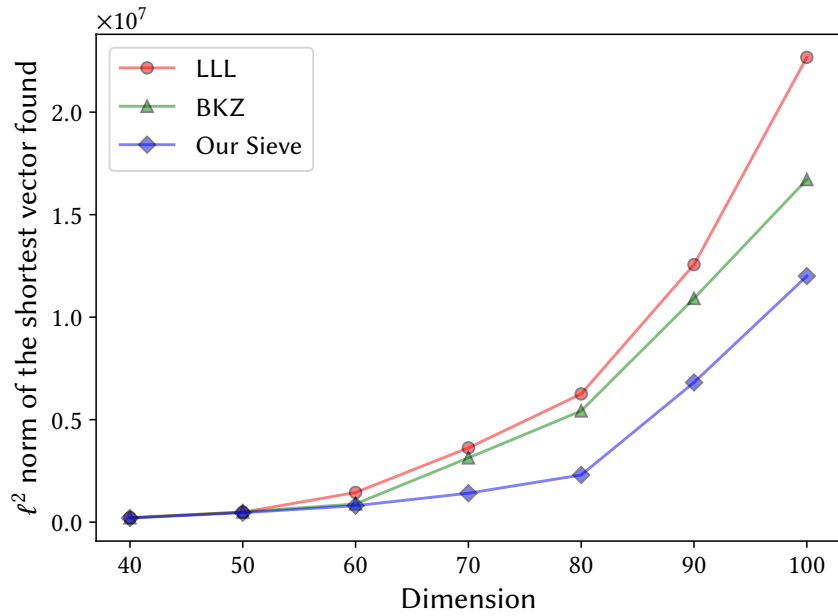
**TABLE 5.1**—A comparison of the shortest vector length in each dimension ( $d$ ) found via the Algorithm 5 ( $\ell$ ) against the expected shortest length due to the Gaussian Heuristic. Recall from Section 2.2.4 that the exact length of the shortest non-zero vector is unknown in the general case and therefore, for a sufficiently large  $d$  (Hoffstein et al., 2008), we may rely on the Gaussian expected shortest length ( $\sigma$ ). By Definition 2.2, we solve apprSVP for all  $\alpha \leq \ell/\sigma$ . The column  $g$  is the total number of generations evolved and  $n$  gives the size of the initial population. Each vector itself is given in Appendix C.

| SVP CHALLENGE LATTICES |          |                   |               |     |         | SVP RANDOM INTEGRAL LATTICES |                   |               |     |         |  |
|------------------------|----------|-------------------|---------------|-----|---------|------------------------------|-------------------|---------------|-----|---------|--|
| $d$                    | $\sigma$ | Alg. 5 ( $\ell$ ) | $\ell/\sigma$ | $g$ | $n$     | $\sigma$                     | Alg. 5 ( $\ell$ ) | $\ell/\sigma$ | $g$ | $n$     |  |
| 40                     | 1560     | 1702.46           | 1.09          | 7   | 1000    | 212828                       | 201343.89         | 0.95          | 10  | 800     |  |
| 50                     | 1746     | 1893.17           | 1.08          | 8   | 1250    | 522364                       | 460101.23         | 0.88          | 13  | 1000    |  |
| 60                     | 1916     | 1943.40           | 1.01          | 16  | 9900    | 1113319                      | 803170.35         | 0.72          | 25  | 3000    |  |
| 70                     | 2065     | 2142.60           | 1.04          | 11  | 49980   | 2071820                      | 1411636.20        | 0.68          | 18  | 28000   |  |
| 80                     | 2205     | 2272.19           | 1.03          | 16  | 240000  | 3351770                      | 2302176.05        | 0.69          | 16  | 160000  |  |
| 90                     | 2348     | 2500.40           | 1.06          | 25  | 1638000 | 5564035                      | 6815686.92        | 1.22          | 18  | 540000  |  |
| 100                    | 2468     | 3578.23           | 1.45          | 16  | 1500000 | 8339504                      | 12002723.15       | 1.44          | 16  | 1000000 |  |

| MODULE LATTICES |          |                        |               |     |      |
|-----------------|----------|------------------------|---------------|-----|------|
| $d$             | $\sigma$ | Algorithm 5 ( $\ell$ ) | $\ell/\sigma$ | $g$ | $n$  |
| 20              | 22947    | 26188.63               | 1.14          | 5   | 2000 |
| 30              | 136309   | 177814.21              | 1.30          | 4   | 3000 |
| 40              | 369551   | 754691.69              | 2.04          | 4   | 4000 |
| 50              | 863938   | 1482020.58             | 1.72          | 5   | 5000 |



(A) Shortest vector length comparison in the SVP challenge lattices.



(B) Shortest vector length comparison in the randomly generated lattices.

**FIGURE 5.1**—Comparison of the shortest vector length found by different algorithms.

out without mutations.

The stopping condition for each row in the results of Table 5.1 can be shown by letting  $w$  of Algorithm 5 be  $\ell$  (the shortest/most fit vector found). In practice, we stopped once the difference of the mean population length between the current and the last generation was less than 1 for consecutively three generations.

## 5.2 Reduction Results

Detailed results for reduction on Darmstadt’s SVP challenges (Nguyen, 2025) in dimensions  $40 \leq d \leq 100$  are given in Section 5.2.1. Parallely, we generated random integral lattices of equal dimensions to further test the performance of Algorithm 5. The results on these lattices are given in Section 5.2.2. Finally, we reduce module lattices of the dimensions 20, 30, 40 and 50, giving the results in Section 5.2.3. Check the Appendix A for an individual reduction curve of each dimension.

### 5.2.1 Reducing SVP Challenge Lattices

We started by reducing each of the lattices using the LLL algorithm as described in Section 3.1.2 for  $\delta = 1 - 10^{-7}$ . Note that the initial population included this reduced  $\mathcal{B}$  as is. This implies that the algorithm must find something better than the final results of the LLL for a  $\delta = 0.9999999$  fairly close to 1.

For the Algorithm 5 parameter  $\rho = 0.01$ , Table 5.1 summarises the results at the final step for each  $40 \leq d \leq 100$ . Up until 80 dimensions, our results (length of the shortest vector) match the best ones found so far. Furthermore, these were all solved on an *M2 MacBook Air* with 24 GB of memory.

### 5.2.2 Reducing Random Integral Lattices

We generate these lattices  $\mathcal{L}(\mathcal{B})$  by uniformly randomly generating each element  $b_{i,j}$  of  $\mathcal{B}$  between  $-d^3$  and  $d^3$ . Afterwards, we compute the Hermite normal form (this is to integral matrices what the reduced echelon form is for the matrices over the reals) of  $\mathcal{B}$  and reduce it with the LLL algorithm for  $\delta = 1 - 10^{-7}$ .

For the Algorithm 5 parameter  $\rho = 0.01$ , Table 5.1 summarises the results at the final step for each  $40 \leq d \leq 100$ . We optimally solve the apprSVP for  $\alpha < 1.5$  for all  $40 \leq d \leq 100$ .

### 5.2.3 Reducing Module Lattices

We generate these lattices  $\mathcal{L}(\mathcal{B})$  by generating each  $\alpha + \beta i \in \mathcal{B}$  containing  $\alpha$  and  $\beta$  picked uniformly randomly between  $-d^3$  and  $d^3$ . Furthermore, to each basis vector  $b_i \in \mathcal{B}$  we add  $d/4$  many other randomly chosen basis vectors  $b_j \in \mathcal{B}$ .

With the same Algorithm 5 parameter  $\rho = 0.01$  as before, Table 5.1 summarises the results at the final step. We solve the apprSVP for  $\alpha < 2.05$  for all  $20 \leq d \leq 50$ .

## 5.3 Interpreting Results

Among the SVP challenge Lattices, for  $d \leq 80$  dimensions, our resultant short vectors match the known-best ones found so far. For all the integral and module lattices, we optimally solve the apprSVP with  $\alpha < 1.5$  and  $\alpha < 2.05$ .

[Laarhoven \(2019\)](#) solved the same (and only attempted) 40-dimensional lattice as we did in our results. They reported 29 iterations starting with a set of 1500 lattice vectors to find the shortest vector of norm 1702 and an average vector of norm 2008. Our approach took 7 iterations starting with a set of 1000 lattice vectors to find the

shortest vector of norm 1702 and an average vector of norm 1981. Laarhoven found the same shortest vector as we did in 1.9 seconds compared to the 0.06 seconds it took our approach. While we cannot faithfully explain the difference in time due to the possible machine/implementation differences, we suspect the difference is due to the absence of the matrix multiplication overhead in our algorithm as explained in Sections [4.2.4](#) and [4.3](#). The reduction in the number of generations can be explained by the optimisations inspired by the classical algorithms in our crossover operator (Section [4.2.2](#)).

Evolutionary sieving for the shortest vector remains a relatively new and under-explored area. While we presently do not anticipate it posing a direct threat to existing cryptographic algorithms, this work demonstrates that interdisciplinary approaches, combining machine learning and number theory, can effectively reduce a lattice of 100 dimensions with a modest approximation factor. This promising intersection of fields opens up exciting future avenues.

## CONCLUSION AND FUTURE WORK

In this chapter, we end with the concluding remarks and provide possible avenues of future work.

### 6.1 Conclusion

Sieving algorithms have become one of the most practical methods for the SVP. We improved [Laarhoven's \(2019\)](#) representation of the Shortest Vector Problem for genetic algorithms to reduce the number of generations and population size required to obtain better results on average. Our improvements are due to the refined genotype and crossover born out of the insights from the classic number theoretic algorithms. We further showed the proposed algorithm's reduction potential on the increasingly significant module lattices.

It is important to recognise that scaling *any* method for dimensions beyond 100 presents considerable challenges due to the inherent complexities of lattice problems. This difficulty underscores the reliance of the state-of-the-art cryptographic schemes on such problems ([Alagic et al., 2019, 2020, 2022, 2025](#)).

## 6.2 Future Work

We can advance our understanding of the reduction capabilities of evolutionary approaches in dimensions greater than 100 in various ways. For instance, managing the population as described by [Goldman and Punch \(2014\)](#); [Bosman et al. \(2016\)](#) or selection through techniques such as tournament selection could yield better results. Additionally, implementing more sophisticated crossover strategies, such as performing multiple steps of Gauss reduction on parent vectors, or employing a nearest neighbour search for parent selection could enhance the effectiveness of these approaches.

Furthermore, evolutionary sieving techniques could be investigated within module lattices in rings beyond Gaussian integers, such as more general cyclotomic fields. This multifaceted exploration holds the potential to improve our ability to address the SVP in higher dimensions, thereby reinforcing the robustness of *Evolutionary Lattice Sieves*.



# BIBLIOGRAPHY

Olga Abramkina, Mubarak Yakubova, Tansaule Serikov, Yenlik Begimbayeva, and Bakhodyr Yakubov. Implementation of lattice theory into the TLS to ensure secure traffic transmission in IP networks based on IP PBX asterisk. *International Journal of Advanced Computer Science and Applications*, 15(10), 2024. doi: 10.14569/IJACSA.2024.0151076. URL <http://dx.doi.org/10.14569/IJACSA.2024.0151076>.

Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610, 2001.

Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status report on the first round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, January 2019. url=<https://csrc.nist.gov/pubs/ir/8240/final>.

Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, July 2020. url=<https://csrc.nist.gov/pubs/ir/8309/final>.

Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, July 2022. url=<https://csrc.nist.gov/pubs/ir/8413/upd1/final>.

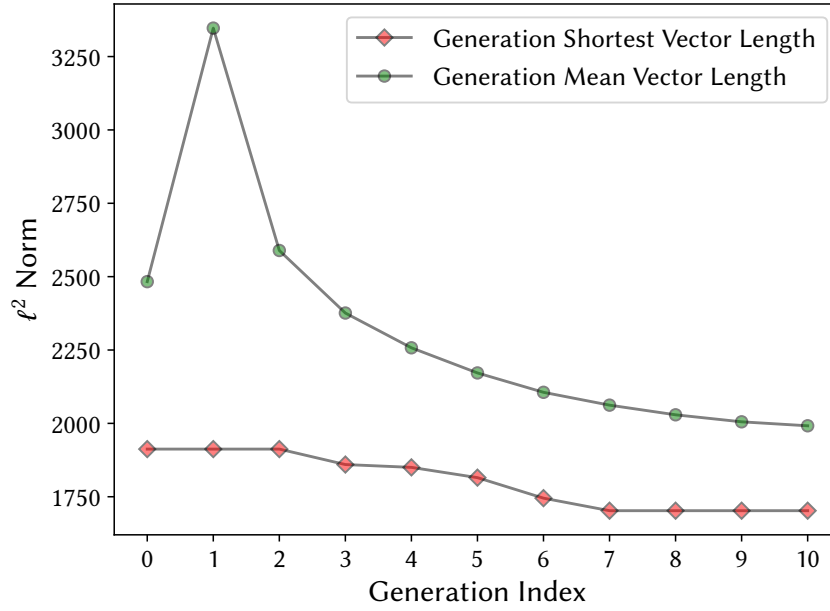
Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. Status report on the fourth round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, March 2025. <https://csrc.nist.gov/pubs/ir/8545/final>.

- László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.
- Sébastien Balny, Claire Delaplace, and Gilles Dequen. Another L makes it better? lagrange meets LLL and may improve BKZ pre-processing. In *2025 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 181–193. SIAM, 2025.
- Peter AN Bosman, Ngoc Hoang Luong, and Dirk Thierens. Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 637–644, 2016.
- Koen de Boer and Wessel van Woerden. Lattice-based cryptography: A survey on the security of the lattice-based NIST finalists. Cryptology ePrint Archive, Paper 2025/304, 2025. URL <https://eprint.iacr.org/2025/304>.
- Nicolas Gama and Phong Q Nguyen. Predicting lattice reduction. In *Advances in Cryptology–EUROCRYPT 2008: 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13–17, 2008. Proceedings 27*, pages 31–51. Springer, 2008.
- Brian W Goldman and William F Punch. Parameter-less population pyramid. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 785–792, 2014.
- Piet Hein. Grooks. *Brigham Young University Studies*, 11(1):32, 1970. url=<https://scholarsarchive.byu.edu/byusq/vol11/iss1/8/>.
- Jeffrey Hoffstein, Jill Pipher, Joseph H Silverman, and Joseph H Silverman. *An introduction to mathematical cryptography*, volume 1, pages 436, 377, 402, 405. Springer, 2008.
- Alex Kalbach and Ted Chinburg. LLL algorithm for lattice basis reduction. *arXiv preprint arXiv:2410.22196*, 2024.
- R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–104, New York, 1972. Plenum.
- Donald Ervin Knuth. *The Art of Computer Programming*, volume 2<sup>nd</sup>, chapter 4, page 335. Mathematical Sciences Publishers, 3<sup>rd</sup> edition, March 2014. url=<https://www-cs-faculty.stanford.edu/~knuth/taocp.html>.
- Thijs Laarhoven. Evolutionary techniques in lattice sieving algorithms. In *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019*, pages 31–39, Setubal, PRT, 2019. SCITEPRESS - Science and Technology Publications,

- Lda. ISBN 9789897583841. doi: 10.5220/0007968800310039. URL <https://doi.org/10.5220/0007968800310039>.
- Jeffrey C Lagarias and Andrew M Odlyzko. Solving low-density subset sum problems. *Journal of the ACM (JACM)*, 32(1):229–246, 1985.
- Rudolf Lidl and Günter Pilz. *Applications of Lattices*, pages 56–119. Springer US, New York, NY, 1984. ISBN 978-1-4615-6465-2. doi: 10.1007/978-1-4615-6465-2\_2. URL [https://doi.org/10.1007/978-1-4615-6465-2\\_2](https://doi.org/10.1007/978-1-4615-6465-2_2).
- Ralph C Merkle and Martin E Hellman. Hiding information and signatures in trapdoor knapsacks. In *Secure communications and asymmetric cryptosystems*, pages 197–215. Routledge, 2019.
- Daniele Micciancio. Lattice-based cryptography. In *Encyclopedia of Cryptography, Security and Privacy*, pages 1400–1403. Springer, 2025.
- Melanie Mitchell. *Complexity: A guided tour*, chapter Chapter 9: Genetic Algorithms. Oxford university press, 2009.
- Giang Nam Nguyen. Shortest vector problem challenge (darmstadt lattice challenge), 2025. URL <https://www.latticechallenge.org/svp-challenge/>. Accessed: 2025-02-06.
- Rafael Pass. Parallel repetition of zero-knowledge proofs and the possibility of basing cryptography on np-hardness. In *21st Annual IEEE Conference on Computational Complexity (CCC’06)*, pages 13–pp. IEEE, 2006.
- Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*, page 129. Pearson, 2016.
- Zedong Sun, Chunxiang Gu, and Yonghui Zheng. A review of sieve algorithms in solving the shortest lattice vector problem. *IEEE Access*, 8:190475–190486, 2020.
- United States Department of Commerce. Federal register/notices. Notice and request for nominations for candidate postquantum algorithms. 244, National Institute of Standards and Technology, December 2016. url=<https://www.federalregister.gov/d/2016-30615>.
- Ziyu Zhao, Jintai Ding, and Bo-Yin Yang. Sieving with streaming memory access. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):362–384, 2025.

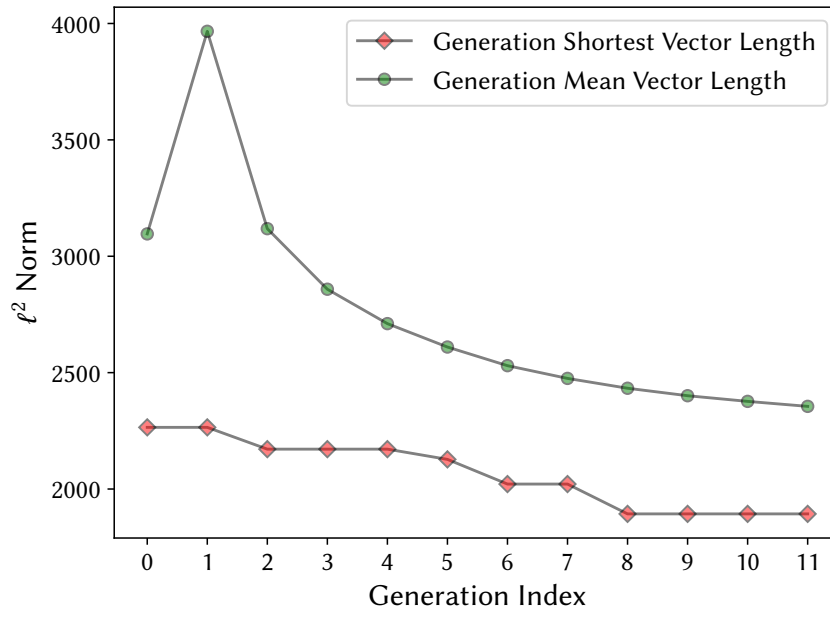
# Appendix A

## FIGURES

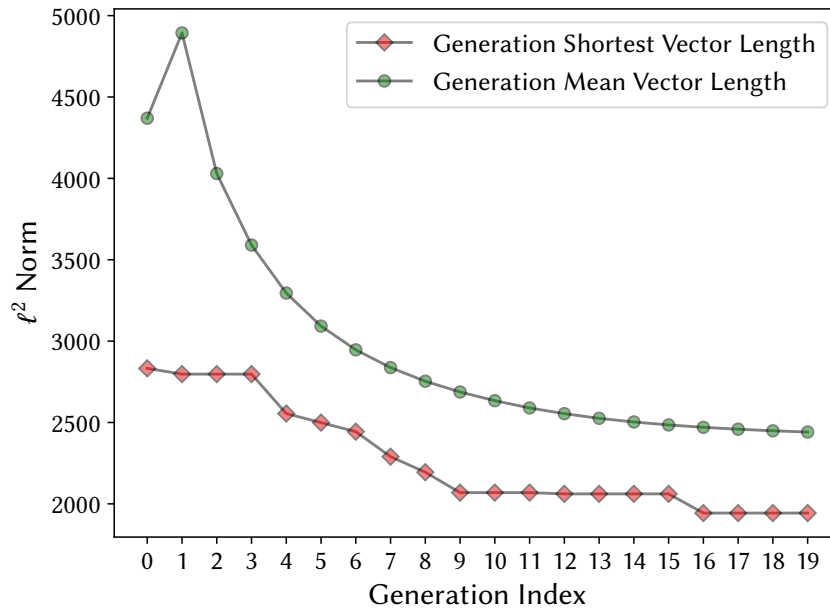


(A) Smallest norm found: 1702.464 Lattice dim.: 40  
Initial Population size: 1000 Sec. taken: 0.06s

**FIGURE A.1**—SVP Challenge Lattice ( $d = 40$ ) Reduction Curve

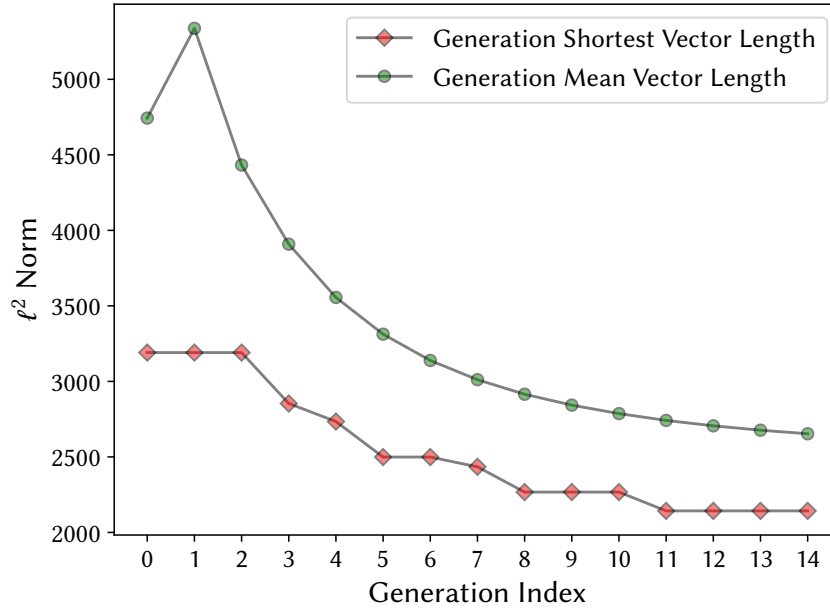


(B) Smallest norm found: 1893.170 Lattice dim.: 50  
Initial Population size: 1250 Sec. taken: 0.10s

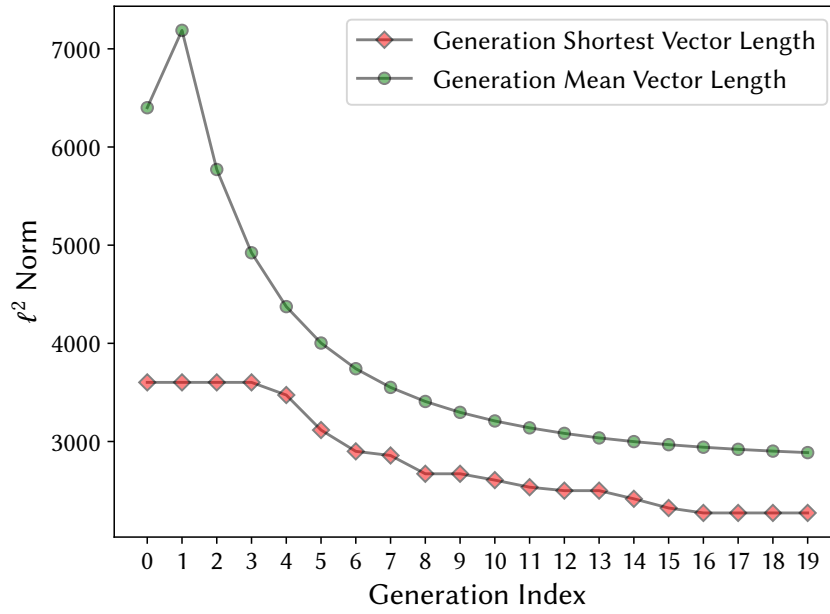


(C) Smallest norm found: 1943.401 Lattice dim.: 60  
Initial Population size: 9900 Sec. taken: 6.97s

FIGURE A.1—SVP Challenge Lattices' ( $d \in \{50, 60\}$ ) Reduction Curve

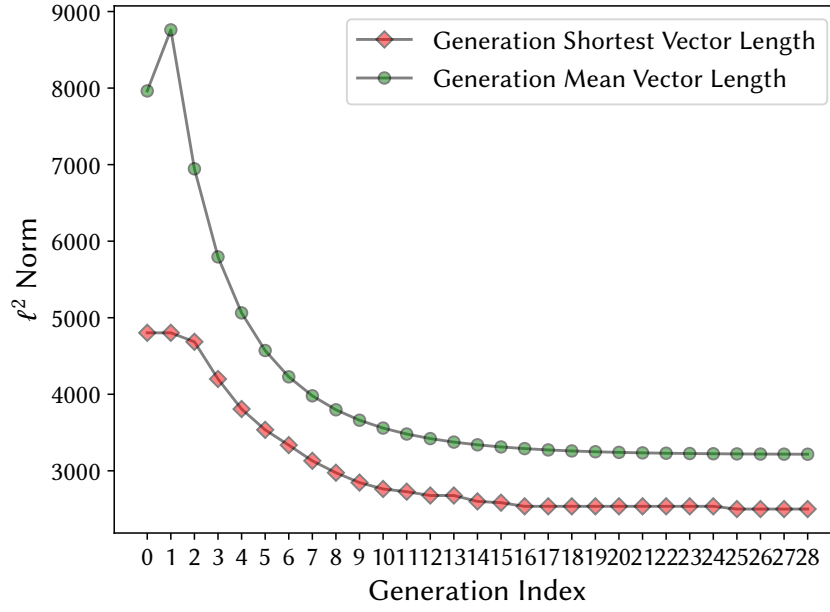


(D) Smallest norm found: 2142.603    Lattice dim.: 70  
Initial Population size: 49980    Sec. taken: 146.43s

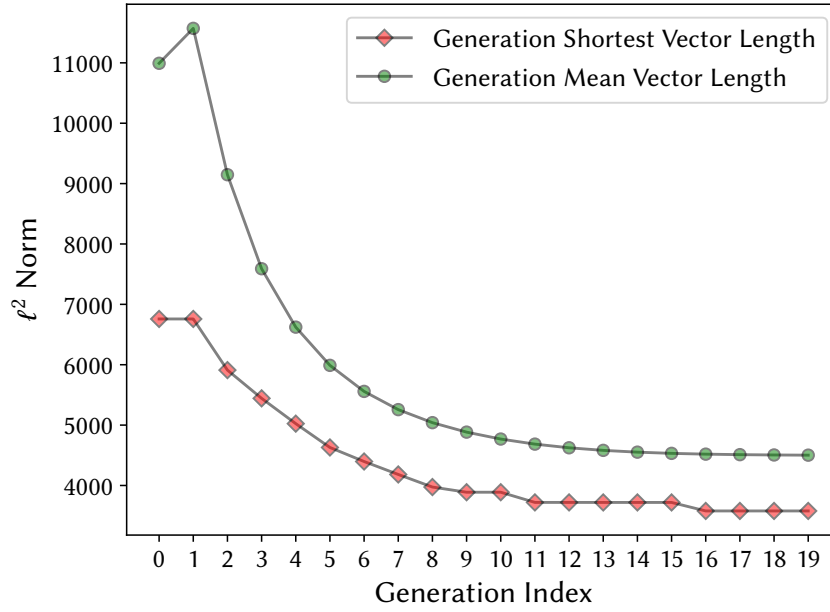


(E) Smallest norm found: 2272.188    Lattice dim.: 80  
Initial Population size: 240000    Sec. taken: 7637.02s

**FIGURE A.1**—SVP Challenge Lattices' ( $d \in \{70, 80\}$ ) Reduction Curve

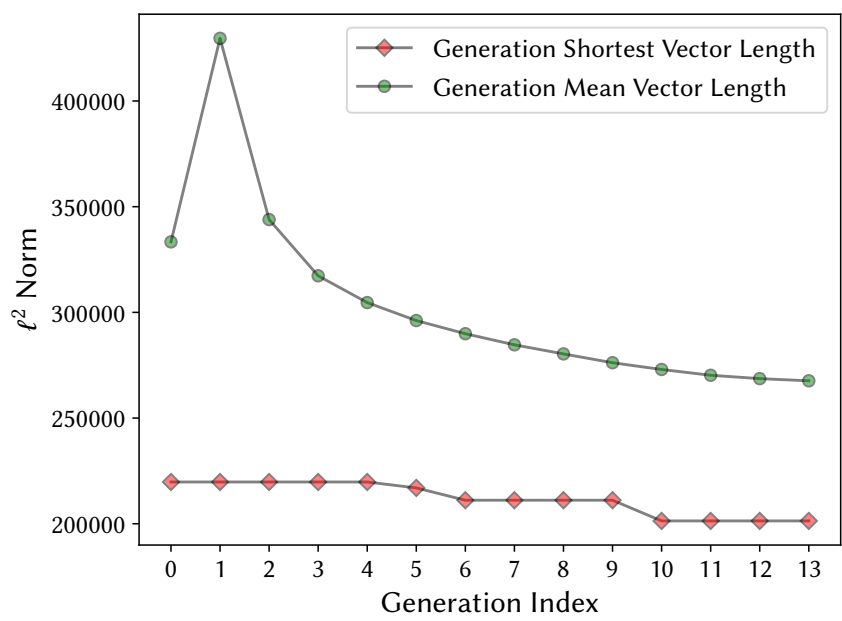


(F) Smallest norm found: 2500.403    Lattice dim.: 90  
 Initial Population size: 1638000    Sec. taken: 74319.49s



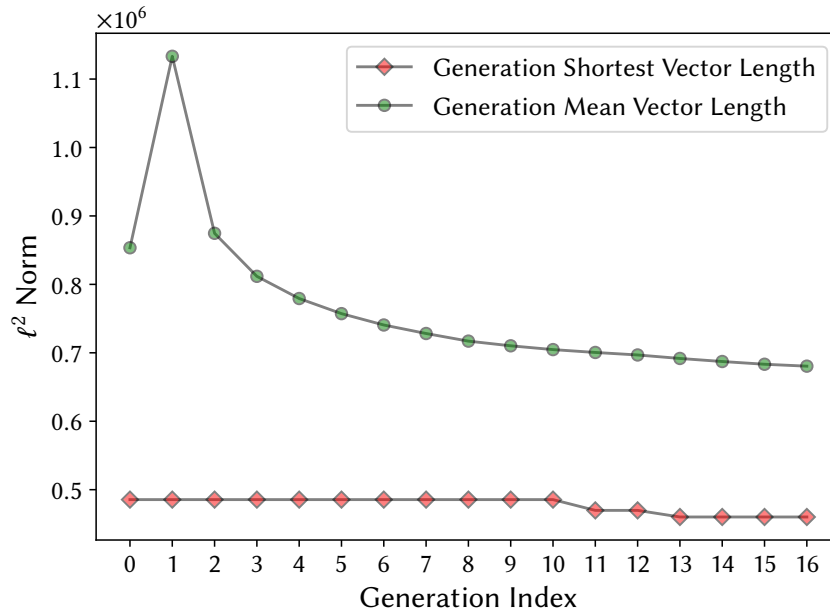
(G) Smallest norm found: 3578.229    Lattice dim.: 100  
 Initial Population size: 1500000    Sec. taken: 42785.54s

**FIGURE A.1**—SVP Challenge Lattices' ( $d \in \{90, 100\}$ ) Reduction Curve

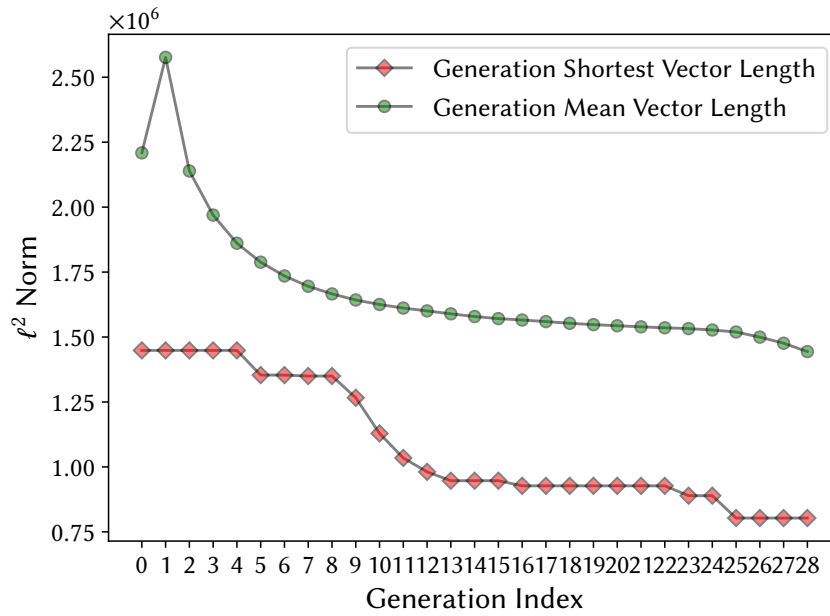


(A) Smallest norm found: 201343.887 Lattice dim.: 40  
Initial Population size: 800 Sec. taken: 0.05s

**FIGURE A.2**—SVP Random Integral Lattice ( $d = 40$ ) Reduction Curve

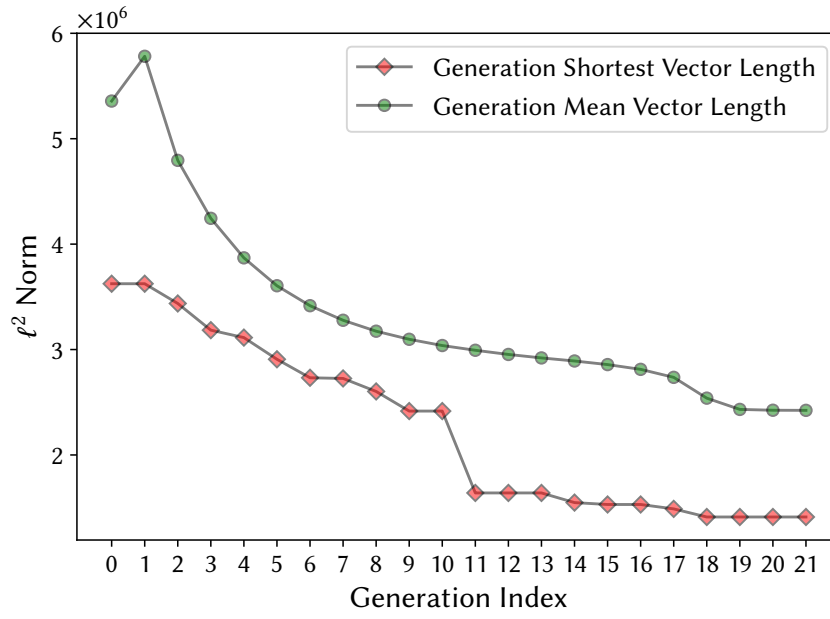


(B) Smallest norm found: 460101.228 Lattice dim.: 50  
Initial Population size: 1000 Sec. taken: 0.10s

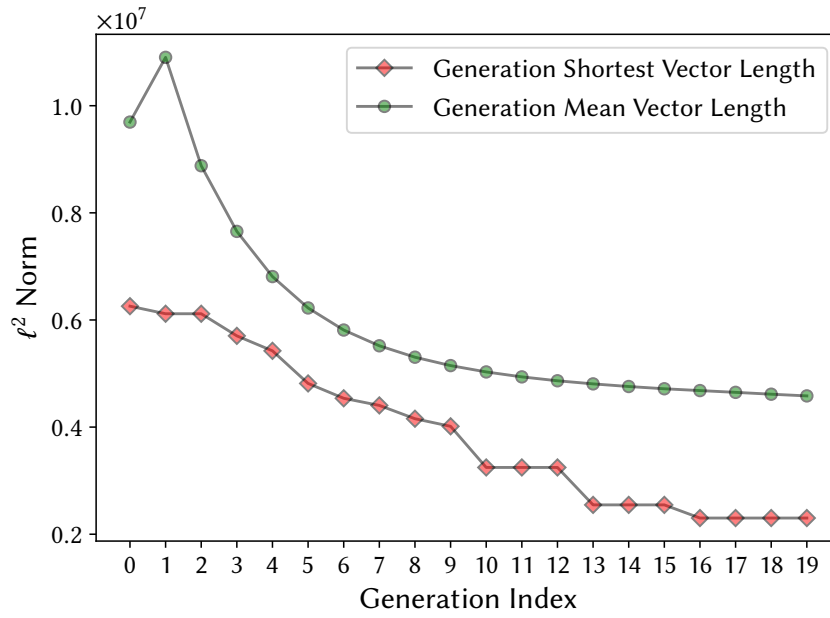


(C) Smallest norm found: 803170.352 Lattice dim.: 60  
Initial Population size: 3000 Sec. taken: 1.20s

FIGURE A.2—SVP Random Integral Lattices' ( $d \in \{50, 60\}$ ) Reduction Curve

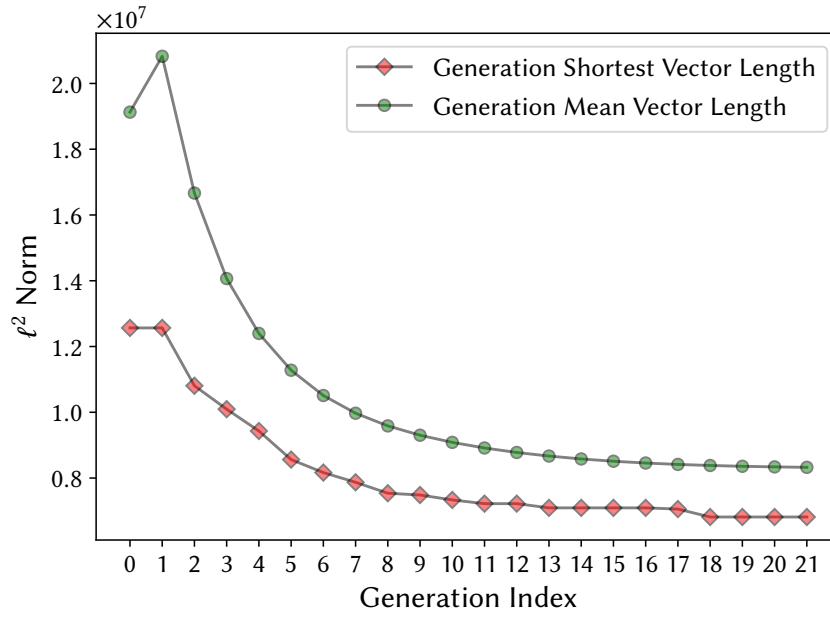


(D) Smallest norm found: 1411636.198    Lattice dim.: 70  
Initial Population size: 28000    Sec. taken: 73.45s

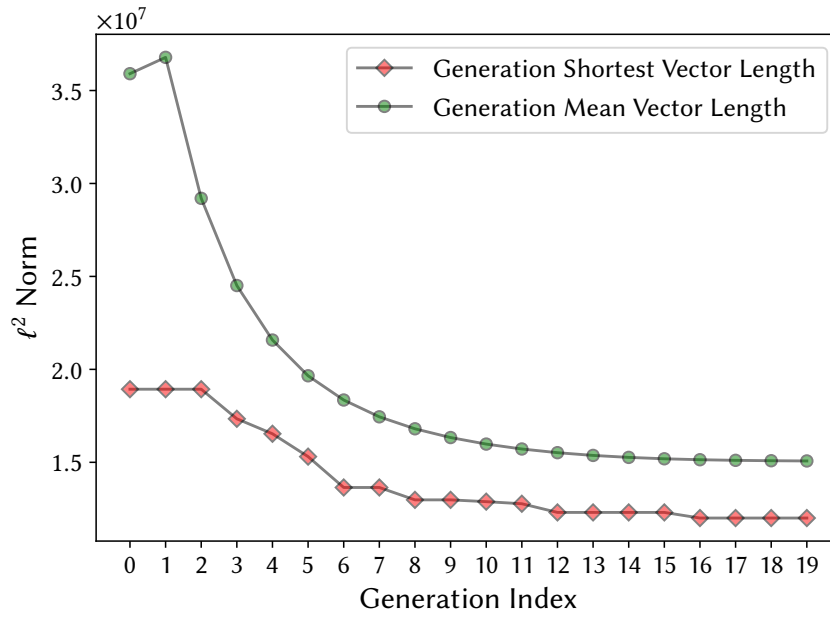


(E) Smallest norm found: 2302176.051    Lattice dim.: 80  
Initial Population size: 160000    Sec. taken: 4221.31s

FIGURE A.2—SVP Random Integral Lattices' ( $d \in \{70, 80\}$ ) Reduction Curve

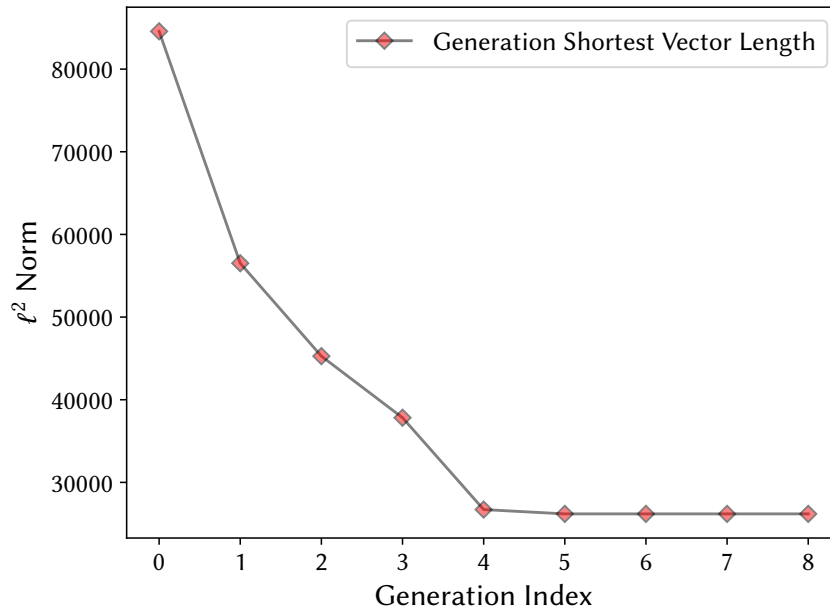


(F) Smallest norm found: 6815686.916 Lattice dim.: 90  
Initial Population size: 540000 Sec. taken: 36314.30s

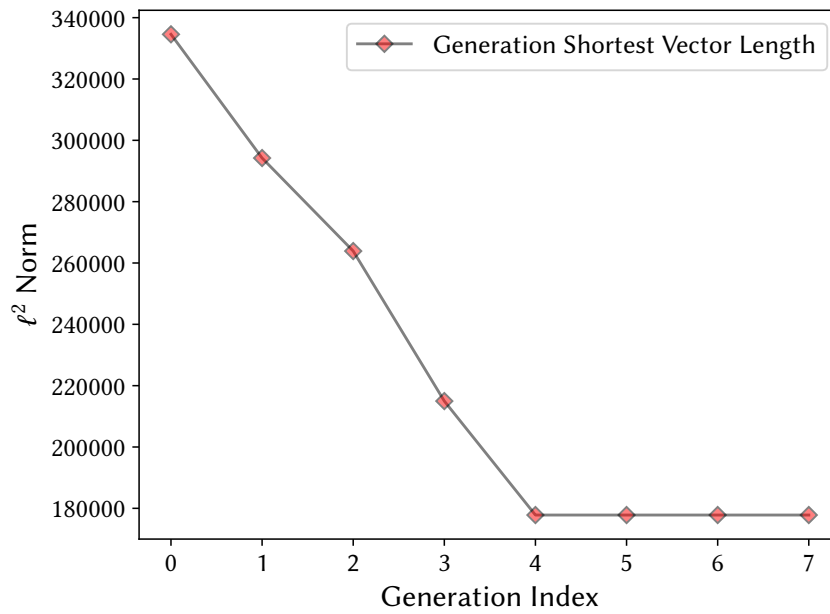


(G) Smallest norm: 12002723.155 Lattice dim.: 100  
Initial Pop. size: 1000000 Sec. taken: 123570.32s

**FIGURE A.2**—SVP Random Integral Lattices' ( $d \in \{90, 100\}$ ) Reduction Curve

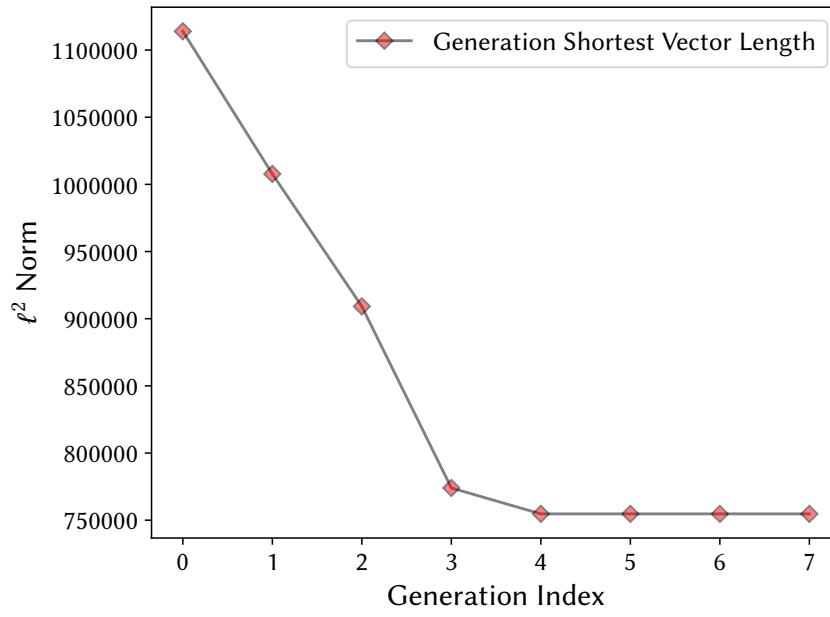


(A) Smallest norm found: 26188.633    Lattice dim.: 20  
Initial Population size: 2000    Sec. taken: 0.17s

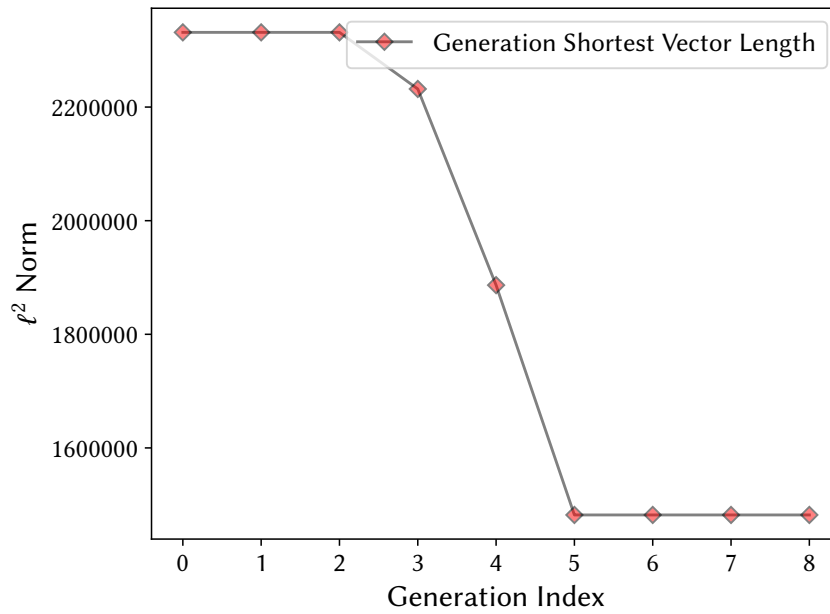


(B) Smallest norm found: 177814.207    Lattice dim.: 30  
Initial Population size: 3000    Sec. taken: 0.29s

**FIGURE A.3**—SVP Module Lattices' ( $d \in \{20, 30\}$ ) Reduction Curve



(c) Smallest norm found: 754691.69    Lattice dim.: 40  
 Initial Population size: 4000    Sec. taken: 0.54s



(d) Smallest norm found: 1482020.579    Lattice dim.: 50  
 Initial Population size: 5000    Sec. taken: 1.01s

**FIGURE A.3**—SVP Module Lattices' ( $d \in \{40, 50\}$ ) Reduction Curve

## CODE LISTINGS

**LISTING B.1**—A Python implementation for Equation 3.1 and Examples 1 and 2.

---

```

1 import numpy as np
2
3 norm = np.linalg.norm
4 B = np.array([[1, 40], [30, 5]], dtype=np.longlong)
5 C = np.array([[15, 7], [2, 1]], dtype=np.longlong)
6 bad = B @ C
7 b1, b2 = bad[:, 0], bad[:, 1]
8
9 def G(a, b, f):
10     if norm(a) > norm(b):
11         return G(b, a, f)
12     print(f'a={a}, b={b}, f(a,b)={f(a,b)}')
13     if f(a, b) == 0:
14         return (a, b)
15     return G(a, b - f(a, b) * a, f)
16
17 def quotient(a, b):
18     return b // a if a > 0 else 0
19
20 def mu(a, b):
21     rounded = np.rint((a.T @ b) / (a.T @ a))
22     return rounded.astype(np.dtype(a[0]))
23
24 G(748, 2024, quotient)
25 G(b1, b2, mu)

```

---

**LISTING B.2**—A Python implementation for Vector projection  $\text{proj}_u(v)$  and Gram–Schmidt orthogonalisation.

---

```
1 import numpy as np
2
3 def proj(u, v): # projecting v onto u
4     mu = (v @ u.T) / (u @ u.T)
5     return mu * u, mu
6
7 def gram_schmidt(B):
8     U = np.array(B, dtype=B.dtype)
9     Mu = np.zeros(shape=B.shape, dtype=B.dtype)
10    for i in range(1, B.shape[1]):
11        for j in range(i):
12            projection, Mu[i][j] = proj(U[:, j], B[:, i])
13            U[:, i] -= projection
14    return U, Mu
```

---

**LISTING B.3**—A Python implementation of the LLL algorithm given as Algorithm 2. We also use it to demonstrate the attack given in Example 4. The Gram–Schmidt function is from Listing B.2.

---

```

1 def lovasz_condition(G, Mu, k, delta):
2     c = delta - Mu[k][k - 1]**2
3     return G[:, k] @ G[:, k].T >= c * (G[:, k - 1] @ G[:, k - 1].T)
4
5 def lll(bad_basis, delta=0.75):
6     B = np.array(bad_basis)
7     G, Mu = gram_schmidt(B) # G are the B*
8     k, n = 1, B.shape[1] - 1
9     while k <= n:
10         for j in range(k - 1, -1, -1):
11             if abs(Mu[k][j]) > 0.5: # size condition not satisfied
12                 B[:, k] -= round(Mu[k][j]) * B[:, j]
13                 G, Mu = gram_schmidt(B)
14             if lovasz_condition(G, Mu, k, delta):
15                 k = k + 1
16             else:
17                 B[:, [k, k - 1]] = B[:, [k - 1, k]] # swap
18                 G, Mu = gram_schmidt(B)
19                 k = max(k - 1, 1)
20     return B
21
22 M = np.array([
23     67108861, 134217725, 268435453, 536870909, 1073741821,
24     2147483645, 4294967293, 8589934589, 17179869181, 34359738365,
25     68719476733, 137438953469, 274877906941, 549755813885,
26     1099511627773, 2199023255549, 4398046511101, 8796093022205,
27     17592186044413, 35184372088829, 70368744177661, 140737488355325,
28     281474976710653, 562949953421309, 1125899906842621
29 ])
30 n = M.shape[0]
31 S = 2002491457667039
32 B = 2*np.identity(n + 1)
33 B[:, -1], B[-1, :-1], B[-1, -1] = 1, M, S
34 B = lll(B)
35 t = B[:-1, 14]
36 print(np.sqrt(t.T @ t))
37 x = (t - 1) % 3
38 encodings = np.hstack((np.zeros((5, 3)), x.reshape((-1, 5))))
39 encodings = np.packbits(encodings.astype(np.uint8))
40 print(''.join([chr(e + ord('A') - 1) for e in encodings]))

```

---

**LISTING B.4**—A slow but illustrative Python implementation of Algorithm 5.

---

```
1 import numpy as np
2
3 RNG = np.random.default_rng(786)
4 PATH = '.././data/basis/challenge/col40_1740.csv'
5 B = np.loadtxt(PATH, delimiter=',')
6 d, n, rho = B.shape[1], 1000, 0.01
7 inflated = np.int64(n**1.5)
8
9 def elite(P, R):
10     PR = np.hstack((P, R))
11     norm = np.linalg.norm(PR, axis=0)
12     norm, idx = np.unique(norm, return_index=True, sorted=True)
13     norm, idx = (norm[1:], idx[1:]) if norm[0] == 0 else (norm, idx)
14     PR, norm = PR[:, idx[:n]], norm[:n]
15     return PR, norm[0]
16
17 def select(P):
18     for u_idx in range(P.shape[1] - 2):
19         for v_idx in range(u_idx + 1, P.shape[1]):
20             yield (P[:, u_idx], P[:, v_idx])
21
22 def cross(u, v): # the integral lattice part
23     return v - (np.round(np.dot(u, v) / np.dot(u, u)) * u)
24
25 C = RNG.binomial(1, rho, size=(d, n - d))
26 P = np.hstack((B @ C, B)) # include the orig. B to the init. pop.
27 R = np.zeros((d, inflated), dtype=P.dtype)
28 while True:
29     P, best = elite(P, R)
30     print(f'Length of the shorst vector found: {best:7.2f}')
31     if best < 1703: break
32     j, R = 0, np.zeros((d, inflated), dtype=P.dtype)
33     for u, v in select(P):
34         if j >= inflated: break
35         t = cross(u, v)
36         ul, vl, tl = np.dot(u, u), np.dot(v, v), np.dot(t, t)
37         in_P = np.any(np.all(P == t[:, None], axis=0))
38         in_R = np.any(np.all(R == t[:, None], axis=0))
39         if not in_P and not in_R and tl != 0 and (tl < ul or tl < vl):
40             R[:, j] = t
41             j += 1
42 print(f'Shortest Vector Found:\n {P[:, 0]}')
```

---

## Appendix C

# CONSOLE OUTPUTS

**LISTING C.1**—Console output for the SVP Challenge Lattice of dimension 40.

---

```
1 Size: 1000
2 Inflated Size: 31622
3
4 0 1912.362, 2482.952 in 0.00453s
5 1 1912.362, 3346.942 in 0.00139s Under population by 132960.00%!
6 2 1912.362, 2589.155 in 0.02070s
7 3 1859.716, 2376.010 in 0.01252s
8 4 1850.081, 2257.555 in 0.00753s
9 5 1815.074, 2171.883 in 0.00638s
10 6 1745.161, 2105.868 in 0.00598s
11 7 1702.464, 2062.122 in 0.00573s
12 8 1702.464, 2029.197 in 0.00522s
13 9 1702.464, 2005.307 in 0.00537s
14 10 1702.464, 1991.773 in 0.00498s
15 11 1702.464, 1985.103 in 0.00546s
16 12 1702.464, 1982.738 in 0.00501s
17 13 1702.464, 1981.683 in 0.00508s
18 14 1702.464, 1981.048 in 0.00520s
19 15 1702.464, 1980.917 in 0.00498s
20 16 1702.464, 1980.839 in 0.00489s
21 17 1702.464, 1980.702 in 0.00517s
22
23 Total time: 0.120924s
24
25 [-398 -305 -268 125 96 214 284 -108 37 -2 402 228 -243 -33
26 -76 -265 -3 558 323 552 -419 -408 217 2 440 375 -153 108
27 79 80 -299 -81 385 -80 -53 -294 -170 380 164 172]
```

---

**LISTING C.2**—Console output for the SVP Challenge Lattice of dimension 50.

---

```
1 Size: 1250
2 Inflated Size: 44194
3
4 0 2264.923, 3096.038 in 0.00205s
5 1 2264.923, 3966.378 in 0.00198s
6 2 2171.312, 3118.497 in 0.03286s
7 3 2171.312, 2858.358 in 0.01430s
8 4 2171.312, 2710.569 in 0.01075s
9 5 2127.365, 2610.033 in 0.00938s
10 6 2021.300, 2529.944 in 0.00896s
11 7 2021.300, 2475.239 in 0.00864s
12 8 1893.170, 2432.966 in 0.00837s
13 9 1893.170, 2400.848 in 0.00829s
14 10 1893.170, 2376.533 in 0.00812s
15 11 1893.170, 2355.122 in 0.00837s
16 12 1893.170, 2339.880 in 0.00792s
17 13 1893.170, 2330.885 in 0.00902s
18 14 1893.170, 2324.235 in 0.00824s
19 15 1893.170, 2318.733 in 0.00892s
20 16 1893.170, 2315.077 in 0.00783s
21 17 1893.170, 2312.895 in 0.00810s
22 18 1893.170, 2311.245 in 0.00859s
23 19 1893.170, 2309.764 in 0.00798s
24 20 1893.170, 2308.277 in 0.00861s
25 21 1893.170, 2306.687 in 0.00820s
26 22 1893.170, 2304.973 in 0.00835s
27 23 1893.170, 2304.221 in 0.00794s
28 24 1893.170, 2303.852 in 0.00860s
29 25 1893.170, 2303.526 in 0.00831s
30 26 1893.170, 2303.190 in 0.00805s
31
32 Total time: 0.254885s
33
34 [ 13 124 146 -277 107 180 -673 311 167 -47 -200 -395 -167 25
35 136 392 -117 165 -147 515 -185 -637 -343 -8 -247 -44 220 146
36 -52 -135 347 369 332 102 -469 285 -1 -167 -397 -84 97 138
37 135 -218 -567 -141 -72 -21 -312 41]
```

---

**LISTING C.3**—Console output for the SVP Challenge Lattice of dimension 60.

---

```
1 Size: 9900
2 Inflated Size: 985037
3
4 0 2832.415, 4369.708 in 0.01192s
5 1 2797.254, 4893.890 in 0.04210s
6 2 2797.254, 4029.609 in 0.68567s
7 3 2797.254, 3590.169 in 0.49701s
8 4 2554.839, 3295.269 in 0.45538s
9 5 2499.194, 3092.210 in 0.48696s
10 6 2443.673, 2946.383 in 0.44536s
11 7 2289.778, 2837.202 in 0.45057s
12 8 2194.555, 2753.496 in 0.43617s
13 9 2069.031, 2687.322 in 0.43801s
14 10 2069.031, 2633.935 in 0.43074s
15 11 2069.031, 2589.439 in 0.42713s
16 12 2061.371, 2554.202 in 0.42440s
17 13 2061.371, 2525.505 in 0.43040s
18 14 2061.371, 2503.208 in 0.45826s
19 15 2061.371, 2484.902 in 0.42335s
20 16 1943.401, 2470.261 in 0.42200s
21 17 1943.401, 2458.589 in 0.44500s
22 18 1943.401, 2448.836 in 0.44036s
23 19 1943.401, 2441.363 in 0.43306s
24 20 1943.401, 2435.557 in 0.43641s
25 21 1943.401, 2431.062 in 0.43768s
26 22 1943.401, 2427.670 in 0.43204s
27 23 1943.401, 2425.110 in 0.43980s
28 24 1943.401, 2423.070 in 0.43830s
29 25 1943.401, 2421.451 in 0.43560s
30 26 1943.401, 2420.085 in 0.42699s
31 27 1943.401, 2419.229 in 0.43449s
32 28 1943.401, 2418.438 in 0.44522s
33 29 1943.401, 2417.773 in 0.42776s
34 30 1943.401, 2417.360 in 0.42558s
35
36 Total time: 13.497959s
37
38 [-348 145 -247 -98 228 -310 -199 -206 -72 -46 -134 -581 -62 -199
39 262 -129 -737 324 -129 11 -377 185 -297 26 -273 -8 -584 342
40 -273 -178 7 28 -102 -121 -322 -307 120 158 -270 -197 91 -96
41 72 79 92 -33 505 107 -291 166 -128 -214 -172 -230 191 -77
42 -342 -201 4 370]
```

---

**LISTING C.4**—Console output for the SVP Challenge Lattice of dimension 70.

---

```
1 Size: 49980
2 Inflated Size: 11173632
3
4 0 3190.723, 4743.050 in 0.05627s
5 1 3190.723, 5337.673 in 0.37334s
6 2 3190.723, 4432.355 in 11.01042s
7 3 2853.678, 3908.552 in 14.56825s
8 4 2734.249, 3555.667 in 14.18696s
9 5 2499.099, 3312.634 in 14.26732s
10 6 2499.099, 3138.656 in 14.55384s
11 7 2434.308, 3011.202 in 16.77271s
12 8 2267.165, 2914.950 in 15.03149s
13 9 2267.165, 2842.833 in 15.24516s
14 10 2267.165, 2786.755 in 15.05966s
15 11 2142.603, 2741.590 in 15.30548s
16 12 2142.603, 2705.863 in 16.44436s
17 13 2142.603, 2676.801 in 17.72142s
18 14 2142.603, 2653.369 in 18.55664s
19 15 2142.603, 2634.869 in 18.90239s
20 16 2142.603, 2620.467 in 18.88113s
21 17 2142.603, 2609.068 in 18.24417s
22 18 2142.603, 2600.314 in 17.90482s
23 19 2142.603, 2593.749 in 17.78622s
24 20 2142.603, 2588.513 in 18.30107s
25 21 2142.603, 2584.447 in 18.32186s
26 22 2142.603, 2581.491 in 18.03650s
27 23 2142.603, 2579.297 in 19.33191s
28 24 2142.603, 2577.715 in 18.63557s
29 25 2142.603, 2576.607 in 17.70138s
30 26 2142.603, 2575.739 in 21.19979s
31 27 2142.603, 2575.091 in 18.04346s
32 28 2142.603, 2574.672 in 17.49826s
33 29 2142.603, 2574.364 in 16.96005s
34
35 Total time: 492.954741s
36
37 [ 196  448  201 -113  515 -216   40   47  -72  145 -191   80 -479 -391
38    55  261  -61  160   69   90  -67 -465  -51   75   10 -388 -379 -753
39  -224  334   18  149  188  374 -105  252 -708  -88 -365  232  307  -32
40   145   66   60  356  190  -61  161  377  -98  139 -311  169 -302   68
41    6   69 -294  199 -227 -225   95  256   44   54  186 -134   91  340]
```

---

**LISTING C.5**—Console output for the SVP Challenge Lattice of dimension 80.

---

```
1 Size: 240000, Inflated Size: 117575507
2   0 3602.245, 6400.256 in 0.20419s
3   1 3602.245, 7188.106 in 5.30953s
4   2 3602.245, 5770.866 in 213.58431s
5   3 3602.245, 4922.667 in 499.28085s
6   4 3473.367, 4373.929 in 506.60903s
7   5 3116.654, 4002.890 in 526.31994s
8   6 2899.646, 3741.577 in 536.73854s
9   7 2857.384, 3550.981 in 545.64130s
10  8 2671.065, 3407.710 in 544.82597s
11  9 2671.065, 3296.663 in 548.27744s
12 10 2606.762, 3209.207 in 545.51023s
13 11 2534.403, 3139.199 in 527.64378s
14 12 2499.262, 3082.623 in 521.89239s
15 13 2499.262, 3036.484 in 530.23218s
16 14 2416.088, 2998.744 in 527.82540s
17 15 2322.553, 2967.647 in 527.60904s
18 16 2272.188, 2941.683 in 529.51876s
19 17 2272.188, 2920.040 in 528.12346s
20 18 2272.188, 2901.947 in 528.46708s
21 19 2272.188, 2886.794 in 530.99946s
22 20 2272.188, 2874.031 in 531.07922s
23 21 2272.188, 2863.364 in 529.84323s
24 22 2272.188, 2854.357 in 531.67754s
25 23 2272.188, 2846.723 in 534.24248s
26 24 2272.188, 2840.266 in 535.98455s
27 25 2272.188, 2834.759 in 533.14570s
28 26 2272.188, 2830.014 in 534.67766s
29 27 2272.188, 2825.955 in 533.44082s
30 28 2272.188, 2822.455 in 533.08935s
31 29 2272.188, 2819.494 in 534.53782s
32 30 2272.188, 2816.988 in 537.36440s
33 31 2272.188, 2814.826 in 535.22696s
34 32 2272.188, 2812.918 in 534.82659s
35 33 2272.188, 2811.222 in 537.18642s
36 34 2272.188, 2809.759 in 535.97022s
37 35 2272.188, 2808.512 in 539.84003s
38 36 2272.188, 2807.405 in 640.95644s
39 37 2272.188, 2806.421 in 598.79527s
40 38 2272.188, 2805.552 in 566.19784s
41 39 2272.188, 2804.779 in 556.53758s
42 40 2272.188, 2804.122 in 548.09959s
43 Total time: 21232.685969s
44 [-188 -157 -157 214 267 -180 112 36 609 104 -272 -70 110 400
45 312 -202 177 227 224 385 -313 195 -135 -193 -191 -281 314 -505
46 -295 186 -140 -455 14 -301 -403 478 242 -36 108 -9 308 316
47 348 -177 192 -53 439 -364 26 154 -61 -355 -45 -140 10 330
48 -52 19 -175 233 -65 -103 -36 332 -229 -376 373 17 -118 192
49 -356 -198 -276 -208 38 -3 363 -371 -43 384]
```

---

**LISTING C.6**—Console output for the SVP Challenge Lattice of dimension 90.

---

```
1 Size: 1638000
2 Inflated Size: 2096384046
3
4 0 4802.754, 7962.551 in 2.04777s
5 1 4802.754, 8760.467 in 526.55140s
6 2 4685.558, 6945.282 in 3802.11324s
7 3 4199.884, 5794.646 in 3511.01676s
8 4 3806.983, 5063.222 in 3349.83799s
9 5 3535.221, 4571.333 in 3085.32839s
10 6 3335.941, 4227.734 in 3205.42542s
11 7 3130.428, 3979.921 in 2614.28067s
12 8 2975.135, 3797.239 in 3366.56690s
13 9 2846.157, 3660.979 in 2692.03135s
14 10 2762.938, 3558.430 in 3324.91824s
15 11 2727.309, 3480.635 in 2673.35937s
16 12 2677.483, 3421.207 in 3253.26347s
17 13 2677.483, 3375.224 in 3023.34232s
18 14 2600.477, 3339.585 in 3324.71870s
19 15 2583.521, 3311.866 in 2636.15401s
20 16 2536.109, 3290.087 in 3327.30191s
21 17 2536.109, 3272.874 in 2698.91925s
22 18 2536.109, 3259.433 in 3307.31688s
23 19 2536.109, 3248.961 in 2779.12259s
24 20 2536.109, 3240.879 in 3289.23681s
25 21 2536.109, 3234.502 in 2649.14628s
26 22 2536.109, 3229.610 in 3284.50986s
27 23 2536.109, 3225.872 in 2691.78710s
28 24 2536.109, 3222.986 in 3238.52975s
29 25 2500.403, 3220.759 in 2662.65983s
30 26 2500.403, 3219.030 in 3282.88799s
31 27 2500.403, 3217.720 in 2659.38487s
32 28 2500.403, 3216.694 in 3229.82086s
33 29 2500.403, 3215.892 in 2773.47250s
34 30 2500.403, 3215.264 in 3308.98351s
35 31 2500.403, 3214.792 in 2728.66728s
36 32 2500.403, 3214.429 in 3251.51758s
37
38 Total time: 98323.792400s
39
40 [ 160 181 132 -193 408 -199 -329 88 -380 -171 -369 137 -161 602
41 461 -171 -190 93 -417 320 74 -207 238 174 337 224 338 -160
42 45 128 32 -70 453 -276 8 -232 144 -190 -426 -86 372 265
43 -21 -434 -270 272 186 671 -77 396 142 -74 -307 326 -469 -164
44 -587 -164 -31 -114 -158 -339 -19 -170 -84 -36 -306 453 301 -313
45 -200 -100 -19 111 142 139 377 -155 211 294 -149 45 -214 -315
46 -243 -230 199 156 42 178]
```

---

---

**LISTING C.7**—Console output for the SVP Challenge Lattice of dimension 100.

---

```

1 Size: 1500000
2 Inflated Size: 1837117307
3
4 0 6758.062, 10992.544 in 2.10386s
5 1 6758.062, 11571.024 in 666.19750s
6 2 5910.955, 9147.078 in 3316.01853s
7 3 5443.246, 7590.006 in 3046.18378s
8 4 5024.089, 6622.353 in 2976.13735s
9 5 4629.458, 5989.541 in 3157.17706s
10 6 4397.103, 5558.231 in 2949.95894s
11 7 4182.527, 5256.305 in 2555.45891s
12 8 3973.899, 5040.451 in 2831.08106s
13 9 3888.233, 4883.698 in 2300.12671s
14 10 3888.233, 4768.940 in 2837.67628s
15 11 3719.575, 4685.027 in 2294.87823s
16 12 3719.575, 4624.391 in 3003.32728s
17 13 3719.063, 4581.565 in 2459.88658s
18 14 3719.063, 4551.908 in 2991.92297s
19 15 3719.063, 4531.885 in 2358.78437s
20 16 3578.229, 4518.881 in 3038.61961s
21 17 3578.229, 4510.505 in 2456.58231s
22 18 3578.229, 4505.271 in 2953.51642s
23 19 3578.229, 4502.075 in 2483.83353s
24 20 3578.229, 4500.166 in 2998.31692s
25 21 3578.229, 4499.055 in 2488.35726s
26 22 3578.229, 4498.420 in 3031.62903s
27 23 3578.229, 4498.062 in 2425.65462s
28 24 3578.229, 4497.862 in 2983.53163s
29 25 3578.229, 4497.747 in 2449.95434s
30
31 Total time: 70129.963139s
32
33 [-201 50 -167 73 113 -208 127 898 -314 -193 -412 -309 -287 -545
34 33 -268 672 81 -162 300 -467 447 -386 -51 -162 -580 405 183
35 5 260 173 -544 0 233 -364 -249 -980 311 -771 154 -38 -345
36 23 641 -184 266 -23 211 -42 80 24 265 -204 413 -40 -15
37 480 586 -5 67 -8 -339 -265 167 60 591 -9 -673 -618 25
38 -197 -151 -605 -107 -23 -397 254 360 -308 -485 -339 -142 -566 547
39 -60 -737 465 196 -484 -712 522 348 -272 -297 118 -61 100 49
40 72 -100]
```

---

**LISTING C.8**—Console output for the Random Integral Lattice of dimension 40.

---

```
1 Size: 800
2 Inflated Size: 22627
3
4 0 219760.103, 333339.306 in 0.00072s
5 1 219760.103, 429639.342 in 0.00100s Under population by 120600.00%!
6 2 219760.103, 343903.400 in 0.01210s
7 3 219760.103, 317301.842 in 0.00635s
8 4 219760.103, 304634.933 in 0.00404s
9 5 216955.079, 296121.475 in 0.00362s
10 6 211134.977, 289910.387 in 0.00356s
11 7 211134.977, 284653.564 in 0.00344s
12 8 211134.977, 280347.604 in 0.00375s
13 9 211134.977, 276182.691 in 0.00378s
14 10 201343.887, 272988.078 in 0.00405s
15 11 201343.887, 270248.346 in 0.00359s
16 12 201343.887, 268656.012 in 0.00402s
17 13 201343.887, 267625.986 in 0.00361s
18 14 201343.887, 267020.410 in 0.00390s
19 15 201343.887, 266502.412 in 0.00356s
20 16 201343.887, 266359.884 in 0.00348s
21 17 201343.887, 266198.477 in 0.00322s
22 18 201343.887, 266125.094 in 0.00361s
23 19 201343.887, 266104.750 in 0.00336s
24 20 201343.887, 266104.750 in 0.00325s
25 21 201343.887, 266104.750 in 0.00315s
26 22 201343.887, 266104.750 in 0.00359s
27 23 201343.887, 266104.750 in 0.00322s
28
29 Total time: 0.095247s
30
31 [-11822 15684 -45152 28359 -27520 -17793 -7284 4555 36085 42504
32 -10380 -34241 50333 -50490 34817 35421 2218 -26427 28771 14000
33 -11449 -6153 51584 -27531 15324 6438 22017 -9540 52162 870
34 -54037 46553 47122 177 -25072 39767 31685 10892 -62516 28061]
```

---

**LISTING C.9**—Console output for the Random Integral Lattice of dimension 50.

---

```

1 Size: 1000, Inflated Size: 31622, Total time: 0.261709s
2   0 485483.202, 853493.711 in 0.00140s
3   1 485483.202, 1133264.958 in 0.00129s Under population by 249250.00%!
4   2 485483.202, 874615.117 in 0.01994s
5   3 485483.202, 811751.319 in 0.01116s
6   4 485483.202, 779323.578 in 0.00783s
7   5 485483.202, 757232.244 in 0.00723s
8   6 485483.202, 740639.579 in 0.00600s
9   7 485483.202, 728210.372 in 0.00590s
10  8 485483.202, 717067.311 in 0.00550s
11  9 485483.202, 710227.792 in 0.00586s
12 10 485483.202, 704691.562 in 0.00601s
13 11 469748.187, 700509.335 in 0.00598s
14 12 469748.187, 696842.358 in 0.00570s
15 13 460101.228, 691739.807 in 0.00605s
16 14 460101.228, 687277.804 in 0.00561s
17 15 460101.228, 683281.979 in 0.00589s
18 16 460101.228, 680482.685 in 0.00652s
19 17 460101.228, 678807.862 in 0.00573s
20 18 460101.228, 677396.506 in 0.00587s
21 19 460101.228, 675920.360 in 0.00600s
22 20 460101.228, 675124.315 in 0.00563s
23 21 460101.228, 674477.382 in 0.00577s
24 22 460101.228, 673666.372 in 0.00583s
25 23 460101.228, 672183.918 in 0.00561s
26 24 460101.228, 670247.099 in 0.00560s
27 25 460101.228, 668945.171 in 0.00577s
28 26 460101.228, 667994.477 in 0.00631s
29 27 460101.228, 667274.119 in 0.00603s
30 28 460101.228, 666310.408 in 0.00591s
31 29 460101.228, 665595.844 in 0.00585s
32 30 460101.228, 665292.645 in 0.00604s
33 31 460101.228, 665173.959 in 0.00576s
34 32 460101.228, 665149.395 in 0.00594s
35 33 460101.228, 665122.114 in 0.00574s
36 34 460101.228, 665103.897 in 0.00591s
37 35 460101.228, 665083.111 in 0.00597s
38 36 460101.228, 665050.966 in 0.00651s
39 37 460101.228, 665050.966 in 0.00610s
40 38 460101.228, 665050.966 in 0.00612s
41 39 460101.228, 665050.966 in 0.00600s
42 40 460101.228, 665050.966 in 0.00591s
43 [ -52896    -669    46403  113395 -109266  124888   35437  -10170
44    74806  -72770   42229  -73679  -15965  -86685   52882   66612
45   -46868   11977   41922   70794   19758   11795   31391  -19277
46    10722   34511   41533   64652   -3765   58374   27577  -18058
47   -49817  -92685 -102376   74539   71469   -1033   83577   56414
48    24364  121086  102025   -8414  -76771   80097   86410 -104023
49   -98873   25943]
```

---

---

**LISTING C.10**—Console output for the Random Integral Lattice of dimension 60.

---

```
1 Size: 3000
2 Inflated Size: 164316
3
4 0 1448476.309, 2208878.997 in 0.00460s
5 1 1448476.309, 2576590.444 in 0.00795s
6 2 1448476.309, 2139298.002 in 0.10097s
7 3 1448476.309, 1969402.460 in 0.05689s
8 4 1448476.309, 1861075.744 in 0.04960s
9 5 1353427.215, 1788076.550 in 0.04501s
10 6 1353427.215, 1735058.587 in 0.04567s
11 7 1349780.408, 1695399.620 in 0.04595s
12 8 1349780.408, 1665927.806 in 0.04387s
13 9 1266180.151, 1642544.512 in 0.04301s
14 10 1128943.840, 1625217.172 in 0.04632s
15 11 1034397.955, 1611483.959 in 0.04731s
16 12 980970.589, 1600496.070 in 0.04350s
17 13 946953.826, 1589294.816 in 0.04594s
18 14 946953.826, 1578833.016 in 0.04592s
19 15 946953.826, 1570851.035 in 0.04443s
20 16 927141.170, 1565452.995 in 0.04364s
21 17 927141.170, 1559333.076 in 0.05074s
22 18 927141.170, 1552924.218 in 0.04726s
23 19 927141.170, 1547590.217 in 0.04962s
24 20 927141.170, 1543461.282 in 0.04707s
25 21 927141.170, 1539222.813 in 0.04875s
26 22 927141.170, 1535512.592 in 0.04879s
27 23 889231.390, 1532339.250 in 0.04901s
28 24 889231.390, 1527014.920 in 0.04835s
29 25 803170.352, 1519301.427 in 0.05005s
30 26 803170.352, 1499592.434 in 0.05051s
31 27 803170.352, 1476119.287 in 0.05169s
32 28 803170.352, 1443925.042 in 0.05014s
33 29 803170.352, 1408395.411 in 0.05415s
34 30 803170.352, 1386765.650 in 0.05719s
35 31 803170.352, 1357105.376 in 0.05710s
36 32 803170.352, 1319596.916 in 0.05991s
37 33 803170.352, 1298643.567 in 0.06209s
38 34 803170.352, 1291535.007 in 0.06115s
39 35 803170.352, 1286786.531 in 0.06014s
40 36 803170.352, 1282817.657 in 0.05786s
41 37 803170.352, 1279825.024 in 0.05903s
42 38 803170.352, 1278609.593 in 0.08358s
43 39 803170.352, 1278403.410 in 0.05919s
44 40 803170.352, 1278380.865 in 0.05710s
45 41 803170.352, 1278377.090 in 0.05688s
46 42 803170.352, 1278377.090 in 0.05722s
47 43 803170.352, 1278377.090 in 0.05767s
48 44 803170.352, 1278377.090 in 0.05733s
49 45 803170.352, 1278377.090 in 0.05697s
50
51 Total time: 2.424794s
52
53 [-158115 -117257 23745 188876 77229 153733 35322 -54239
54 67101 16721 74998 -68439 -119977 157571 101159 -64955
55 -67592 10777 110698 47284 10935 151288 -46966 90884
56 96746 13573 -56523 -95711 24070 40706 -27078 109800
57 25259 162078 141634 79334 198293 38828 -16140 -22370
58 -91366 140046 142157 -133950 -71764 -195013 -14497 -41983
59 206758 -96633 106786 -105683 34328 -108014 -130982 163790
60 79162 -70951 26350 -168436]
```

---

**LISTING C.11**—Console output for the Random Integral Lattice of dimension 70.

---

```
1 Size: 28000
2 Inflated Size: 4685296
3
4 0 3624313.352, 5357164.374 in 0.03990s
5 1 3624313.352, 5781789.822 in 0.20764s
6 2 3436989.726, 4794611.072 in 4.27418s
7 3 3183945.524, 4244689.928 in 4.25896s
8 4 3113344.207, 3869759.530 in 4.27081s
9 5 2907719.861, 3605198.866 in 4.20983s
10 6 2732166.422, 3415955.866 in 4.15198s
11 7 2725827.011, 3277696.307 in 4.21459s
12 8 2602472.035, 3174011.025 in 4.24249s
13 9 2416321.343, 3097259.206 in 4.28496s
14 10 2416321.343, 3038069.372 in 4.26918s
15 11 1639760.777, 2993132.429 in 4.29235s
16 12 1639760.777, 2953455.132 in 4.31546s
17 13 1639760.777, 2920472.578 in 4.33092s
18 14 1548087.723, 2891398.649 in 4.32737s
19 15 1530108.942, 2857294.611 in 4.34399s
20 16 1530108.942, 2812047.000 in 4.51221s
21 17 1488440.888, 2736221.267 in 4.38877s
22 18 1411636.198, 2538922.997 in 4.51314s
23 19 1411636.198, 2432132.104 in 4.77360s
24 20 1411636.198, 2424202.805 in 4.60956s
25 21 1411636.198, 2423190.823 in 4.61656s
26 22 1411636.198, 2422773.398 in 4.63874s
27 23 1411636.198, 2422574.176 in 4.64031s
28 24 1411636.198, 2422458.936 in 5.46922s
29 25 1411636.198, 2422396.227 in 5.21679s
30 26 1411636.198, 2422376.763 in 5.24844s
31 27 1411636.198, 2422373.431 in 5.06411s
32 28 1411636.198, 2422372.401 in 5.01949s
33 29 1411636.198, 2422372.401 in 4.99477s
34 30 1411636.198, 2422372.401 in 5.03338s
35 31 1411636.198, 2422372.401 in 5.06708s
36 32 1411636.198, 2422372.401 in 5.14832s
37
38 Total time: 148.409826s
39
40 [ 240970 -95604 -61899 -220998 202603 -214800 -101244 52209
41 19850 -16089 53058 218204 -90382 -12989 -42687 116804
42 118495 82037 -62119 231697 -128506 226601 -60507 -194619
43 204138 -96605 165845 -195629 -254510 292286 56970 113492
44 -63500 228393 -93541 289345 -20327 104909 325849 -146297
45 -92398 -113776 -322611 -341329 -97790 -177525 152554 47390
46 -266178 -34775 -162322 -27743 -24872 7346 -175824 283400
47 -326279 28153 -60227 3352 217938 -337252 -219792 -10995
48 -96224 102092 84054 192341 65107 176132]
```

---

**LISTING C.12**—Console output for the Random Integral Lattice of dimension 80.

```
1 Size: 160000
2 Inflated Size: 64000000
3
4 0 6257257.096, 9694439.684 in 0.15028s
5 1 6116148.924, 10902909.407 in 2.64776s
6 2 6116148.924, 8879369.829 in 119.08098s
7 3 5703282.571, 7653573.743 in 284.12619s
8 4 5423893.477, 6810133.166 in 245.76060s
9 5 4814974.772, 6224297.750 in 326.84768s
10 6 4537776.690, 5812025.980 in 303.67311s
11 7 4404788.693, 5517883.209 in 353.61353s
12 8 4156369.547, 5304968.597 in 252.93271s
13 9 4014426.359, 5148101.362 in 317.25944s
14 10 3246557.751, 5029824.180 in 326.86234s
15 11 3246557.751, 4937503.165 in 286.79181s
16 12 3246557.751, 4864752.398 in 272.87774s
17 13 2547741.585, 4806816.542 in 275.05248s
18 14 2547741.585, 4757661.546 in 290.52130s
19 15 2547741.585, 4715698.973 in 280.03832s
20 16 2302176.051, 4681748.832 in 283.07632s
21 17 2302176.051, 4648624.644 in 268.20530s
22 18 2302176.051, 4614228.960 in 254.14041s
23 19 2302176.051, 4582301.982 in 257.84574s
24 20 2302176.051, 4549535.483 in 256.40394s
25 21 2302176.051, 4502756.862 in 261.86602s
26 22 2302176.051, 4425269.328 in 263.04549s
27 23 2302176.051, 4282201.250 in 257.43772s
28 24 2302176.051, 4124508.978 in 293.82975s
29 25 2302176.051, 4085143.344 in 288.17532s
30 26 2302176.051, 4073004.917 in 294.70780s
31 27 2302176.051, 4064383.098 in 272.93611s
32 28 2302176.051, 4058093.835 in 270.17014s
33 29 2302176.051, 4053500.591 in 282.28608s
34 30 2302176.051, 4049981.234 in 305.65096s
35 31 2302176.051, 4047478.727 in 306.23214s
36 32 2302176.051, 4045653.070 in 268.57056s
37 33 2302176.051, 4044266.926 in 235.55561s
38 34 2302176.051, 4043261.358 in 233.46650s, 35 2302176.051, 4042516.466 in 238.02031s
39 36 2302176.051, 4041948.850 in 237.61914s, 37 2302176.051, 4041593.058 in 238.18756s
40 38 2302176.051, 4041294.865 in 236.84551s, 39 2302176.051, 4041085.270 in 237.16316s
41 40 2302176.051, 4040947.383 in 238.66290s, 41 2302176.051, 4040840.853 in 237.09202s
42 42 2302176.051, 4040765.225 in 236.26786s, 43 2302176.051, 4040714.555 in 237.45015s
43 44 2302176.051, 4040685.335 in 236.55129s, 45 2302176.051, 4040664.789 in 228.61187s
44 46 2302176.051, 4040650.566 in 222.66057s, 47 2302176.051, 4040640.882 in 223.82387s
45 48 2302176.051, 4040639.527 in 225.29655s, 49 2302176.051, 4040639.246 in 219.71104s
46 50 2302176.051, 4040638.352 in 218.92976s, 51 2302176.051, 4040637.346 in 221.38802s
47 52 2302176.051, 4040637.346 in 223.68960s, 53 2302176.051, 4040637.346 in 223.46427s
48 54 2302176.051, 4040637.346 in 222.89142s, 55 2302176.051, 4040637.346 in 224.58783s
49
50 Total time: 14146.268915s
51
52 [-339600 197368 -370671 354276 -70166 28695 -183448 -33069
53 80872 351769 132081 -201838 -9909 12119 -511889 -89837
54 -109625 8406 471259 -478732 384369 263323 47326 -69645
55 -240045 170426 -93560 -310323 -313859 -214420 230818 55934
56 118551 19638 56894 70908 97547 -301618 -240504 133062
57 -238342 -368412 -415858 396812 -318413 -382454 276124 252161
58 175288 359572 375526 -138434 394571 -314 -449954 442751
59 63585 5978 265059 -3108 -255629 35222 301053 -253249
60 -292464 -344766 425684 -14475 -429019 -254691 -44114 386152
61 295044 84301 74539 -100242 147268 174423 -3448 -276519]
```

---

**LISTING C.13**—Console output for the Random Integral Lattice of dimension 90.

---

```
1 Size: 540000
2 Inflated Size: 396817338
3
4 0 12565757.069, 19127898.656 in 0.82601s
5 1 12565757.069, 20825879.235 in 89.10190s
6 2 10806508.510, 16665756.501 in 1452.70228s
7 3 10094765.913, 14066187.946 in 3551.72896s
8 4 9430505.393, 12397811.062 in 1627.44096s
9 5 8562074.669, 11279434.603 in 3276.42792s
10 6 8166596.293, 10509634.338 in 1472.16558s
11 7 7870455.985, 9970886.094 in 1842.91860s
12 8 7538141.003, 9585989.255 in 1396.93509s
13 9 7484180.180, 9302168.233 in 1915.37626s
14 10 7334216.318, 9084781.626 in 1308.45306s
15 11 7221309.785, 8914041.545 in 2616.37581s
16 12 7221309.785, 8777556.677 in 2629.00043s
17 13 7092864.269, 8668527.591 in 2415.60366s
18 14 7092864.269, 8580829.793 in 1861.28425s
19 15 7092864.269, 8511124.650 in 2010.19133s
20 16 7092864.269, 8456532.372 in 1888.41986s
21 17 7056191.085, 8414024.882 in 2780.85258s
22 18 6815686.916, 8381860.775 in 2178.49765s
23 19 6815686.916, 8357392.419 in 2837.67543s
24 20 6815686.916, 8339054.162 in 2292.39425s
25 21 6815686.916, 8325504.422 in 2806.69109s
26 22 6815686.916, 8315583.028 in 2494.34028s
27 23 6815686.916, 8308371.345 in 2679.80559s
28 24 6815686.916, 8303201.226 in 2422.09905s
29 25 6815686.916, 8299505.668 in 2906.39287s
30 26 6815686.916, 8296929.808 in 2554.34885s
31 27 6815686.916, 8295106.530 in 2693.58219s
32 28 6815686.916, 8293797.510 in 2706.98771s
33 29 6815686.916, 8292915.201 in 2914.62314s
34 30 6815686.916, 8292316.899 in 2465.32231s
35 31 6815686.916, 8291916.141 in 3017.27342s
36 32 6815686.916, 8291631.523 in 2730.85150s
37 33 6815686.916, 8291434.344 in 2960.18202s
38 34 6815686.916, 8291303.694 in 2598.44423s
39 35 6815686.916, 8291211.576 in 2994.06267s
40 36 6815686.916, 8291136.596 in 2636.13106s
41 37 6815686.916, 8291084.270 in 2847.07248s
42 38 6815686.916, 8291041.302 in 2573.06227s, 39 6815686.916, 8291013.118 in 2379.01917s
43 40 6815686.916, 8290998.512 in 2448.44565s, 41 6815686.916, 8290989.700 in 2840.56000s
44 42 6815686.916, 8290985.248 in 2377.91228s, 43 6815686.916, 8290981.762 in 2708.42486s
45 44 6815686.916, 8290981.451 in 2652.30086s, 45 6815686.916, 8290981.451 in 3152.77417s
46 46 6815686.916, 8290981.451 in 2396.80088s, 47 6815686.916, 8290981.451 in 2866.93481s
47
48 Total time: 116719.233408s
49
50 [ -216392 -254160 -244361 1766807 33053 1196316 11135
51 517195 -65584 -880197 819334 1237370 -799418 -1065279
52 -289625 780252 -969293 1028042 297939 -712459 -536801
53 1168224 -257619 -508 -87883 614536 -366143 -1164554
54 -410758 877895 598200 155800 104513 -939216 346565
55 61115 724200 1649076 774776 667148 -926478 1086801
56 -787578 704054 1004884 667448 431540 858463 415670
57 -646366 -776581 -466631 -1483678 -478087 -235591 535390
58 -514709 734993 -116042 1392641 1148413 -647904 66027
59 -239725 687792 -1413944 21462 139370 159633 428087
60 286160 -397630 37336 -303788 -914635 1162143 -326583
61 411202 417682 960167 -47407 -397410 621647 557571
62 -389183 -400704 -629228 -387812 283521 729130]
```

---

**LISTING C.14**—Console output for the Random Integral Lattice of dimension 100.

---

```
1 Size: 1000000
2 Inflated Size: 1000000000
3
4 0 18932551.080, 35905816.976 in 1.63902s
5 1 18932551.080, 36784253.466 in 405.14790s
6 2 18932551.080, 29200366.806 in 8221.33308s
7 3 17341109.451, 24509853.424 in 6375.88184s
8 4 16535359.178, 21577716.471 in 13498.76059s
9 5 15306289.538, 19654802.755 in 5725.73987s
10 6 13646030.847, 18351638.569 in 14781.12272s
11 7 13646030.847, 17447482.293 in 5746.77149s
12 8 12980291.337, 16803638.381 in 12181.41788s
13 9 12980291.337, 16332587.029 in 7109.19432s
14 10 12883833.694, 15980877.583 in 8148.12825s
15 11 12771542.588, 15716687.636 in 6982.93687s
16 12 12304771.787, 15518317.160 in 5983.46464s
17 13 12304771.787, 15371265.351 in 5936.68367s
18 14 12304771.787, 15265086.489 in 6090.40237s
19 15 12304771.787, 15190219.528 in 10265.19854s
20 16 12002723.155, 15139714.819 in 6116.50036s
21 17 12002723.155, 15106752.617 in 9259.10887s
22 18 12002723.155, 15085846.841 in 8552.15365s
23 19 12002723.155, 15073277.214 in 9290.71900s
24 20 12002723.155, 15065830.260 in 5858.20754s
25 21 12002723.155, 15061488.163 in 7701.04737s
26 22 12002723.155, 15059112.235 in 10227.58401s
27 23 12002723.155, 15057809.398 in 7817.36694s
28 24 12002723.155, 15057135.045 in 6480.14918s
29 25 12002723.155, 15056783.454 in 11237.07091s
30 26 12002723.155, 15056592.965 in 5973.43812s
31 27 12002723.155, 15056477.569 in 9263.21726s
32 28 12002723.155, 15056418.092 in 9276.35056s
33 29 12002723.155, 15056390.185 in 9603.15581s
34 30 12002723.155, 15056378.214 in 8656.51352s
35 31 12002723.155, 15056372.563 in 11048.24149s
36 32 12002723.155, 15056370.787 in 9794.78303s
37 33 12002723.155, 15056368.351 in 9672.32885s
38 34 12002723.155, 15056366.072 in 6564.05774s
39 35 12002723.155, 15056364.664 in 8545.35274s
40 36 12002723.155, 15056363.830 in 8292.60240s
41 37 12002723.155, 15056363.596 in 6700.94190s
42 38 12002723.155, 15056363.596 in 10819.73680s
43 39 12002723.155, 15056363.596 in 11447.02165s
44
45 Total time: 335564.200763s
46
47 [ 983865 1799148 473859 -969291 192315 1301830 -1631373
48 -1651302 -937285 1049723 309804 1532689 3293333 -910264
49 -2058932 1335463 757675 -91444 -2924404 -1731557 44941
50 -421904 221037 -1391387 -470548 -282955 -432116 270009
51 393836 366214 -972271 -251675 -1974602 1996262 1641988
52 917285 -2268231 988695 -71500 1068972 147292 1040916
53 -1550687 -1115148 -1896057 -1272706 -1725401 1013424 1828959
54 145626 894488 -1344353 425125 1042887 -479725 34465
55 1349575 -64678 -556409 -1433394 197250 -1596157 1749975
56 1491244 -821677 -521348 -598619 -1573184 -639620 164524
57 1634194 453567 91261 -786734 -647058 604649 1484280
58 985203 433457 976618 -396071 491306 240511 112622
59 -2433305 261072 -327653 -2400143 -987475 -895515 1211204
60 689744 806986 1859051 1859357 -1329242 701309 536525
61 -868104 1169102]
```

---

---

**LISTING C.15**—Console output for the Module Lattice of dimension 20.

---

```
1 Size: 2000
2 Inflated Size: 89442
3
4   0 84572.456, 2885542.698 in 0.00239s
5   1 56512.009, 330246.922 in 0.00667s
6   2 45271.838, 148697.380 in 0.03829s
7   3 37823.791, 92776.321 in 0.03728s
8   4 26712.196, 68455.422 in 0.04117s
9   5 26188.633, 57348.574 in 0.04540s
10  6 26188.633, 50314.085 in 0.04594s
11  7 26188.633, 45977.444 in 0.04559s
12  8 26188.633, 44083.630 in 0.04282s
13  9 26188.633, 43778.022 in 0.03710s
14 10 26188.633, 43749.143 in 0.03396s
15 11 26188.633, 43742.382 in 0.03497s
16 12 26188.633, 43741.917 in 0.03315s
17 13 26188.633, 43741.631 in 0.03374s
18 14 26188.633, 43741.631 in 0.03500s
19 15 26188.633, 43741.631 in 0.03390s
20
21 Total time: 0.581106s
22
23 1600-5396i -728-5460i -1192-2753i -7297+2331i 2324-3458i -124-3604
    i 5401-86i -6980-7192i 7404-707i 230-1860i -7983+2436i -735-5560
    i -2987-3823i -2804-550i 7984+2634i 5244-1786i -91+3385i
    -752-1213i 7847+1170i -1166+5033i
```

---

**LISTING C.16**—Console output for the Module Lattice of dimension 30.

---

```
1 Size: 3000
2 Inflated Size: 164316
3
4 0 334568.004, 21747063.356 in 0.00615s
5 1 294224.804, 1487358.019 in 0.03163s
6 2 263893.768, 705033.180 in 0.07590s
7 3 214971.211, 493852.888 in 0.08234s
8 4 177814.207, 411461.226 in 0.09774s
9 5 177814.207, 370046.210 in 0.09718s
10 6 177814.207, 362966.907 in 0.09276s
11 7 177814.207, 362551.056 in 0.08917s
12 8 177814.207, 362463.843 in 0.08689s
13 9 177814.207, 362412.711 in 0.08409s
14 10 177814.207, 362396.942 in 0.08484s
15 11 177814.207, 362394.583 in 0.08545s
16 12 177814.207, 362394.583 in 0.08558s
17 13 177814.207, 362394.583 in 0.08440s
18 14 177814.207, 362394.583 in 0.08494s
19 15 177814.207, 362394.583 in 0.08506s
20
21 Total time: 1.338962s
22
23 -8741-4733i 8859-5602i -42503-19669i -46447+3036i -9672-4718i
    2888-571i -6846-3248i 19652+6075i 34057+30172i -26-2903i
    -11580+38969i 6514-26744i -36880+23014i -33827-32961i 16724+7857i
    11835+30385i -14525+27562i -35545+5801i -40979-2466i 47909+7576
    i 9023+3992i 9128-50037i -23429+21309i -6679-11767i -24203+15939
    i -26135+14197i 4070+43941i 352+3678i -38671-5801i -27394+16997i
```

---

---

**LISTING C.17**—Console output for the Module Lattice of dimension 40.

---

```
1 Size: 4000
2 Inflated Size: 252982
3
4 0 1113941.869, 898699777.831 in 0.01028s
5 1 1007763.969, 4232297.760 in 0.09705s
6 2 909163.639, 2074518.285 in 0.11462s
7 3 773994.067, 1675561.399 in 0.15030s
8 4 754691.690, 1474536.963 in 0.16465s
9 5 754691.690, 1414358.419 in 0.16045s
10 6 754691.690, 1413043.787 in 0.16165s
11 7 754691.690, 1413043.787 in 0.15100s
12 8 754691.690, 1413043.787 in 0.15337s
13 9 754691.690, 1413043.787 in 0.15217s
14 10 754691.690, 1413043.787 in 0.15419s
15
16 Total time: 1.622789s
17
18 62470+29798i 49157-32538i -91404+88941i -77679+87276i 108618+54958i
    -86152+5762i 99799+48388i 119157+194334i -20294+8614i
    -12750+39326i 36446+2289i -71975+141937i 114465-52125i
    143865-25491i 103157+53997i 119441+55502i 90724-26739i 65520+5981
    i 55285+24160i 172740-45810i -110845+34940i 43688+29043i
    -77274-44924i -85432-2991i 15432-122988i 25079+119796i
    -246658-125800i -146801+109064i -25954-43398i 89964-85570i
    17282+70242i 38789-29997i -20040+40871i -163607+3065i
    -80957+65858i 65496-9054i -118557-140110i 28394-8809i 77110+72349i
    87221+63005i
```

---

---

**LISTING C.18**—Console output for the Module Lattice of dimension 50.

---

```
1 Size: 5000
2 Inflated Size: 353553
3
4   0 2331490.683, 63388190389.113 in 0.01517s
5   1 2331490.683, 16749892.983 in 0.16518s
6   2 2331490.683, 6685854.143 in 0.16639s
7   3 2231875.927, 4499576.229 in 0.20083s
8   4 1886516.046, 3876792.822 in 0.22822s
9   5 1482020.579, 3453654.532 in 0.23793s
10  6 1482020.579, 3278451.566 in 0.23943s
11  7 1482020.579, 3272285.708 in 0.23716s
12  8 1482020.579, 3272285.708 in 0.23528s
13  9 1482020.579, 3272285.708 in 0.26416s
14 10 1482020.579, 3272285.708 in 0.23741s
15 11 1482020.579, 3272285.708 in 0.23264s
16
17 Total time: 2.695209s
18
19 -18282-158354i -20262+141970i -62034+224094i 88861-101376i
    237565+125756i 3699-258810i 77698-162477i -223799+99823i
    -74136+150303i -8723-224707i -44319+94620i -71863+17105i
    -259618-185551i -16178-125873i 120536+86175i 235464-114872i
    -56016-14808i 8742-37899i 127076+209981i -88031+92802i
    -34763+268220i -87796+82321i 3550-49607i 39678-422485i
    148582+118812i 127052-175541i -55005-67925i 94046+25798i
    36275+143761i 29278-94065i -172090+190080i -11190+234192i
    -68030-322406i -261112+188055i -16935+259181i 97967-18591i
    194337+106060i 122534+86970i -43881-200066i 7347-296673i
    -74312-93190i -43577-163199i 22274-20261i 84479-198683i
    -188067+101651i -22710-166330i 123499-117948i 103556-221304i
    -66064-107835i -194101+375115i
```

---