Coding large lattice-free ellipsoids and investigating linear codes' building methods

Mattheo Yang

Number Theory Lab

Summer Project

September 2025

Supervisor
Amal Seddas
EPFL / Number
Theory Lab

Director
Maryna Viazovska
EPFL / Number
Theory Lab

Contents

1 Introduction **3**
- 1.1 Abstract . . . 3
- 1.2 Motivation . . . 3

2 Programming a cn^2 volume ellipsoid without lattice points **4**
- 2.1 Definitions and Notations . . . 4
- 2.2 Klartag's method . . . 6
- 2.3 Applying the method to the lattice $\mathbb{Z}^2$ . . . 7
- 2.4 Applying the method to the lattice $\mathbb{Z}^n$ . . . 13
- 2.5 Applying the method to any lattice $L \subset \mathbb{R}^n$ . . . 13
- 2.6 Finding the corresponding lattice sphere packing . . . 14

3 Investigating methods to build good linear codes **16**
- 3.1 Rank 1 Random Noise (R1) . . . 17
- 3.2 Simulated Annealing (SA) . . . 21
- 3.3 Random Sampling . . . 22
- 3.4 Comparison . . . 23

4 Acknowledgments **25**

References **26**

1 Introduction

1.1 Abstract

In 2025, Boaz Klartag gave a construction for an origin-centered ellipsoid that accumulates $n(n+1)$ lattice points on its boundary, while having no non-zero lattice points in its interior, for any $n \geq 1$ and a lattice $L \subset \mathbb{R}^n$. We implement his proof in programming, which gives the explicit expression for an example of such an ellipsoid. Moreover, we give an algorithm that gives a lattice sphere packing whose density is $cn^2 2^{-n}$, for any $n \geq 1$. An analogous problem in $\mathbb{F}_2$ is building $[n, k]$ linear codes with high minimum Hamming distance, and it is constitutes active field of research. We investigate two different methods, *simulated annealing* and *rank 1 random noise*, to see which one is most promising, and we try to make improvements.

1.2 Motivation

Let $n \geq 1$, and let $S \subset \mathbb{R}^n$ be convex and origin symmetric, i.e. if x is in S, then $-x$ is in S. Minkowski's convex body theorem states that if the volume of S is strictly superior than 2^n, then S contains a non-zero integer point. Equivalently, if S doesn't have a non-zero integer point, then its volume is $\leq 2^n$. Therefore, there exists ellipsoids of very large volume, and that doesn't contain any integer points. However, when Minkowski proved this theorem in 1896, he didn't provide a construction for such an ellipsoid. In the 20th and 21st century, existence proofs for ellipsoids with volume cn, were given, first by Rogers [3] in 1947. Improvements on the constant c were made throughout the century. Recently, Boaz Klartag [2] published a paper giving a construction for an ellipsoid (no integer points in the ellipsoid's interior), whose volume is $cn^2, c > 0$ which is a great improvement from the previous lower bound dn, where $d > 0$. This ellipsoid has $n(n+1)$ points on its boundary, whilst having no non-zero lattice points in its interior. Equivalently, he improved the previous lattice sphere packing density bound $cn2^{-n}$ to $cn^2 2^{-n}$. Most importantly, he gives an explicit construction for such an ellipsoid. One of our goals was to implement his ideas in SageMath, in order to explicitly construct approximations of such ellipsoids, given $n \geq 2$ and a lattice $L \subset \mathbb{R}^n$. Moreover, we also give an algorithm to find a lattice sphere packing of volume at least $cn^2 2^{-n}$, for any $n \geq 1$.

An analogous problem can be stated in $\mathbb{F}_2$. A binary linear code of length n and dimension k is a subspace $C \subset \mathbb{F}_2^n$ of dimension k. We write $[n, k]$ to represent such a code. Where humans use words to communicate, computers use codes to communicate. However, during transmission, errors can occur and the intended message is changed, which is problematic. Some linear codes are called *error correcting codes*, as they have the ability to correct and detect errors that might have happened during the transmission, up to some extent. Now, equip $\mathbb{F}_2^n$ with the Hamming distance d defined by

$$d(x, y) = \sum_{j=0}^{n} \delta(x_i, y_i),$$

where $x = (x_1, \ldots, x_n), y = (y_1, \ldots, y_n) \in \mathbb{F}_2^n$, and δ represents the Kronecker symbol.

Basically, the hamming distance between two vectors is the number of coordinates in which they differ.

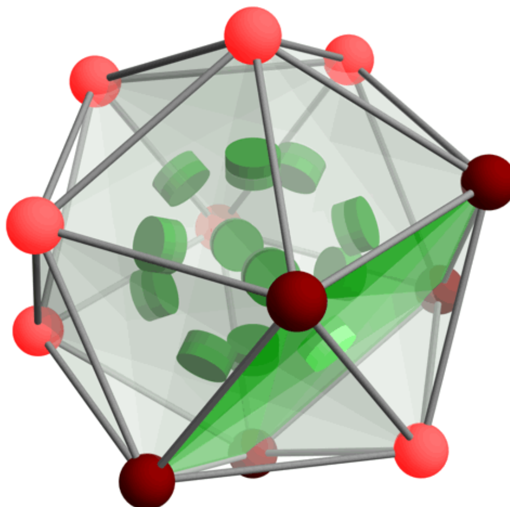


Figure 1: Extended Binary Golay Code - Greg Egan

Moreover, given that $\mathbb{F}_2^n$ is finite, any linear code $C \subset \mathbb{F}_2^n$ is finite, one can define the *minimum hamming distance* (or simply *minimum distance*) of C as

$$d_{\min}(C) = \min_{x,y \in C} d(x, y).$$

One can show that error-correcting codes can correct up to $\left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$ errors, and detect up to $d_{\min} - 1$ errors. Hence, a code with a higher minimum distance will be more effective.

However, given n and k, there is no efficient algorithm to produce an $[n, k]$ code with a high minimum distance. Although mathematicians have found lower and higher bounds for the minimum distance of some $[n, k]$ codes, such bounds are unknown for random codes (most codes). A problem is to create an algorithm that, given n, k, consistently creates a code that reaches a high minimum distance. Amal Seddas and director Maryna Viazovska worked on creating such algorithms, using two different methods. They tested their methods mostly on the extended Binary Golay code $[24, 12, 8]$ which has known minimum distance 8. Our goal was to compare these codes to determine which would be most promising for future work, and try to optimize them, in the sense make them faster or make modifications in their methods to yield better results. Another idea was to create a new method, but we were advised not to go down that path.

2 Programming a cn^2 volume ellipsoid without lattice points

In any dimension $n \geq 1$, Klartag gives a method to construct an origin-centred ellipsoid that accumulates $n(n + 1)$ lattice points on its boundary, while keeping its interior lattice-free. In this section, we implement his algorithm in SageMath, which uses the programming language Python. Moreover, we compute the corresponding lattice sphere packing. The following program will be on GitHub.

2.1 Definitions and Notations

We start with some definitions and notations, mainly used in Klartag's article.

Let $n \geq 2$. The space of symmetric real matrices is denoted by $\mathbb{R}^{n\times n}_{\text{sym}}$. A *lattice* $L \subset \mathbb{R}^n$ is the image of $\mathbb{Z}^n$ under an invertible, linear transformation $T : \mathbb{R}^n \to \mathbb{R}^n$. A *lattice sphere packing* is a collection of disjoint euclidean balls or radius $r > 0$, whose centers form a lattice.

For a lattice L, its *covolume* is defined as

$$Vol\,(\mathbb{R}^n/L) := |\det T|,$$

and a lattice's *sphere packing density* (take balls of radius r) is

$$\frac{Vol\,(rB^n)}{Vol\,(\mathbb{R}^n/L)}.$$

An open subset $\mathcal{E} \subset \mathbb{R}^n$ is said to be L-free if $\mathcal{E} \cap L \subset \{0\}$. An origin-centred ellipsoid $\mathcal{E} \subset \mathbb{R}^n$ is the set

$$\{x \in \mathbb{R}^n \mid x^t Ax \leq 1\},$$

where $A \in \mathcal{M}_n(\mathbb{R})$ is symmetric and positive-definite. Given a symmetric, positive-definite matrix M, we will sometimes refer to the ellipsoid given by M simply as $\mathcal{E}_M$, or even just $\mathcal{E}$ if there isn't any confusion possible.
Denote the open Euclidean unit ball in $\mathbb{R}^n$ by B^n. Its volume is defined by

$$Vol_n(B^n) := \frac{\pi^{n/2}}{\Gamma\left(\frac{n}{2}+1\right)}.$$

The volume of an ellipsoid is

$$Vol_n(\mathcal{E}_A) = \frac{Vol_n\,(B^n)}{\sqrt{\det(A)}}$$

The Frobenius inner product is noted, for $A, B \in \mathcal{M}_n(\mathbb{R})$:

$$\langle A, B\rangle_F = \operatorname{Tr}\left(A^t B\right),$$

and the standard dot product in $\mathbb{R}^n$ will be noted $u \cdot v$.
For $M \in \mathcal{M}_n(\mathbb{R})$, $M = \sum_{i=1}^n \sum_{j=1}^n m_{ij} e_{ij}$, we write

$$\operatorname{Vec}(M) = \begin{pmatrix} m_{11} \\ m_{12} \\ \cdots \\ m_{1n} \\ \cdots \\ m_{n1} \\ m_{nn} \end{pmatrix},$$

where $(e_{ij})_{1\leq i,j\leq n}$ is the canonical basis of $\mathcal{M}_n(\mathbb{R})$. Moreover, we will use Brownian motion, which is a mathematical concept to describe "random evolution". One doesn't need to completely understand it to be able to understand this document, but one can refer to [1] for more information.

2.2 Klartag's method

In his article, Klartag explains in detail how he constructs such an ellipsoid and why his method works, but I will only expose the different steps he used. For the sake of clarity, we decompose his method in 9 steps (1)–(9).

Phase 0. Fix a lattice $L \subset \mathbb{R}^n$ of covolume equal to $Vol_n(B^n)$, verifying

$$B\left(0, 1 - \frac{1}{n}\right) \cap L = \{0\}. \tag{1}$$

Then, take a standard Brownian motion in $\mathbb{R}^{n\times n}_{\text{sym}}$:

$$(W_t)_{t\geq 0}\,, \text{where } W_0 = 0. \tag{2}$$

Phase 1. Define

$$A_0 = \left(1 - \frac{1}{n}\right)^{-2} I_n, \tag{3}$$

and

$$\tau_1 \coloneqq \sup\left\{t \geq 0 \mid \mathcal{E}_s \text{ is } L\text{-free with } \partial\mathcal{E}_s \cap L = \partial\mathcal{E}_0 \cap L, \forall s \in [0, t]\right\}. \tag{4}$$

Then, for $t \leq \tau_1$, set

$$A_t = A_0 + W_t \tag{5}$$

where we denoted $\mathcal{E}_t \coloneqq \{x \in \mathbb{R}^n \mid x^t A x < 1\}$ the interior of the origin-centred ellipsoid associated to A_t, and Stop step 1 once $t = \tau_1$.

Phase $i \geq 2$. For any $t \geq 0$, consider the subspace

$$F_t \coloneqq \left\{B \in \mathbb{R}^{n\times n}_{\text{sym}} \mid x^t B x = 0, \forall x \in \partial\mathcal{E}_t \cap L\right\}. \tag{6}$$

"'maybe explain what this represents, or useless"' Let B_t be an orthonormal basis of F_t, and denote the orthogonal projection onto F_t by

$$\begin{aligned} \pi_t : \mathbb{R}^{n\times n}_{\text{sym}} &\to \mathbb{R}^{n\times n}_{\text{sym}} \\ A &\mapsto \sum_{e_i \in B_t} \langle A, e_i\rangle e_i. \end{aligned} \tag{7}$$

Once again, define

$$\tau_i = \sup\left\{t \mid \mathcal{E}_s \text{ is } L\text{-free with } \partial\mathcal{E}_s \cap L = \partial\mathcal{E}_{\tau_{i-1}} \cap L, \forall s \in [\tau_{i-1}, t]\right\}. \tag{8}$$

Then, for $\tau_{i-1} \leq t \leq \tau_i$, set

$$A_t = A_{\tau_{i-1}} + \pi_{\tau_{i-1}}\left(W_t - W_{\tau_{i-1}}\right), \tag{9}$$

where similarly, Reiterate this step until $\pi_t = 0$. This is guaranteed to happen, see [2] for a proof.
At this stage we are done, and the last matrix A_{t_f} obtained gives the desired ellipsoid.

2.3 Applying the method to the lattice $\mathbb{Z}^2$

As advised by Seddas, we used the programming language Python and the software SageMath to apply Klartag's method. We started with an easy case, the case $L = \mathbb{Z}^2$. The main problem is translating each step in Python, which isn't so trivial. To simplify and for the sake of clarity, we will heavily modularise the code.

Phase 0. $\mathbb{Z}^2$ already verifies (1). For step (2), define

```
def symmetric_bmotion(size, dt = 1.0) :
    # Diagonal part: independent N(0, dt)
    diag = np.random.normal(0, np.sqrt(dt), size)

    # Strictly upper triangular part: independent N(0, dt)
    G = np.triu(np.random.normal(0, np.sqrt(dt), (size,size)), 1)

    # Build symmetric matrix
    M = np.diag(diag) + (G + G.T)/np.sqrt(2)

    return matrix(RDF, M)
```

Given `dt`, this returns a matrix whose entries follow a Brownian motion of variance `dt`. This is how we will build $(W_t)_{t\geq 0}$. Then, step (3) is straightforward :

```
def initialize(n) :
    return (1-1/n)**(-2) *identity_matrix(RDF, n, n)
```

Steps (4) and (5) are a bit more delicate, and we need to define auxiliary functions. First, we define a function that verifies if for a fixed s, an ellipsoid $\mathcal{E}_A$, verifies, $\partial\mathcal{E}_s \cap L = \partial\mathcal{E}_0 \cap L$:

```
def tau_condition_2(A, E) :
    if Counter(boundary_lattice_points(A)) != Counter(
        boundary_lattice_points(E)) :
         return False
    return True
```

where `E` represents $\mathcal{E}_0$. Now, given a matrix `A`, we define `boundary_lattice_points(A)` as follows :

```
def boundary_lattice_points(M, tol = 0.05) :
    lambda_m = min(M.eigenvalues())
    if lambda_m <= 0:
        return []
    x_max = floor(1/sqrt(lambda_m)) + 5
    L = []
    for i in range(-x_max, x_max+1):
        for j in range(-x_max, x_max+1):
            if (i,j) != (0,0) and on_ellipsoid(M, i, j, tol):
                L.append((i,j))
    return L
```

where we defined

```
def on_ellipsoid(M, i, j, tol = 0.05) :
    x = vector(RDF, [i,j])
    return abs((x *(M * x)) - 1) < tol
```

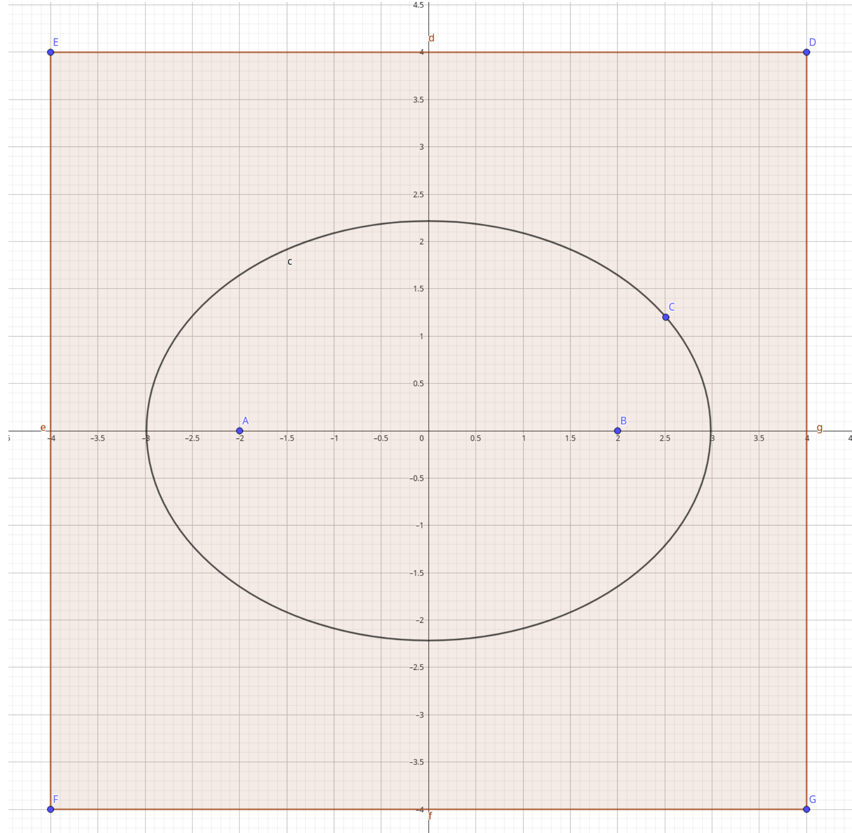


Figure 2: Visual Representation of `boundary_lattice_points` - GeoGebra

`boundary_lattice_points` returns the list of lattice points sitting on the ellipsoid $\mathcal{E}_M$, with a arbitrary precision `tol`. Let us explain how it works. Given a symmetric definite-positive matrix M, the spectral theorem and the definition of definite-positiveness gives that all eigenvalues are strictly positive. Then, denote the smallest eigenvalue by λ_m. One can show that for all $x \in \mathcal{E}_A$,

$$\|x\|_2 \leq \frac{1}{\sqrt{\lambda_m}} \leq \left\lfloor \frac{1}{\sqrt{\lambda_m}} \right\rfloor + 1.$$

Therefore, to determine all lattice points in $\mathcal{E}_A$'s boundary, it suffices to check that for all lattice points in the origin-centred square of length $\left\lfloor \frac{1}{\sqrt{\lambda_m}} \right\rfloor + 1$ are in $\mathcal{E}_A$'s boundary. Then, for the other condition, we define

```
1 def tau_condition_1(A) :
2     if has_lattice_point_interior(A) :
3         return False
4     return True
```

and we also defined `has_lattice_points_interior` :

```
1 def has_lattice_point_interior(M) :
2     E = M.eigenvalues()
3     lambda_m = min(E)
4     if ellipsoid_volume(M) > 2**n :
5         return True                              #Minkowski's
          theorem for case 1
```

```
6     x_max = floor(1/sqrt(lambda_m)) + 1
7     for i in range(x_max) :
8         for j in range(-x_max, x_max) : #take all lattice points in
              a square that contains the ellipsoid
9             if (i, j) == (0, 0) :
10                continue
11            if in_ellipsoid(M, i, j) or in_ellipsoid(M, -i, j) :
12                return True
13
14    return False
```

The function `in_ellipsoid` is defined exactly like `on_ellipsoid`, but with the return statement being `return (not on_ellipsoid(M, i, j)) and abs((x *(M * x))+tol) < 1`. Now, we have functions allowing us to find τ_1. Now, we define a method determining τ_1, and that also updates our list of matrices `A_t` :

```
1 def step1(A_list, Brownian_list) : #find tau and update list of
      matrices A_t
2     previous_tau = len(A_list) - 1 #tau_{i-1}
3     l = previous_tau
4     while tau_condition(A_list[-1], A_list[previous_tau]):
5         A_list.append(A_list[previous_tau] + Brownian_list[l])
6         l += 1
7     return (l-1, A_list)
```

Phase 1 is over.

Phase i**.** The hard part here is step (6). *A priori*, we don't have any programming tools to compute an orthonormal basis of an abstract vector space. However, SageMath has the functions `right_kernel(A)` and `basis()`, and given a matrix `A`, `right_kernel(A).basis()` returns a basis for `A`'s kernel. Let's try to turn this problem into a kernel problem.

Let A be symmetric and definite-positive. Write $\partial\mathcal{E}_t \cap L = \{x_1, \ldots, x_l\}$. For any fixed t and $x \in \partial\mathcal{E}_t \cap L$, we have

$$\begin{aligned} x^t A x &= \langle x, Ax \rangle_F \\ &= \operatorname{Tr}\left(x^t A x\right) \\ &= \operatorname{Tr}\left(x^t x A\right) \\ &= \operatorname{Vec}\left(x^t x\right)^t \cdot \operatorname{Vec}(A), \end{aligned}$$

using $\operatorname{Tr}(AB) = \operatorname{Tr}(\text{BA})$ for all $A, B \in \mathcal{M}_n(\mathbb{R})$. Then, write

$$M = \begin{pmatrix} \operatorname{Vec}(x_1^t x_1)^t \\ \operatorname{Vec}(x_2^t x_2)^t \\ \cdots \\ \operatorname{Vec}(x_l^t x_l)^t \end{pmatrix}.$$

Hence, to find a basis of F_t in step (6), it suffices to find a basis of M's kernel. Hence, we define

```
1 def Ft_basis(boundary_points) :
2     if boundary_points == [] :
3         return []
4     l = len(boundary_points) #number of boundary lattice points
```

```
5       n = len(boundary_points[0]) #size of the matrix
6       rows = []
7       for p in boundary_points :
8           x = vector(RDF, p).column()
9           xxT = x * x.transpose()
10          rows.append(sym_to_vec(xxT))
11      A = matrix(RDF, rows)
12      B = A.right_kernel().basis()
13      return [vec_to_sym(v, n) for v in B]
```

sym_to_vec is the function that takes an $n \times n$ matrix (supposed symmetric), and returns Vec(matrix) in a modified way : we ignore the duplicates, thus the vector has $\frac{n(n+1)}{2}$ entries. vec_to_sym is its inverse function. At this point, we have a basis for F_t. Now, we orthonormalise it using the Gram-Schmidt algorithm, already implemented in SageMath :

```
1   def symmetric_orthornormalisation(B) : #B is a list of matrices (
        basis)
2       if B == [] :
3           return []
4       n = B[0].nrows()
5       L = [sym_to_vec(M) for M in B]
6       M = matrix(RDF, L)
7       orthonormalised_vectors_matrix = M.gram_schmidt(orthonormal =
           True)
8       orthonormalised_vectors = [v for v in
           orthonormalised_vectors_matrix[0]]
9       result = [vec_to_sym(v,n) for v in orthonormalised_vectors]
10      return result
```

sym_to_vec ensures the returned basis is indeed symmetric, as desired. Step (6) is now complete. Step (7) is straightforward :

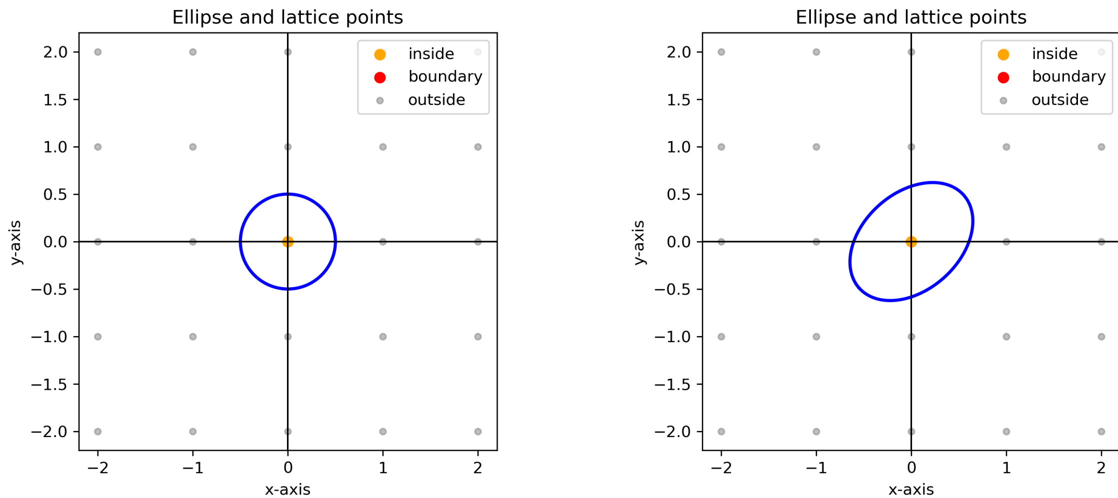
```
1   def projection_t(M, A_list, t) : #project M onto F_t
2       n = M.nrows()
3       blp = boundary_lattice_points(A_list[t])
4       if blp == [] :
5           return zero_matrix(RDF, n, n)
6       B = Ft_basis(blp)
7       new_basis = symmetric_orthornormalisation(B)
8       projection = zero_matrix(RDF, n, n)
9       for V in new_basis :
10          projection += frobenius_product(M, V) * V
11      return projection
```

Moreover, steps $(8) - (9)$ are analogous to steps $(3) - (4)$:

```
1   def stepi(A_list, Brownian_list) : #find tau and update list of
        matrices A_t
2       previous_tau = len(A_list) - 1 #tau_{i-1}
3       l = previous_tau
4       while tau_condition(A_list[-1], A_list[previous_tau]):
5           A_list.append(A_list[previous_tau] + projection_t(
               Brownian_list[l + 1] - Brownian_list[previous_tau],
               A_list, previous_tau))
6           l += 1
7           if l%1000 == 0 :
```

```
8                print(f"Step {l}")
9                print(f"Matrix {l}, volume : ", ellipsoid_volume(At[l])
                     .n(digits = 3))
10               print("boundary :", boundary_lattice_points(At[l]))
11               print("int :", interior_lattice_points(At[l]))
12               plot_ellipse_with_all_lattice(At[l], bound=6)
13       if not tau_condition_1(A_list[-1]):
14           print("condition1")
15       else :
16           print("condition2")
17
18       print(f"Step {l}")
19       print("Matrix ", len(At)," volume : ", ellipsoid_volume(At[-1])
             .n(digits = 3))
20       print("boundary :", boundary_lattice_points(At[-1]))
21       print("int :", interior_lattice_points(At[-1]))
22       plot_ellipse_with_all_lattice(At[l], bound=6)
23       return (l-1, A_list)
```

Finally, our program is complete. To run the program, we choose the Brownian motion's covariance to be very small to make the ellipsoid move very slowly, as big changes to its shape would be detrimental to its execution (due to the imprecisions induced by our arbitrary choice of the tolerance). Here is an example :



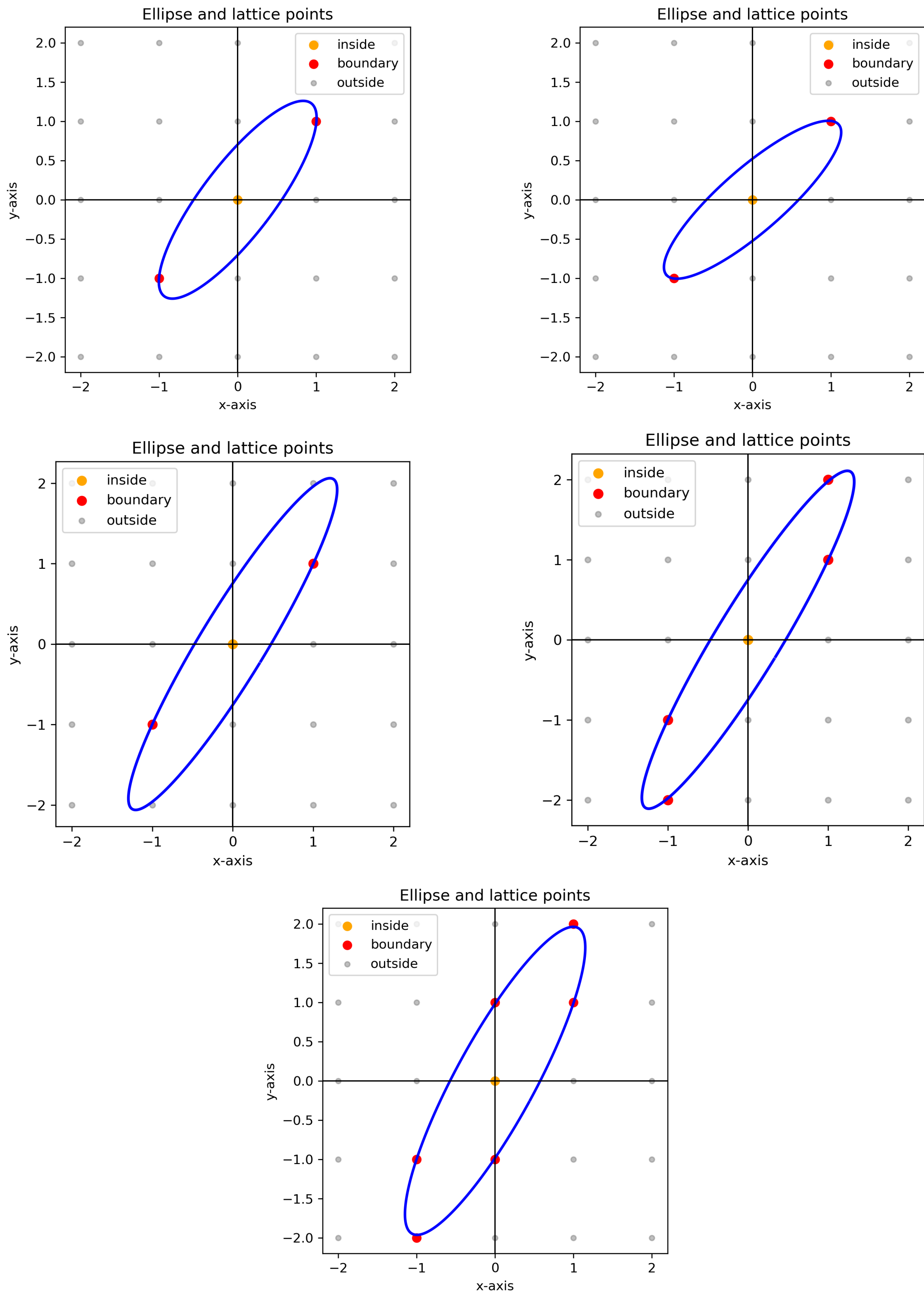


Figure 4: Evolution of the ellipsoid over time, steps 0, 900, 1350, 4000, 7035, 7040, 7582

This ellipsoid contains $n(n+1) = 6$ boundary points while having no non-zero lattice

points in its interior. The boundary points are

$$(1,2),(-1,-2),(1,1),(-1,-1),(0,1),(0,-1).$$

Its volume is approximately $1.1293909598967877\pi \approx 3.55$, and the associated matrix is

$$\mathcal{E}_{\textbf{final}} \equiv \begin{pmatrix} 2.9269 & -1.7176 \\ -1.716 & 1.2731 \end{pmatrix}.$$

2.4 Applying the method to the lattice $\mathbb{Z}^n$

Now, all the functions work in $\mathbb{Z}^n$, by fixing n in the functions (and making some easy adaptations in some). To avoid making this document longer than needed, we will not provide results here but the code is available on GitHub.

2.5 Applying the method to any lattice $L \subset \mathbb{R}^n$

Now, we have only tested the algorithm for lattices of the form $\mathbb{Z}^n$. But it also works for any lattice L, by writing $L = T(\mathbb{Z}^n)$. Then, it suffices to determine an ellipsoid for $\mathbb{Z}^n$, and scale using T. We claim the following :

Proposition 2.5.1. *Consider the lattice $\mathbb{Z}^n, n \geq 1$. Let $\mathcal{E}$ be an origin-centred ellipsoid, with $n(n+1)$ lattice points on its boundary. Fix a lattice $L = T(\mathbb{Z}^n)$, where $T : \mathbb{R}^n \to \mathbb{R}^n$ is linear, and invertible. Then,*

$$T(\mathcal{E}) = \{Tx \mid x \in \mathcal{E}\} \text{ is an ellipsoid,}$$

and the associated matrix is

$$A' = \left(T^{-1}\right)^t AT^{-1}.$$

Moreover, $T(\mathcal{E})$ is L-free and has exactly $n(n+1)$ lattice points on its boundary.

Proof. Let $\mathcal{E} = \{x \in \mathbb{R}^n \mid x^t Ax \leq 1\}$. Let $y \in \mathcal{E}$. Then by bijectivity of T, there exists a unique $x \in \mathbb{R}$ such that $y = Tx \iff x = T^{-1}y$. Recall that here, T, T^{-1} are matrices, using the identification $\mathcal{L}(\mathbb{R}^n) \cong \mathcal{M}_n(\mathbb{R})$.
Then, we have the following equivalencies :

$$\begin{aligned} x^t Ax \leq 1 &\iff \left(T^{-1}y\right)^t A\left(T^{-1}y\right) \leq 1 \\ &\iff y^t\left(\left(T^{-1}\right)^t AT^{-1}\right) y \leq 1 \end{aligned}$$

With equality if and only if $x^t Ax = 1$. Notice that this also shows that $\left(\left(T^{-1}\right)^t AT^{-1}\right)$ is positive-definite. Indeed, if we just take $x \in \mathbb{R}^n$, we have $x^t Ax > 0 \iff 0 < y^t\left(\left(T^{-1}\right)^t AT^{-1}\right) y$, because A is positive-definite, and this is true for any $y \in \mathbb{R}^n$ by surjectivity of T. We have proved that $T(\mathcal{E})$ is an ellispoid, because for any $y \in T(\mathcal{E})$, we showed that $y^t A' y \leq 1$.

Then, both ellipsoids have the exact number of boundary lattice points, as $x^t Ax = 1 \iff y^t A' y = 1$, where for a given y, such an $x \in \mathcal{E}$ is unique, by bijectivity of T. Moreover, the new boundary lattice points are exactly the image of the initial boundary points, i.e.

$\partial\mathcal{E}' \cap L = T(\partial\mathcal{E} \cap \mathbb{Z}^n)$. Lastly, the ellipsoid $T(\mathcal{E})$ doesn't have any interior lattice points, because if that were the case, there would exist $u \in L$ such that

$$u^t A' u < 1 \iff T^{-1}uAT^{-1}u < 1,$$

but by definition, $T^{-1}u \in \mathbb{Z}^n$, which would mean that $\mathcal{E}$ has an interior lattice point, which is absurd. The proposition is proved. $\square$

Now, if we're given a lattice L through an invertible, linear map T, the proposition proves that it suffices to determine an ellipsoid (matrix) for $\mathbb{Z}^n$ using our algorithm, then the matrix $\left(T^{-1}\right)^t AT^{-1}$ gives the wanted ellipsoid.

Let's give two nice examples. First, an example in $\mathbb{R}^2$. It is a known fact that the optimal lattice sphere packing in $\mathbb{R}^2$ is the hexagonal lattice, given by the image of $\mathbb{Z}^2$ under the linear, invertible map

$$T : \mathbb{R}^2 \to \mathbb{R}^2 \equiv \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} \end{pmatrix}.$$

Then,

$$T^{-1} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{3}} \end{pmatrix}.$$

Using the ellipsoid computed in section 2.3, we approximate

$$\left(T^{-1}\right)^t \mathcal{E}_{\text{final}} T^{-1} \approx \begin{pmatrix} 2.9269 & -3.6722 \\ -3.6722 & 4.9622 \end{pmatrix}.$$

And this gives our new ellipsoid.

2.6 Finding the corresponding lattice sphere packing

In Klartag's article, he states in his abstract that the existence of an origin centred, $\mathbb{Z}^n$-free ellispoid of volume cn^2 is equivalent to the existence of a lattice sphere packing whose density is at least $cn^2 2^{-n}$. We know how to find such ellipsoids. Now, how can we find such a lattice sphere packing ?

First, it is a known fact that any ellipsoid is the image of the euclidean unit ball under an invertible, linear map T, i.e $T(B^n) = \mathcal{E} \iff B^n = T^{-1}(\mathcal{E})$. But given an ellipsoid, how does one find a map from that ellipsoid to the unit ball ? By the previous proposition, it suffices to find $T \in \mathcal{M}_n(\mathbb{R})$ such that

$$TT^t = A,$$

where A is the matrix associated to $\mathcal{E}$. The following proposition gives a method to, given an ellipsoid $\mathcal{E}$, find an invertible, linear map T such that $T(\mathcal{E}) = B^n$.

Proposition 2.6.1. *Let $\mathcal{E} \subset \mathbb{R}^n$ be an ellipsoid, and denote its associated matrix by A. Let $T \in \mathcal{M}_n(\mathbb{R})$ such that its i-th column is $\sqrt{\lambda_i}v_i$, where λ_i and v_i are A's respective i-th eigenvalue and (basis) eigenvector. Then,*

$$T(\mathcal{E}) = B^n.$$

Proof. By hypothesis, A is a real, symmetric and positive-definite matrix. Hence, by the spectral theorem, A is orthogonally diagonalizable, and because it is positive-definite, all its eigenvalues are strictly positive. Let $(\lambda_i)_{1\leq i\leq n}$ be its eigenvalues such that

$$0 < \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n,$$

and let $(v_i)_{1\leq i\leq n}$ be a basis of eigenvectors such that for each i, $Av_i = \lambda_i v_i$. Let T be like in the proposition. Let $D = \text{diag}(\lambda_1, \ldots, \lambda_n)$, and let P^t, P respectively be the change of basis matrices, from the canonical basis to B, and from B to the canonical basis. Hence,

$$A = PDP^t.$$

For any $1 \leq i, j \leq n$, we have

$$\begin{aligned}(TT^t)_{ij} &= \sum_{k=1}^{n} T_{ik}T_{jk} \\ &= \sum_{k=1}^{n} \sqrt{\lambda_k}\,(v_k)_i \sqrt{\lambda_k}\,(v_k)_j \\ &= \sum_{k=1}^{n} \lambda_k\,(v_k)_i\,(v_k)_j\end{aligned}$$

and

$$\begin{aligned}\left(PDP^t\right)_{ij} &= \sum_{k=1}^{n}\sum_{l=1}^{n} P_{il}D_{lk}P^t_{kj} \\ &= \sum_{k=1}^{n} P_{ik}D_{kk}P_{jk} \\ &= \sum_{k=1}^{n} \lambda_k\,(v_k)_i\,(v_k)_j \\ &= \left(TT^t\right)_{ij}.\end{aligned}$$

This proves that $TT^t = A$. The equation implies $\det(TT^t) = \det(T)^2 = \det(A) \neq 0$, which means that T is bijective. Now, we prove that $T(\mathcal{E}) = B^n$. First, we see that for any $x \in \mathcal{E}$,

$$\begin{aligned}\|Tx\| &= \langle Tx, Tx\rangle \\ &= x^tT^tTx \\ &= x^tAx \leq 1.\end{aligned}$$

This shows that $T(\mathcal{E}) \subset B^n$. Conversely, for any $y \in B^n$, surjectivity of T gives the existence of $x \in \mathbb{R}^n$ (a priori, not in $\mathcal{E}$), such that $x = T^{-1}y \iff Tx = y$. Let's show that x is in $\mathcal{E}$. We have

$$\begin{aligned}y^ty \leq 1 &\iff (Tx)^tTx \leq 1 \\ &\iff x^tT^tTx \leq 1 \\ &\iff x^tAx \leq 1\end{aligned}$$

Then $x \in \mathcal{E}$, and $y \in T(\mathcal{E})$. Thus, $B^n \subset T(\mathcal{E})$ and $B^n = T(\mathcal{E})$, which proves the proposition. □

Now, we have a method to transform any ellipsoid into the unit euclidean ball and by scaling, we also have a method to transform any ellipsoid into the euclidean ball of any given volume. We want to find the corresponding lattice sphere packing talked about previously.

Proposition 2.6.2. *Let $\mathcal{E}$ be an $\mathbb{Z}^n$-free, origin-centred ellipsoid with volume cn^2. Let T be an invertible, linear map such that $T(\mathcal{E}) = T(rB^n)$ and $r > 0$ is such that $Vol(rB^n) = Vol(\mathcal{E}) = cn^2$. Let $L = T(\mathbb{Z}^n)$. Then, the lattice sphere packing*

$$x + \frac{r}{2}B^n \ (x \in L)$$

has density $cn^2 2^{-n}$.

Proof. First, let's show that T is well defined. By Proposition 2.6.1, there exists an invertible linear map $T : \mathbb{R}^n \to \mathbb{R}^n$ such that $T(\mathcal{E}) = B^n$. Then, the map $T' = rT$ is linear and invertible, and we have

$$\begin{aligned} T'(\mathcal{E}) &= rT(\mathcal{E}) \\ &= rB^n. \end{aligned}$$

Moreover, $\det T = 1$. This is true because

$$\begin{aligned} Vol(rB^n) &= Vol(T(\mathcal{E})) \\ &= Vol(\mathcal{E})\,|\det T|, \end{aligned}$$

and $Vol(rB^n) = Vol(T(\mathcal{E}))$ implies the claim. This means that our lattice's covolume is equal to 1.

Finally, consider the lattice sphere packing $x + \frac{r}{2}B^n$. Then, its density is

$$\begin{aligned} \frac{Vol(rB^n)}{Vol(\mathbb{R}^n/L)} &= Vol\left(\frac{1}{2}T(\mathcal{E})\right) \\ &= \frac{1}{2^n}Vol(T(\mathcal{E})) \\ &= \frac{1}{2^n}|\det T|Vol(\mathcal{E}) \\ &= \frac{1}{2^n}cn^2, \end{aligned}$$

which is exactly what we wanted. □

This is very useful, because it means we can now explicitly find in any dimension n, a lattice sphere packing whose density is $cn^2 2^{-n}$.
Once again for the sake of not overcrowding this document, we haven't given an example but feel free to consult the code on GitHub.

3 Investigating methods to build good linear codes

As said in the introduction, we will investigate two different methods, simulated annealing (SA), and Rank 1 Random Noise (R1). A $[n, k]$ linear code can be represented by a generator matrix $G \in \mathcal{M}_{k\times n}(\mathbb{F}_2)$, and the set of codewords is the set of row vectors

$v = uG, u \in \mathbb{F}_2^n$, where u is a row vector. Both methods start with a randomly generated matrix $G \in \mathcal{M}_{k\times n}(\mathbb{R})$. They differ in the way they update the matrix.

For all methods, we fix $n = 24$, $k = 12$ as an example but one can input any n, k. This is taken as example as this is the extended Binary Golay Code, with known minimum distance $d_{\min} = 8$. Working on this "baby case" will enable us to see if our results have the potential to work on other, harder cases. Moreover, every matrix will have binary entries.

Please note that for the sake of clarity, the following SageMath codes have been simplified, but a complete version is available on GitHub.

3.1 Rank 1 Random Noise (R1)

This was one of the methods used by Maryna Viazovska, and the following code is hers (translated from Mathematica).

Again, start from a random matrix, built as the product of two matrices. The update decision here is based not on entropy nor hamming distance, but the number of vectors that have hamming weight less than an arbitrary weight. We call them *short vectors*. Update the current matrix by adding a random rank 1 *noise* matrix. For the updated and current matrix, compute the number of short vectors. If the updated matrix has less or the same amount of short vectors, accept the update and reiterate the process.

A priori, this doesn't have a "deep idea" behind it, it was a series of tests and this happened to yield the best results amongst them.

```
def R1(length, rank, iterations, min_dist_minus_one = 7) :

    """Start with a random matrix"""
    B = random_matrix(GF(2), rank, dim0, density = 0.5)
    C = random_matrix(GF(2), dim0, length, density = 0.5)
    while B*C == zero_matrix(GF(2), rank, length):          # ensure you start with a non-zero matrix
        B = random_matrix(GF(2), rank, dim0, density = 0.5)
        C = random_matrix(GF(2), dim0, length, density = 0.5)
    A = B*C

    """Start the process"""
    for j in range(iterations) :
        H = matrix(GF(2), A.right_kernel().basis())
        J = matrix(GF(2), H.right_kernel().basis())    # J is a basis of RowSpace(A)
        s = ShortVectors(J, min_dist_minus_one)

        N = RandomR(rank, length, 1)                     # to update matrix, add a random rank 1 noise matrix
        C = matrix(GF(2), (A+N).right_kernel().basis())
        D = matrix(GF(2), C.right_kernel().basis())     # D is a basis of RowSpace(A+N)
        s2 = ShortVectors(D, min_dist_minus_one)
        if s2 <= s :                                   # only update if updated matrix has less short vectors.
```

```
            A += N
        E.append(ShortVectors(J, min_dist_minus_one+1))

    """Return the generator matrix"""
    return A
```

Some optimisation in terms of execution time has been made, notably in the `ShortVectors` function :

```
    def ShortVectors(G, r) :
    """This returns number of vectors with weight <= r given a
        generator matrix G"""
    A = LinearCode(G).weight_distribution()
    s = 0
    for i in range(r+1) :
        s += A[i]
    return s
```

We ran this method for 100000 steps. With a large number of iterations (≥ 500), this consistently yields a matrix of rank 8 and minimum distance 8, which is known to be optimal (refer to this websitefor bounds on minimum distances). However, the issue with R1 is that the matrix's rank isn't 12, as desired. Hence, a few ideas come to mind to fix this issue.

The standard form for a generator matrix of length n and rank k is called the *systematic form*, and it is

$$G = \begin{pmatrix} I_k & G' \end{pmatrix}.$$

This is coherent because then, G is of rank k. Now, back to our ideas to fix the rank issue.

Idea 1. *How about we start with a matrix in systematic form, and add a noise matrix of the form $N = \begin{pmatrix} 0_k & R_{n-k} \end{pmatrix}$, where R_{n-k} is a random $n-k \times n-k$ rank 1 matrix ?*

This method was unsuccessful, as it almost always yielded a matrix of minimum distance $2-3$, even with a very high number of steps. However, another idea comes to mind :

Idea 2. *What if instead of only adding a single rank 1 noise matrix, we added more rank 1 noise matrices?*

Like before, we use Python to test this. Let's try running 50 steps, where for test i, we add i rank 1 noise matrices at each step, and the results are the following :

i	$d_{\min}$
5	4
10	3
15	2
20	2
25	2
30	3
35	2
40	3
45	3
50	3

Figure 5: Minimum distances observed when adding i rank 1 random matrices, $1 \leq i \leq 50$, 10 000 steps for each test.

We only display results for steps that are mutliples of 5, but results for other values are the same. Unfortunately, this idea isn't very good too. However, we've seen that inputting rank = 12 in R1 gives a matrix of rank 8, and the following idea comes :

Idea 3. *If inputting rank* = 12 *in the program consistently gives a code of dimension* 8*, can we input some rank* > 12 *which would consistently yield a matrix of rank* 12 *?*

We try this idea for $13 \leq \text{rank} \leq 30$, and the results are as follows :

entered rank	**returned rank**
12	8
13	8
14	8
15	8
16	8
17	8
18	8
19	8
20	8
21	8
22	8
23	9
24	8
25	8
26	8
27	8
28	8
29	8
30	8

Figure 6: Returned rank by R1, when entered rank ranges from 12 to 30. 10 000 steps.

This idea also doesn't seem like a good one. The initial rank always starts out at 7 and only climbs up to 8, always in the same way :

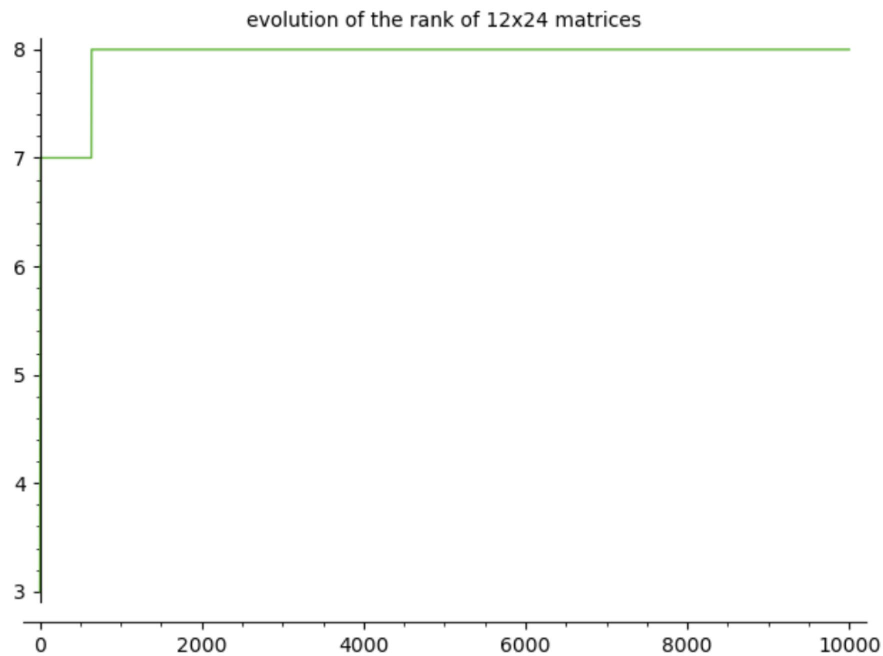


Figure 7: Evolution of the rank when inputting rank = 17, 10 000 steps.

So, maybe the problem comes from the initial matrix ?

Idea 4. *In R1, we start with the initial matrix* $R_{rank \times d_0} R'_{d_0 \times length}$*, where* $d_0 = 4$*. What if the rank problem comes from this ? Can we find a good value for* d_0 *that would fix the rank issue ?*

Let's try this for $1 \leq d_0 \leq 25$. We would like to obtain a matrix of rank 12. Here are the results :

d_0	**rank**	d_0	**rank**
1	9	14	12
2	8	15	11
3	8	16	12
4	8	17	11
5	8	18	11
6	8	19	10
7	9	20	11
8	8	21	12
9	9	22	12
10	10	23	11
11	10	24	11
12	11	25	11
13	11		

Figure 8: Returned rank by R1, when inputting rank 12, $1 \leq d_0 \leq 25$. 10 000 steps.

The issue seems resolved ! Now, let's check if the value $d_0 = 14$ consistently yields a matrix of rank 12. We chose $d_0 = 14$ because that implies a smaller matrix to calculate. Results for consistency :

Test	returned rank	d_{min}
1	12	2
2	12	2
3	12	3
4	12	2
5	12	2

Returned rank by R1, when inputting rank 12, and $d_0 = 14$. 10 000 steps.

The results are very consistent ! Now, two issues arise.

First, the minimum distances we are getting are very bad. Another issue is that we would like to find, for any given rank, a value of $d_0 = d_0(\text{rank})$ such that when inputting d_0, R1 yields a matrix of wanted rank. Could there be a formula for this ? Another possibility is to given a rank, randomly search for d_0 but this is extremely inefficient.

Although the summer project is over, we plan on looking for and testing ideas during the semester, and this document might not be up to date.

3.2 Simulated Annealing (SA)

Simulated annealing was the method used by my supervisor Amal Seddas and the following code is her's.

Start with a random matrix M_0 and define a score parameter `score = hamming_distance + entropy`. Now, to update the matrix, either flip a random bit, or swap a random pair of rows, each event being of probability 1/2. If the new matrix's score is higher than the current one, accept the update. If not, re-update the matrix.

```
1  def simulated_annealing_with_entropy(n, k, iterations=30000,
      init_temp=10.0, cooling_rate=0.9999):
2      """Simulated annealing to find binary codes with a good balance
           between minimum distance and entropy."""
3      G = random_matrix(GF(2), k, n)
4      current_code = LinearCode(G)
5      current_distance = current_code.minimum_distance()
6      current_entropy = entropy_weight_distribution(current_code)
7
8      best_code = current_code
9      best_distance = current_distance
10     best_entropy = current_entropy
11
12     temperature = init_temp
13
14     for i in range(iterations):
15         new_G = perturb_matrix(current_code.generator_matrix())
16         new_code = LinearCode(new_G)
17         new_distance = new_code.minimum_distance()
18         new_entropy = entropy_weight_distribution(new_code)
```

```

        new_score = new_distance + new_entropy
        current_score = current_distance + current_entropy

        if new_score > current_score or np.exp((new_score - current_score) / temperature) > random.random():
            current_code = new_code
            current_distance = new_distance
            current_entropy = new_entropy

            if new_distance > best_distance or (new_distance == best_distance and new_entropy > best_entropy):
                best_code = new_code
                best_distance = new_distance
                best_entropy = new_entropy

        temperature *= cooling_rate

    return best_code, (best_distance, best_entropy)
```

We also ran this method 100 times, each time for 100000 steps. Unlike R1, this method always returns a rank 12 (more generally, given k it returns a matrix of rank k). Hence, for the sake of comparison, we ran this method with $k = 8$ as this was the rank of the matrix given by R1. This method then consistently yields minimum distance 8 (if you run it for enough steps), which is also optimal. We still ran this method for $k = 12$, and 500000 steps, and it consistently gives a code of minimum distance 5-6, which isn't a very satisfying result, given that the optimal minimum distance is 8.

According to Seddas, this algorithm is optimal, so there isn't any point in trying to optimise it, unlike R1.

3.3 Random Sampling

This is a method used to see how good the two methods are compared to some random choice of generator matrices. Given a code length and a dimension, this generates a random matrix at each step. If the newly generated random matrix has a higher minimum distance, accept the update.

```
def random_sampling(length, dim, steps)
    for j in range(steps) :
                E = systematic_generator_matrix(length, rank)
                dis = LinearCode(E).minimum_distance()
                I.append(dis)
                if LinearCode(M_).minimum_distance() < dis :
                    M_ = E
```

Here, `systematic_generator_matrix(length, rank)` returns a matrix of rank `rank`, and which has the form $\begin{pmatrix} I_{k\times k} & M_{k\times n-k} \end{pmatrix}$. Every generator matrix can be written in the systematic form. Once again, we ran this with $k = 12$, and it consistently gave a code of minimum distance 7. If random sampling gives such a good minimum distance, either it is a good method, or the space of codes of length 24 and dimension 8 has plently of high minimum distance codes.

Let's test this method for $k = 12$ and 500 000 steps. This consistently gives $d_{\min} = 5$,

with the following distribution :

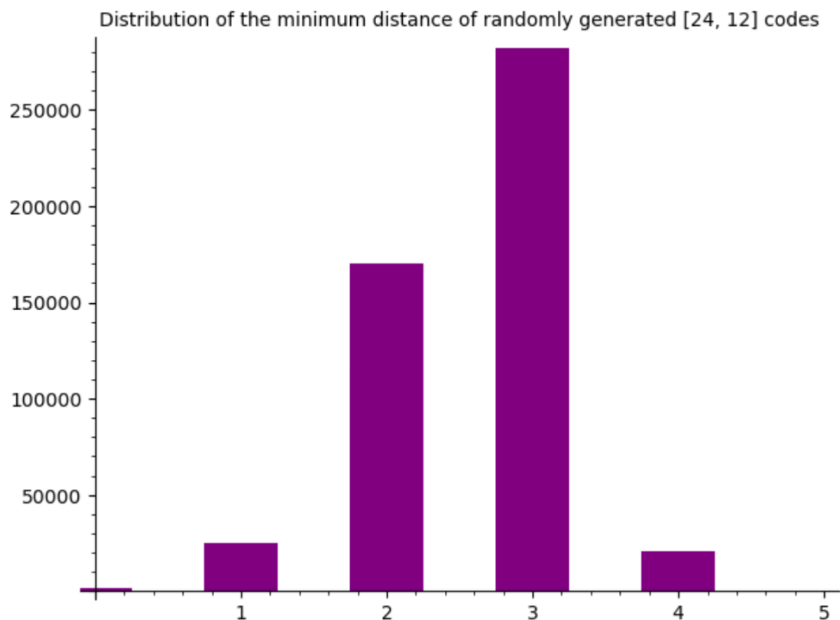


Figure 9: Distribution of minimum distance in 500 000 randomly sampled rank 12 12× 24 matrices

As expected, most randomly sampled matrices won't give a minimum distance close to the desired value of 8. Moreover, the matrix with the maximal minimum distance was reached after only 14 steps. Running 500 000 steps took about 4450 seconds, which is 1 hour and 15 minutes, or approximately 0.009 seconds per step.

3.4 Comparison

Now that we have given results for the methods, we will compare the three methods, based on a few criteria such as :

- Final minimum distance
- Execution time
- Consistency
- Rank (dimension)
- Step of last improvement

The step of last improvement is defined as the smallest step n_0 such that for any step $l > n_0$, $d_{\min}(l) \leq d_{\min}(n_0)$. Moreover, we will also use the Gilbert-Varshamov (GV) bound as a reference. This bound, given n, k, returns the minimum distance of an $[n, k]$ linear code as if it was "chosen randomly". The bound is given by the largest integer d such that

$$\sum_{k=0}^{d-2} \binom{n-1}{i} < 2^{n-k}.$$

Of course, we want our minimum distance to be at least the GV bound.

First, let's run all the tests for $n = 24$, $k = 12$. As said before, R1 will give a matrix of rank < 12, so for the sake of comparison we will run SA, RS, GV with $k =$ the rank of the generator matrix returned by R1. Let's run a test with 1 000 000 iterations. This yields :

Method	Rank	Min Dist
R1	9	8
SA	9	7
RS	9	7
GV	9	6

Figure 10: Results for methods R1, SA, RS, GV, $n = 24, k = 9$, 1 000 000 steps

It seems R1 is the best method in terms of achieving the highest minimum distance. Recall the $[24, 9]$ linear code has known optimal minimum distance 8, hence R1 is optimal. Surprisingly enough, random sampling achieved a very good minimum distance. Given that GV and RS both have returned pretty good results, it is safe to assume the space of $[24, 9]$ linear codes have many high minimum distance codes. On top of that, all methods are very consistent, meaning they return the same minimum distance, provided you run them for long enough, giving them enough time to explore the space of matrices.

Now, let's look at execution time :

Method	Total Execution Time	Execution Time per step
R1	7878.30 s	0.007878 s
SA	3538.07 s	0.003538 s
RS	6340.19 s	0.006340 s

Figure 11: Comparison of execution time, 1 000 000 steps, $n = 24, k = 9$

Clearly, SA is the best method in terms of execution time, being twice as fast as R1 per step. This will be useful when looking for very large codes. R1 is very inefficient, as running 1 million steps only enabled us to reach rank 9, which took more than 2 hours, whereas we wanted rank 12. Finally, let's have a look at the steps of last improvement for each method, when running 1 000 000 steps:

Method	n_0
R1	0
SA	23066
RS	47

Figure 12: Comparison of the step of last improvement for methods R1, SA, RS, $n = 24, k = 9$, 1 000 000 steps

This gives us an idea of "how fast each method reaches its final result", or "how efficient it is per step." A guess is that these values aren't consistent and vary a lot each time, and one might need to run many tests to compute an average value (or median, whatever...), but given that one needs many steps, one needs a lot of time (more than 6 hours to run each method once, so a few days of non-stop computation to compute an average (median)). Moreover, it's now safe to assume achieving a good minimum distance is easy for these parameters, given the little amount of steps it took to reach the final minimum distance for each method. An advantage of SA is that it always gives a code of any desired dimension. For instance, running SA and RS for $n = 24, k = 12$ gives

Method	Rank	Minimum Distance
SA	12	6
RS	12	6
GV	12	5

Figure 13: Comparison between SA, RS, GV, $n = 24, k = 12$, 1 000 000 steps

where the optimal minimum distance here is known to be 8 (extended binary Golay code). Currently, methods to get R1 (refer to 3.1) to construct a rank 12 code have returned codes with minimum distance around 3, which isn't a good result. There's room for improvement there.

So it seems that while we are unable to fix the issue with R1 not being able to fix the rank and return a high minimum distance consistently, SA is the most promising method, even though the minimum distance it yields usually isn't as good the one given by R1, when fixing the same rank for both methods.
However, should a method be given to fix the rank for R1, while giving a good minimum distance, then R1 would probably become the most promising method.

4 Acknowledgments

I am most thankful to my supervisor Amal Seddas for guiding me through this unforgettable experience. Her guidance, support and enthusiasm were crucial. I am also very grateful for the opportunity I was given by Maryna Viazovska, director of the Number Theory Lab, and for supervising me and advising me when needed. I also want to thank the Laidlaw Foundation, and the Laidlaw team at EPFL for I wouldn't have had this

internship without all their support.

A special thanks goes out to my long-time friend and fellow scholar Bruno Siniciali, without whom I might never have had the chance to join the Laidlaw program.

References

[1] Brownian motion, August 2025. Page Version ID: 1308037588.

[2] Boaz Klartag. Lattice packing of spheres in high dimensions using a stochastically evolving ellipsoid, April 2025. arXiv:2504.05042.

[3] C. A. Rogers. Existence Theorems in the Geometry of Numbers. *The Annals of Mathematics*, 48(4):994, October 1947.