5-2022

A GPU Accelerated Genetic Algorithm for the Construction of Hadamard Matrices

Raven I. Ruiz
The University of Texas Rio Grande Valley

A GPU ACCELERATED GENETIC ALGORITHM FOR THE CONSTRUCTION OF HADAMARD MATRICES

A Thesis

by

RAVEN I. RUIZ

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Mathematics

The University of Texas Rio Grande Valley

May 2022

A GPU ACCELERATED GENETIC ALGORITHM FOR THE CONSTRUCTION OF HADAMARD MATRICES

A Thesis
by
RAVEN I. RUIZ

COMMITTEE MEMBERS

Dr. Andras Balogh
Chair of Committee

Dr. Dambaru Bhatta
Committee Member

Dr. Tamer Oraby
Committee Member

Dr. Zhijun Qiao
Committee Member

May 2022

ABSTRACT

Ruiz, Raven I., A GPU accelerated Genetic Algorithm for the Construction of Hadamard Matrices. Master of Science (MS), May, 2022, 63 pp., 8 tables, 15 figures, references, 28 titles.

Hadamard matrices are square matrices with $+1$ and -1 entries and with columns that are mutually orthogonal. The applications include signal processing and quantum computing. There are several methods for constructing Hadamard matrices of order 2^k for every positive integer k. The Hadamard conjecture proposes that there are also Hadamard matrices of order $4k$ for every positive integer k. We use a genetic algorithm to construct (search for) Hadamard Matrices. The initial population of random matrices is generated to have a balanced number of $+1$ and -1 entries in each column. Several fitness functions are implemented exploiting the basic matrix property that $Q^T Q$ is diagonal if and only if the columns of matrix Q are orthogonal. The crossover process creates offspring matrix population by exchanging columns of the parent matrix population. The mutation process flips $+1$ and -1 entry pairs in random columns, several methods were implemented to achieve this. The use of CuPy library in Python on graphics processing units enables us to handle populations of thousands of matrices and matrix operations in parallel.

DEDICATION

I dedicate this thesis to my parents Araceli Ruiz and Rigoberto Ruiz Sr., to my siblings Rigoberto Ruiz Jr. and Andrew A. Ruiz, and to my significant other Adanary Ramirez. All of my accomplishments are thanks to the constant support of my loved ones, they're my source of motivation and happiness. They have blessed me with their continued support in everything I have pursued, for that I am grateful.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my family for accommodating all my needs throughout my college career. To my significant other, Adanary Ramirez for supporting me emotionally and inspiring me everyday. Adanary contributed to this project as an undergraduate student in 2019. She implemented the Simulated Annealing Algorithm, which was later built upon.

I would also like to send my sincere gratitude to my committee chair, Dr. Andras Balogh. He introduced me to Hadamard matrices while I was an undergraduate and I did my final project on this topic with his guidance. As a graduate student, Dr. Balogh became my research advisor and we continued our research upon this project. Dr. Balogh has taught me a considerable amount over the years, without his guidance and enthusiasm, this paper would not have been possible, thank you.

In addition, I would like to send my gratitude to Dr. Dambaru Bhatta, Dr. Tamer Oraby, and Dr. Zhijun Qiao for being my committee members. My deepest appreciation to Dr. Zhijun Qiao, who recommend me for the Dean's Graduate Assistantship Award and encouraged me to enter the graduate program.

A special thank you to the UTRGV College of Sciences for honoring me with the College of Sciences Dean's Graduate Assistantship Award. This prestigious award provided me the opportunity to continue my education and pursue greater achievements.

Lastly, a thank you to Dr. Cristina Villalobos, who has inspired and guided me since I entered the mathematics program as an undergraduate. Dr. Villalobos taught me as her student and as her teaching assistant, I appreciate everything you have done for me.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Hadamard Matrices

1.1.1 Definition and Basic Properties

Hadamard matrices are $m \times m$ square matrices with $+1$ and -1 entries. Their rows or columns are mutually orthogonal, meaning that any pair of rows or columns are perpendicular to each other. In this paper, we will pursue column orthogonality. When achieving column orthogonality, it follows that each Hadamard matrix must have a balanced number of $+1$ and -1 entries in each column except for the first. In this case, the first column consists of all $+1$ entries since it is perpendicular to each other column.

In Figure 1.1, we demonstrate what an 8×8 Hadamard matrix could look like. In Figure 1.2, we represent the same Hadamard matrix, but in tile form. The $+1$ entries in Figure 1.1 are represented as white tiles and the -1 entries are represented as black tiles. Throughout this paper, we will use both representations.

There are two important properties of Hadamard matrices that will be taken advantage of to create functions that determine if a matrix is an Hadamard matrix. In this section, we will define

$$\begin{pmatrix} + & + & + & - & - & - & + & - \\ + & - & - & + & + & - & + & - \\ + & + & - & + & - & - & - & + \\ + & + & - & - & + & + & - & - \\ + & - & - & - & - & + & + & + \\ + & - & + & - & + & - & - & + \\ + & + & + & + & + & + & + & + \\ + & - & + & + & - & + & - & - \end{pmatrix}$$

Figure 1.1: An 8×8 Hadamard Matrix

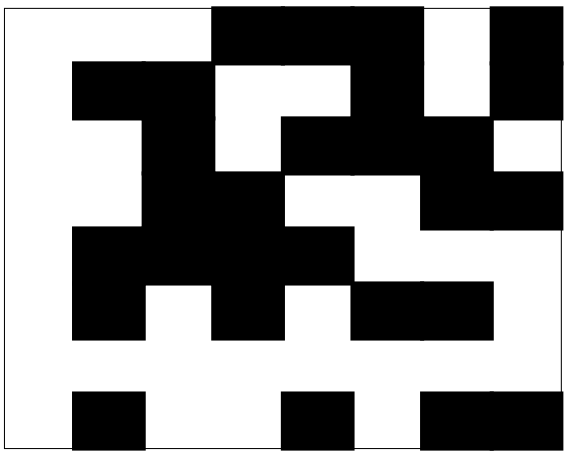


Figure 1.2: Representing an 8×8 Hadamard matrix using black and white tiles

one of these functions and explain how it is used. The two properties are given as follows

1. If H is an $m \times m$ Hadamard matrix, then $H^T H = mI_m$.

2. If H is an $m \times m$ Hadamard matrix, then $\det(H) = \pm m^{m/2}$.

Let us consider the first property, due to its orthogonality, an Hadamard matrix H gives us the following form

$$H^T H = \begin{pmatrix} m & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & m \end{pmatrix} = mI_m. \tag{1.1}$$

where H^T is the transpose of H and I_m is an $m \times m$ identity matrix. In general, an $m \times m$ matrix Q with a balanced number of $+1$ and -1 entries have the following form:

$$Q^T Q = \begin{pmatrix} m & * & \cdots & * \\ * & \ddots & & \vdots \\ \vdots & & \ddots & * \\ * & \cdots & * & m \end{pmatrix} \tag{1.2}$$

where Q^T is the transpose of Q and $*$ is an arbitrary constant. Now, we define the following function:

$$F(Q) = \sum_{i,j} \left|Q^T Q\right| - m^2 \tag{1.3}$$

where $\sum_{i,j} \left|Q^T Q\right|$ is the sum of entries in $|Q^T Q|$ and m^2 is the size of the matrix Q. By use of the first property, if Q is an Hadamard matrix, then $\sum_{i,j} \left|Q^T Q\right| = m^2$ which implies $F(Q) = 0$. If Q is not an Hadamard matrix with a balanced number of $+1$ and -1 entries, then $\sum_{i,j} \left|Q^T Q\right| > m^2$ which implies $F(Q) > m^2$. Therefore, we can define the function as follows

$$F(Q) = \sum_{i,j} \left|Q^T Q\right| - m^2 \geq 0. \tag{1.4}$$

The purpose of this function is to measures how close a matrix Q is to being an Hadamard matrix. A matrix Q is an Hadamard matrix if and only if $F(Q) = 0$. While Q is not an Hadamard matrix, $F(Q) > 0$. This function was implemented into two algorithms, in Chapter III it was implemented as an energy defining function and in Chapter IV it was implemented as a fitness function. This is one of several fitness functions implemented. Additional functions will be defined and explained in Section 4.3.

Now let us consider the second property, if H is an Hadamard matrix, then

$$|\det(H)| = m^{m/2} \tag{1.5}$$

where $\det(H)$ is the determinant of H. If Q is an $m \times m$ matrix with a balanced number of $+1$ and -1, then using the Hadamard inequality by Jacques Hadamard (Hadamard 1893), we have that

$$|\det(Q)| \leq m^{m/2} \tag{1.6}$$

where $|\det(Q)| < m^{m/2}$ while Q is not an Hadamard matrix and notice that $m^{m/2}$ is maximal among all possible Q matrices. In Section 4.3, we will define a fitness function using this property.

1.1.2 Literature

Hadamard matrices were first discovered by James Sylvester in 1867 (Sylvester 1867) who found matrices of order 2^k where $k \in \mathbb{N}$. In 1893, Jacques Hadamard introduced the Hadamard Conjecture (Hadamard 1893), stating that an Hadamard matrix exists when $m = 1$, $m = 2$, and $m = 4k$ where $k \in \mathbb{N}$. This conjecture covers matrices of sizes 12×12, 20×20, 24×24, and other multiples of 4 which are not covered in James Sylvester's matrices of order 2^k. Deterministic algorithms are known for creating Hadamard matrices of order 2^k and only a few other Hadamard matrices of order $4k$.

There are other publications that have made substantial progress in the study of Hadamard matrices using several methods. In 1970, J.M. Goethals and J.J. Seidel (Goethals and Seidel 1970) constructed a Theorem that proves there exist skew Hadamard matrices of order $m = 36$ and $m = 52$. In 2014, the Goethals Siedel method was later implemented on a software to construct 32 in-equivalent Hadamard matrices of order $m = 404$ by A. Jayathilake, A. Perera, and M. Chamikara (A. Jayathilake, Perera, and Chamikara 2014). In 2019, A. Mohammadian and B. Tayfeh-Rezaie delves into the classification of Hadamard matrices using types of quadruples of rows with two distinct values (Mohammadian and Tayfeh-Rezaie 2019). In 1972, M. Plotkin proposes that each Hadamard matrix of order mn is decomposable into m components for $m = 4$ or $m = 8$ and achieves a decomposition $D(24, 8)$ for a matrix of order 24 (Plotkin 1972). In 1992, J. Seberry and M. Yamada highlights important Hadamard matrix theorems, use several methods to construct Hadamard matrices, display results, and analyze their findings from each method (Seberry and Yamada 1992). In 2016, Andriyan Suksmono developed a Simulated Annealing Algorithm to construct Hadamard matrices (A. Suksmono 2016). There have been several other approaches implemented in the construction of Hadamard matrices. However, to the best of our knowledge no new Hadamard matrices resulted from these approaches, perhaps due to the computational limitations on CPUs.

The following is a list of currently know constructions results for Hadamard matrices of different orders (see (Browne et al. 2021)).

1. 2^t for $t \geq 0$ (Sylvester 1867).

2. $p^a + 1$ where p is prime and $p^a \equiv 3 \text{mod} 4$ (Paley 1933).

3. $2(p^a + 1)$ where p is prime and $p^a \equiv 1 \text{mod} 4$ (Paley 1933).

4. $p(p+2)+1$ where p and $p+2$ are twin primes (Stanton and Sprott 1958).

5. $4p^{4t}$ where p is prime and $t \geq 1$ (Xia 1992).

6. $4t$ for all values of $t \leq 250$ except for $t \in \{167, 179, 223\}$ (Kharaghani and Tayfeh-Rezaie 2005).

7. $n = ab/2$ or $n = abcd/16$ where a, b, c, d are orders of Hadamard matrices (Craigen, Seberry, and Zhang 1992; Seberry and Yamada 1992).

8. There exist constants α and β such that, if t is an odd positive integer, then there exists a Hadamard matrix of order $2^{\lceil \alpha + \beta \log_2(t) \rceil} t$; see (Craigen 1995; Seberry 2017).

1.1.3 Applications

Hadamard matrices are used in applications such as signal processing and quantum computing. Signal processing is a branch of electrical engineering that focuses on analyzing and modifying signals. There are several types of signals used in daily life such as communication channels, radio, and movie. Signals serve as a batch of information, when these signals are being sent and received they must be transformed. The purpose of this transformation is to verify that the original signal sent was received as intended. In 2013, an article by Chathranee Jayathilake, A.A.I Perera, and M.A.P. Chamikara was published which uses the Walsh-Hadamard transformation that decomposes a signal into a set of Walsh functions (C. Jayathilake, Perera, and Chamikara 2013). Walsh functions (Walsh 1923) are defined using Sylvester's Hadamard matrices (Sylvester 1867) and the purpose of the Walsh-Hadamard transformation is to remove noise from the signals that are sent and received. The Walsh-Hadamard transformation leads to more applications such as power spectrum analysis, filtering, and error correcting code.

Quantum computing is a type of computation process that uses quantum state properties to perform these computations. Quantum computers are needed to perform these computations as they

are thousands of times faster than regular computers. In 2019, an article by Andriyan Suksmono and Yuichiro Minato was published which implemented a quantum annealing machine to find Hadamard matrices (A.B. Suksmono and Minato 2019).

1.1.4 Thesis Objective

In this work we develop a genetic algorithm for the construction of Hadamard matrices. The parallel numerical code uses CUDA GPUs to accelerate the computations.

1.1.5 Thesis Organization

In Section 1.2 we describe how Genetic Algorithms work. In Chapter II we describe the difference between traditional CPU computing and our approach of using GPUs to accelerate computations. In Chapter III we describe a previous stochastic approach by Suksmono (A. Suksmono 2016) that we implemented on the GPU. In Chapter IV we describe our parallel implementation of a Genetic Algorithm for finding Hadamard matrices. In Chapter V we discuss results found by use of this Genetic Algorithm.

1.2 Genetic Algorithm Method

1.2.1 Natural Selection

Genetic algorithms are a search heuristic that was inspired by the Natural Selection process (Wirsansky 2020). Natural Selection is a process where individuals adapt and change based on the environment and other variables. Each individual in the population is unique, meaning that each individual has different traits. Some individuals may have better traits than others, this allows those with better traits to live longer and reproduce. Those superior traits are then passed down to the next generation, with some variation, this is known as evolution. Evolution is a key component of genetic algorithms, it allows the algorithms to search for the optimal solution of several problems in mathematics. We will continue to discuss evolution in the following section.

1.2.2 Darwin's Evolution Theory

Genetic algorithms use Darwin's evolution theory of natural selection as a bases to search for optimal solutions. There are three main points behind Darwin's theory of evolution that are given as follows

1. *Inheritance*: Each individual inherits traits that were passed down from their parents. The traits that are most likely to be passed down are those that will improve chances of survival.
2. *Variation*: Each individual in a population will have variation that is unique to them, even those that are related to each other.
3. *Fitness and Selection*: The most fit individuals are more likely to survive, the surviving individuals are then able to reproduce and pass on their genes to future generations.

Ultimately, Darwin's evolution theory suggests that the individuals with the best traits will survive and maintain the population with offspring of their own. The offspring will most likely be better equipped for survival and each generation there after will become more adaptive. This leads to a genetic algorithm that will be used throughout this paper.

1.2.3 Genetic Algorithm

1. *Generate Initial Population*: Create set of individuals in a population.
2. *Compute Fitness*: A numerical measure of how close an individual is to becoming 'fit'.
3. *REPEAT*
    (a) *Selection*: Select the most fit individuals which is based on the fitness.
    (b) *Crossover*: The selected individuals become parents, the parents are paired and create a pair of offspring that inherit traits from the parents.
    (c) *Mutation*: Each offspring will have some form of variation to them.
    (d) *Compute Fitness*: A numerical measure of how close an individual is to becoming 'fit'.
4. *UNTIL Population has converged*: An individual has met the required fitness.

Table 1.1: Order given a Schema Example

Schema	Order
$****$	0
$***0$	1
$**10$	2
$1*10$	3
1110	4

1.2.4 Fundamental Theorem of Genetic Algorithms

The fundamental theorem of genetic algorithms (Bridges and Goldberg 1987) is also know as Holland's schema theorem (Holland 1975) which was proposed by John Holland in the 1970's. While the mathematics is quite involved, we will focus on the basic understanding of the theorem. The theorem suggests that an individual will prevail if it has an above average fitness. First, some terminology

1. A *schema* is a binary string with entries $\{0,1,*\}$ where $*$ can take on any value of either 0 or 1. For example consider the binary string $*0*101$, then the possible binary strings are

$$*0*101 = \{101101, 100101, 001101, 000101\}. \tag{1.7}$$

2. The *order* of a schema is the number of fixed values. For example consider Table 1.1

3. The *defining length* of a schema is the distance between the furthest fixed values. For example consider Table 1.2

The schema theorem states that the schema with above average fitness is said to have a low order and a small defining length.

Table 1.2: Defining Length given a Schema Example

Schema	Defining Length
$****$	0
$***0$	0
$**10$	1
$*1*0$	2
$*110$	2
$1*10$	3
1110	3

CHAPTER II

COMPUTING ON THE GRAPHICS PROCESSING UNITS

2.1 CPU vs GPU

Stochastic algorithms and matrix calculations involve large number of calculations. A CPU (Central Processing Unit) does calculations mostly in serial way, which can take a large amount of time if working with large number of large matrices. GPUs (Graphics Processing Units) were developed for fast graphics rendering by calculating what to do and when with the millions of pixels in a computer screen. Since a GPU can do thousands of calculations simultaneously (in parallel), the time working with large number of matrices can be reduced significantly. The use of GPUs allows us to do parallel calculations more effectively than using CPUs. Most of the calculations have been performed on an EVGA GeForce RTX 3080 XC3 black which is shown in Figure 2.1, (Han and Rochford 2020), it has 8704 CUDA cores, 10 GB memory, and CUDA capability 8.6. The technical specifications is listed in Table 2.1. With the use of GPUs our genetic algorithm was able to handle for example a population of $40,000$ matrices of size 20×20. Clearly, the number of operations needed to work with so many matrices exceeds 8704, which is the number of CUDA cores even for the most basic matrix operations. Hence, not all calculations are done simultaneously, but the GPU can distribute the work over the threads much more effectively than the CPU with its limited number of cores.

2.2 CuPy vs. Numpy

NumPy (Harris et al. 2020) is a popular library for the Python programming language. It supports overloaded mathematical functions operation on multi-dimensional arrays. For example, if A is a 100×100 matrix defined as `A=numpy.random.rand(100,100)`, then `B=numpy.sin(A)`

Figure 2.1: EVGA GeForce RTX 3080 XC3 black

Table 2.1: Technical Specifications per Compute Capability

Technical Specifications	Compute Capability (8.6)
Maximum number of resident grids per device	128
Maximum dimension of a grid of thread blocks (x, y, z)	$(2^{31} - 1, 65535, 65535)$
Maximum dimensionality of a thread block (x, y, z)	$(1024, 1024, 64)$
Maximum number of threads per block	1024

produces a matrix containing the sine of the elements of matrix A. The calculations of the $10,000$ elements are done one-by-one, in serial fashion on the CPU.

For this project, we are using CuPy (Okuta et al. 2017), which is a Python library accelerated with NVIDIA CUDA (NVIDIA, Vingelmann, and Fitzek 2020) for GPUs. It was created specifically to be highly compatible with NumPy. For example modifying the previously mentioned NumPy matrix example to `A=cupy.random.rand(100,100)` and `B=cupy.sin(A)`, the calculations are distributed on the thousands of CUDA threads to be done in parallel.

Using the CuPy library and in general CUDA on the GPU involves the following steps. It is important to note that the CPU controls the GPU too.

1. Data is copied from CPU memory to GPU memory.
2. CPU initiates the calculations on the GPU.
3. GPU executes the calculations.
4. Results are copied back from GPU to CPU.

Each of these steps require time. The speed up of the GPU execution has to be significant in order to balance the extra time required by copying the data back and forth between CPU and GPU.

We include here a simple speed comparison of using Numpy (serial calculations) vs. CuPy (parallel calculations). Since our actual code with Genetic Algorithm includes raw kernels, we never wrote a serial version of it, but due to the complicated nature of the calculations with large multi-dimensional arrays, we expect the parallel computations to be hundreds of times faster than the serial code. The example creates a list of $N = 10,000$ random matrices of size 100×100, and then multiplies each with their transform, repeating the calculation M times.

NumPy version:

```
import numpy as np
m=100
N=10**4
```

```
Pop = np.random.rand(N, m, m)
for i in range(M):
    F=np.matmul(np.transpose(Pop,axes=(0,2,1)),Pop)
```

CuPy version:

```
import cupy as cp
m=100
N=10**4
Pop=cp.random.rand(N, m, m)
for i in range(M):
    F=cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop)
```

Note the identical syntax other than the use of "cp" for CuPy vs. "np" for NumPy. The time required for the calculations are summarized in Table 2.2 and in Figure 2.2. For $M = 1$, when the matrix calculations are done only once, the CuPy and NumPy calculations take the same 3.9 seconds. Although the GPU does the calculation faster than the CPU, the time it takes to load the GPU slows the completion of the code down. For repeated calculations of the batched matrix multiplications the Numpy code execution time increases by about 2.12 seconds for each additional loop steps taken. The CuPy calculation only increases by about 0.07 seconds for each loop steps. This means that the CuPy operation is 30 times faster than the NumPy operation after ignoring the initial data copy between the CPU and GPU.

It is also possible to combine CuPy code with CUDA kernel function that use C++ syntax. We use several so-called "raw kernels" when we need more complicated operations than simple matrix-vector ones. This requires to know how threads are arranged in grids of blocks, as explained in the next section.

Table 2.2: CuPy vs. NumPy Execution Times (seconds)

M	NumPy	CuPy
1	3.9	3.862
2	6.2	3.994
3	8.6	4.038
4	10.7	4.095
5	13.0	4.175
6	14.6	4.252
7	17.0	4.287
8	19.2	4.330
9	21.6	4.394
10	23.0	4.454
⋮	⋮	⋮
100	218.6	9.916

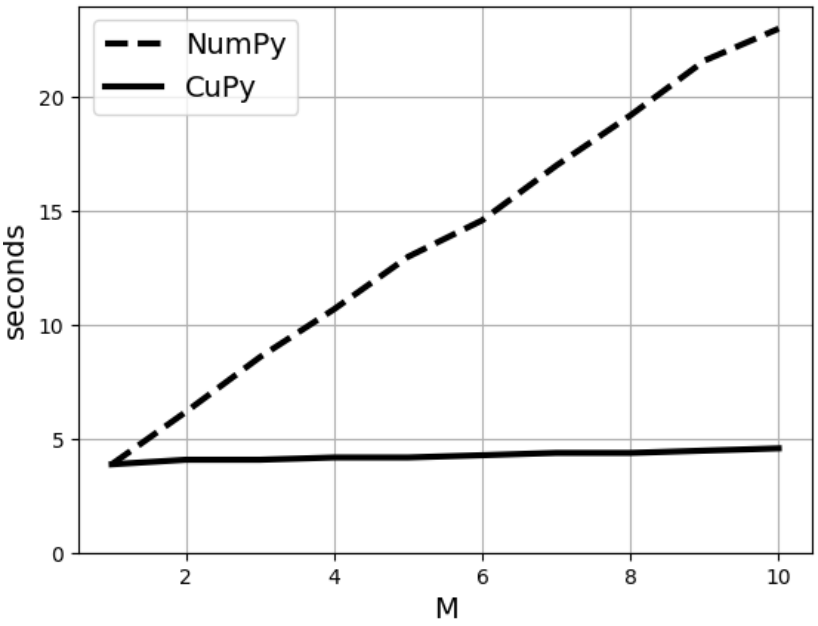


Figure 2.2: Cupy vs. Numpy Execution Times (seconds)

2.3 Blocks, Grids, and Threads

Without going deep into the technical description of how the GPU architecture and the CUDA programming model works, it is necessary to introduce it briefly, because not all of our calculations can be done effectively using the NumPy-compatible CuPy operations in basic vector-matrix format. For more complicated operations we use CUDA kernel functions where operations are executed in parallel different CUDA threads. The CUDA threads are grouped into CUDA blocks, and the CUDA blocks are grouped into a grid. These can be arranged in 1, 2, or 3-dimensions. While the basic CuPy commands automatically arranges the operations over certain number of grids and blocks and threads, we have to do this ourselves in the case of more complicated operations. More complicated CUDA kernel functions are presented later in our work. Here we only demonstrate with a simple example the arrangement of grids and blocks. As a simple 1-dimensional example, consider adding together random vectors a and b, each of length $n = 10,000$ and the result is stored in vector c. The maximum number of threads per block is typically 1024 in today's Nvidia GPUs. The $n = 10,000$ elements need then $\lceil n/1024 \rceil$ number of grids in order to have enough threads. Then the CUDA kernel function receives the three arrays, their size, and the block and grid arrangements. The corresponding CuPy code is

```
import cupy as cp
import math
n = 10000
a = cp.random.rand(n)
b = cp.random.rand(n)
c = cp.zeros(n)
blocksizex = 1024
blocks = (blocksizex, 1, 1)
grids = (math.ceil(n/blocksizex), 1, 1)
addvectors(grids, blocks, (a, b, c, n))
```

In the kernel code the threads are indexed using the built-in 3D variable `threadIdx` and the blocks are indexed using the built-in 3D variable `blockIdx`. Since the total number of threads reserved `math.ceil(n/blocksizex)*blocksizex` is larger then the vectors length n, we use an `if (i<n)` statement in the kernel in order to avoid going out of bound with the index. The corresponding kernel code follows.

```
addvectors = cp.RawKernel(r'''
extern "C" __global__
void addvectors(double *a, double *b, double *c, const int n){
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if (i<n){ c[i]=a[i]+b[i]; }
}
''', 'addvectors')
```

CHAPTER III

PREVIOUS APPROACH

3.1 Simulated Annealing Algorithm

In a previous project, we were inspired by Andriyan Suksmono's (A. Suksmono 2016) work which uses a method called The Simulated Annealing Algorithm with a Metropolis update criteria on an ising model. The Simulated Annealing is a stochastic algorithm to find global minimum of a function. This process is similar to the annealing of metals where the metal is heated up and then slowly cooled down in order to reduce its hardness. The Simulated Annealing Algorithm goes as follows:

1. Start by randomly selecting a Q matrix with balanced $+1$ and -1 entries in each column except for the first, the first column consists of all $+1$ entries.

2. For a matrix Q, we define its energy as (shown in Section 1.1.1):

$$E(Q) = \sum_{i,j} |Q^T Q| - m^2 \geq 0. \tag{3.1}$$

3. While $E(Q) > 0$ we randomly flip $+1$ and -1 entries from random columns. This is done by rearranging a pair of rows.

   (a) If the energy decreases then we accept the change and accept the new Q matrix.

   (b) If the energy does not decrease, the change is not accepted and the flipping continues.

   (c) If $E(Q) = 0$, then Q is a Hadamard matrix.

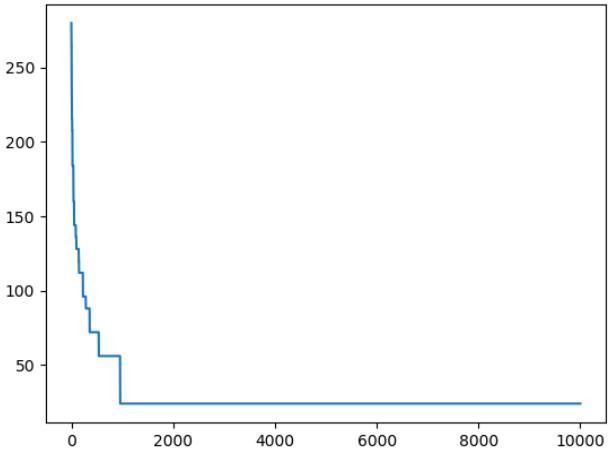


Figure 3.1: Energy of matrix vs number of iterations

The problem with the algorithm on its own is that a matrix can get stuck in a bad configuration such that the matrix does not show anymore improvement. We show an example of this in the Figure 3.1.

In Figure 3.1, between 0 to 1000 iterations. the energy decreased from an energy of approximately 250 to an energy of approximately 15. Then for the next 9000 iterations, the energy stayed the same. This problem happened a numerous amount of times, even for smaller matrices. A solution to this problem is called The Metropolis-Hasting method.

The Metropolis-Hasting method uses a probability as a function of time such that we accept a new matrix with some probability even if its energy is not smaller in-order to avoid getting stuck in a bad configuration. We used two probability functions, the first being the following exponential function:

$$P = 0.5e^{-Ct} \quad \text{where } C \text{ is a constant.} \tag{3.2}$$

We chose this exponential function because it approaches zero (shown in Figure 3.2), which is exactly what we want for the energy. The metropolis constant C is a value that is chosen. After many tests of working with this metropolis constant we found that this constant value depended on the size of the matrix we are working with. If C is too large, then the process might get stuck. If C is too small, then the convergence will be too slow. The second probability function that we chose was the following:

$$P = \frac{0.5}{Ct+1} \quad \text{where } C \text{ is a constant.} \tag{3.3}$$

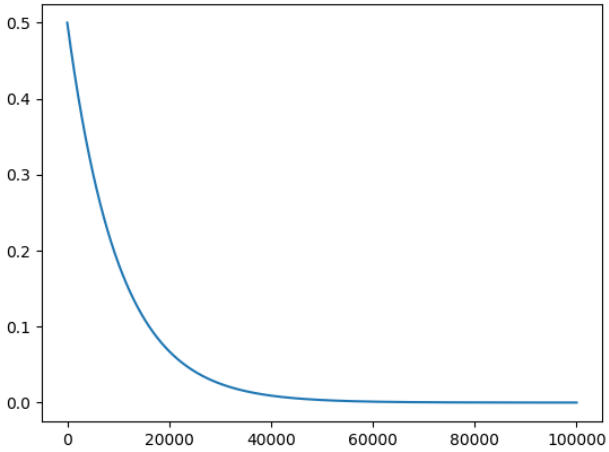


Figure 3.2: Probability as a function of time

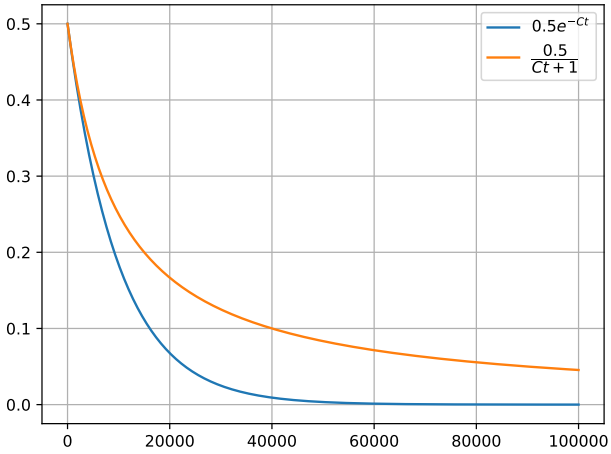


Figure 3.3: Probability as a function of time comparison

This $1/t$ function was chosen since it approaches zero slower than the exponential function. In theory, this would mean we accept more flips even if the energy does not decrease. The hope is that the algorithm would get stuck less often and it would yield better results than the exponential function did. In Figure 3.3, we compared the two functions using the metropolis constant $C = 10^{-4}$ such that

$$P = 0.5e^{-Ct} \text{ vs. } P = \frac{0.5}{Ct+1} \quad \text{for } C = 10^{-4}. \tag{3.4}$$

After many tests with the $1/t$ function we saw no improvement. We did not get stuck but we also did not find any Hadamard matrices due to the convergence being too slow. Overall, the exponential function seemed to be better than the $1/t$ function.

Using the Simulated Annealing Algorithm, the largest Hadamard matrix found was of the size 16×16. When attempting to find larger matrices, the energy would get stuck too often. In addition, choosing the "correct" metropolis number became tedious as it depends on the size of the matrix. Moreover, this implementation of the algorithm allowed us to only work with one matrix at a time. For these reasons, we implemented a Genetic Algorithm that works with multiple matrices at a time as well as other reasons. The details of this Genetic Algorithm will be discussed in Chapter IV and its results in V.

CHAPTER IV

GENETIC ALGORITHM AND HADAMARD MATRICES

4.1 Preface

In Section 1.2, we discussed the idea of the genetic algorithm method. To summarize, genetic algorithms are a search heuristic which uses the idea of Natural Selection and Darwin's evolution theory. These algorithms find and select the fittest of individuals in a population to reproduce offspring for future generations. Genetic algorithms are useful for finding optimal solutions of several mathematics problems. In this section, we will discuss how this algorithm can be implemented to search for Hadamard matrices. We will use the following algorithm

1. *Generate Initial Population*
2. *Compute Fitness*
3. *REPEAT*
    (a) *Selection*
    (b) *Crossover*
    (c) *Mutation*
    (d) *Compute Fitness*
4. *UNTIL Population has converged*

In Section 4.2, we detail how an initial population of matrices is generated. For this process, two methods were implemented and we discuss their differences. In Section 4.3, we use a fitness function to compute the fitness of each matrix in a population. These fitness values are used as a

measurement to see how close a matrix is to becoming an Hadamard matrix. There are four fitness functions that were implemented and we discuss their differences. In Section 4.4, we use the fitness values of each matrix to select parent matrices from a population. In Section 4.5, we use a process that creates offspring matrices from selected parent matrices. In Section 4.6, we discuss how each offspring matrix is given some variation or mutation. Three methods were implemented that achieve this.

4.2 Population

This algorithm starts by randomly generating an initial population of $4N$ matrices. The initial population is a 3-dimensional array that has the form:

$$\text{Initial Population} = [Q_1, \ldots, Q_N, Q_{N+1}, \ldots, Q_{2N}, Q_{2N+1}, \ldots, Q_{3N}, O_{3N+1}, \ldots, Q_{4N}]. \tag{4.1}$$

Here, $Q_1, \ldots, Q_{2N}$ are $m \times m$ matrices with a balanced number of $+1$ and -1 entries in each column except for the first. The first column consists of all $+1$ entries. In addition, $Q_{2N+1}, \ldots, Q_{4N}$ are $m \times m$ matrices with entries consisting of all $+1$ entries. There are three reasons for this:

1. The matrices with all $+1$ entries will have a larger fitness value than the matrices that have a balanced number of $+1$ and -1 entries. As a result, these matrices won't be selected to become parents during the selection process.

2. During the crossover process, all the matrices consisting of $+1$ entries will be overridden by the offspring matrices.

3. The calculations are cut in half, hence the time it takes to fulfill these calculations is shorter.

After the initial population is generated, it is modified by means of the Selection process which selects parent matrices and then the Crossover process which creates offspring matrices. Consequently, after the first iteration of the algorithm, the population will have the form:

$$\text{Population} = [P_1, \ldots, P_N, P_{N+1}, \ldots, P_{2N}, O_1, \ldots, O_N, O_{N+1}, \ldots, O_{2N}] \tag{4.2}$$

where $P_1, \ldots, P_{2N}$ and $O_1, \ldots, O_{2N}$ are $m \times m$ matrices with a balanced number of $+1$ and -1 entries in each column except for the first. The first column consists of all $+1$ entries. We define $P_1, \ldots, P_{2N}$ as the parent matrices and $O_1, \ldots, O_{2N}$ as the offspring matrices. The details of how we define these parent and offspring matrices will be discussed in Sections 4.4 and 4.5.

In this section, we implemented two methods of generating the initial population that was detailed earlier and has the form given in equation (4.1). The first method randomly generates the initial population completely through the CPU. The second method generates an initial population that done in parallel on the GPU. The details of what is being done in the CPU and GPU will be discussed later in this section.

4.2.1 Initial Population by means of the CPU

The first method creates a 3-dimensional initial population of $4N$ matrices of sizes $m \times m$. The first $2N$ matrices are randomly generated to have a balanced number of $+1$ and -1 entries in each column except for the first. The first column consists of all $+1$ entries. The last $2N$ matrices consist of all $+1$ entries. This was detailed at the beginning of the section. This implementation of the initial population is completely done through the CPU and is coded as follows:

```
Pop = cp.ones((4*N, m, m), dtype=cp.int8)
for n in range(2*N):
    for j in range(1,m):
        Pop[n, :, j] = cp.sign(cp.random.permutation(m)*2-(m-1))
```

We start with the line `Pop = cp.ones((4*N, m, m), dtype=cp.int8)` which creates a population of $4N$ matrices of sizes $m \times m$ with $+1$ entries. We use a `for n in range(2*N)` loop that refers to the first $2N$ matrices of the population. Within this for loop, we use another `for j in range(1,m)` loop that refers to each column (except the first) of the $2N$ matrices. Within both for loops, we use the line `Pop[n,:,j]=cp.sign(cp.random.permutation(m)*2-(m-1))` that creates balanced number of random $+1$ and -1 entries in each column (except the first) of the $2N$ matrices.

For further explanation, the population `Pop`, uses the indexing `[n,:,j]` to specify the matrices and columns that will be operated upon. In this case, it was specified in the double for loop which was explained earlier. For `cp.sign(cp.random.permutation(m)*2-(m-1))`, the CuPy function `cp.random.permutation(m)` creates an m array with entries that has a permutation range between 0 and $m-1$. Multiplying this by 2, we have a array of all even numbers. Subtracting by $(m-1)$ we have an array with $m/2$ positive integer and $m/2$ negative integers. Then, using the CuPy function `cp.sign(...)`, it return an array with with a balanced number of $+1$ and -1 entries. This is done for $2N$ matrices and for each column (except the first). We could have also used the random perturbation (shuffle) of a predetermined array with balanced ± 1 entries.

The result is a randomly generated initial population that has the form given in equation (4.1). The issue with this method is that it is being done in two serial for loops. As mentioned in Section 2.1, serial calculations can take a large amount of time if working with large number of large matrices. For this algorithm, we do exactly that. We quickly discovered that our algorithm spent more time on creating the initial population than on the rest of the Genetic Algorithm. For this reason, we implemented a second method that does these calculations in parallel on the GPU.

4.2.2 Initial Population by means of the GPU

Similar to the first method, the second method creates a 3-dimensional initial population of $4N$ matrices of sizes $m \times m$. The first $2N$ matrices are randomly generated to have a balanced number of $+1$ and -1 entries in each column except for the first. The first column consists of all $+1$ entries. The last $2N$ matrices consist of all $+1$ entries. This implementation of the initial population is done in parallel on the GPU and is coded as follows

```
Pop = cp.ones((4*N, m, m), dtype=cp.int8)
Pop[0:2*N,0:2*k,1:m] = -1
seed = int.from_bytes(os.urandom(4), 'big')
Shuffle_Column(grids, blocks,(Pop, seed, m, N))
```

We start with the line `Pop = cp.ones((4*N, m, m), dtype=cp.int8)` which creates a

population of $4N$ matrices of sizes $m \times m$ with $+1$ entries. Then we modify the population, using `Pop[0:2*N,0:2*k,1:m] = -1`, this sets specific entries of matrices of the population equal to -1. We specify which entries to modify by using `[0:2*N,0:2*k,1:m]` which refers to the first $2N$ matrices, the first $m/2$ rows, and for all columns except for the first. Hence, for the first $2N$ matrices and each column except the first, the line `Pop[0:2*N,0:2*k,1:m] = -1` changes the first $m/2$ rows to -1 entries. As a result, the first $2N$ matrices will have a balanced number of $+1$ and -1 entries in each column except the first. For the first $2N$ matrices, the first $m/2$ rows will consist of -1 entries and the last $m/2$ rows will consist of all $+1$ entries.

The line `seed = int.from_bytes(os.urandom(4), 'big')` returns the integer represented by the given array of bytes. The input `os.urandom(4)` returns a string of size 4 which represents random bytes and the input `'big'` determines the byte order used to represent the integer, in this case, the most significant byte is at the beginning of the byte array.

Now we must randomize the order of these entries for each column. We achieve this by implementing a raw kernel function called `Shuffle_Column`. Raw kernel functions were briefly discussed in Chapter II. The idea of the shuffling algorithm is based on the Fisher-Yates shuffle (Fisher and Yates 1938) which was later implemented for computers by Richard Durstenfeld in 1964. The shuffling algorithm that we have implemented is known as the modern version of the Fisher-Yates shuffle (Fisher and Yates 1938; Durstenfeld 1964). While the shuffling in each column is done in serial, the columns are handled simultaneously in parallel. The `Shuffle_Column` kernel function is coded as follows:

```
Shuffle_Column = cp.RawKernel(r'''
#include <curand_kernel.h>
extern "C" __global__
void Shuffle_Column(char *Q, int seed, const int m, const int N)
{
    int k = blockDim.x*blockIdx.x+threadIdx.x;
    int i = blockDim.y*blockIdx.y+threadIdx.y;
```

```
    int j = blockDim.z*blockIdx.z+threadIdx.z;
    unsigned long int seq, offset;
    int i1,i2; char ti1,ti2;
    seed=seed+k; seq = j; offset = 0; curandState h;
    if ((k<2*N)&&(j<m)&&(i==0)){
        curand_init(seed,seq,offset,&h);
        for(i1=0; i1<m-1; i1++){
            i2=(int)(curand_uniform(&h)*(m-i1)+i1);
            ti1=Q[m*m*k+m*i1+j];
            ti2=Q[m*m*k+m*i2+j];
            Q[m*m*k+m*i1+j]=ti2;
            Q[m*m*k+m*i2+j]=ti1;
        }
    }
}
''', 'Shuffle_Column', backend='nvcc')
```

The `Shuffle_Column` kernel function has the following inputs:

- The input `char *Q` is the `Pop` consisting of $4N$ matrices with balanced $+1$ and -1 entries in each column where the first $m/2$ rows consist of -1 entries.
- The input `int seed` is given by `int.from_bytes(os.urandom(4), 'big')` which returns the integer represented by the given array of bytes.
- The input `const int m` is an integer that is represented by the size of the matrices in `Pop`.
- The input `const int N` is an integer such that `Pop` has $4N$ matrices.

In the Raw Kernel function `int k` is the index of matrices, `int i` is the index of rows, and `int j` is the index of columns. The shuffling steps are distributed over all columns of all parent matrices,

but by setting $i == 0$ we prevent the distributed over each row. Instead, a for loop goes through all the rows from 0 to $m-2$, and the entry $i1$ is switched with entry $i2$, where $i2$ is a random index between $i1$ and $m-1$. This shuffles the columns from lowest index to highest.

4.3 Fitness Function

In the natural selection process, the most fit person will most likely survive. For this genetic algorithm, the fitness function measures how close a matrix is to becoming an Hadamard matrix. In this section, we test four fitness functions which are stated as follows:

$$F_1 = \sum_{i,j} \left| Q^T Q \right| - m^2 \geq 0. \tag{4.3}$$

$$F_2 = \text{nonzero}\left(Q^T Q\right) - m \geq 0. \tag{4.4}$$

$$F_3 = m^{m/2} - |\det(\text{Population})| \geq 0. \tag{4.5}$$

$$F_4 = \frac{1}{\text{nonzero}\left(Q^T Q\right) - m + 1} \leq 1. \tag{4.6}$$

where Q is an $m \times m$ matrix within a population. For the functions F_1, F_2, and F_3, we are computing the minimum fitness value to find the Hadamard matrix, in these cases, the minimum fitness value is 0. For the function F_4, we are computing the maximum fitness value to find the Hadamard matrix, in this case, the maximum fitness value is 1. In Chapter V, we will compare the minimizing fitness functions and explain which fitness function yielded the best results in a speed comparison test.

Each function yields the fitness of each matrix in the population which has the form:

$$F = [f_1, \ldots, f_N, f_{N+1}, \ldots, f_{2N}, f_{2N+1}, \ldots, f_{3N}, f_{3N+1}, \ldots, f_{4N}] \tag{4.7}$$

where $f_1, \ldots, f_{2N}$ and $f_{2N+1}, \ldots, f_{4N}$ refers to the fitness of $P_1, \ldots, P_{2N}$ and $O_1, \ldots, O_{2N}$ in equation (4.2), respectively. The purpose of testing these four fitness functions is to evaluate which one yields the best results.

4.3.1 First Fitness Function

The first fitness function is given as follows:

$$F_1 = \sum_{i,j} \left| Q^T Q \right| - m^2 \geq 0. \tag{4.8}$$

Notice that it is the same as equation (3.1) used in the previous algorithm, Chapter III. However, in the simulated annealing algorithm, we are only working with one matrix at a time. In this algorithm, we are working with $4N$ matrices at a time. The idea is it gives us a higher chance of finding an Hadamard matrix, maybe even several at a time. The fitness function in equation 4.8 is coded as follows:

```
F=cp.sum(cp.absolute(cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop)),
    axis=(1,2))-m2
```

The operation on the matrices are done in parallel batches. The 3-dimensional array `Pop` was described in the population form (4.2). For the sake of understanding the use of the axes in Python, `Pop` has the following form

$$\texttt{Pop} = \overbrace{\left[1 \underbrace{\left\{ [Q_1], \ldots, [Q_{4N}] \right.}_{2} \right]}^{0} \tag{4.9}$$

where 0,1, and 2 represent a permutation of the dimensions. We use this information to perform some basic matrix operation for each of the matrices in `Pop`. First, consider the following:

```
cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop)
```

The CuPy function `cp.transpose(Pop,axes=(0,2,1))` returns a transpose for each of the matrices in `Pop`. Here, the `axes` is changed from `(0,1,2)` to `(0,2,1)`. This means that the entries in the 1st dimension are exchanged with the entries in the 2nd dimension. Therefore, the

result is Q^T. The CuPy function `cp.matmul(...)` returns the product of two matrices. This implies that the line `cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop)` returns the operation Q^TQ. Continuing upon the operation, we have the following

```
F=cp.sum(cp.absolute(...), axis=(1,2))-m2
```

The CuPy function `cp.absolute(...)` returns an elementwise absolute value of each entry for every matrix in `Pop`. The CuPy function `cp.sum(...,axis=(1,2))` uses the `axis` to specify along which dimension the sum is taken. The sum taken is the `axis=(1,2)`, this refers back to the dimensions in equation (4.9). Meaning, for each matrix, the absolute sum of the entries are calculated. Consequently, we have the operation $\sum_{i,j}\left|Q^TQ\right|$. Subtracting `m2`, which is just m^2, we have the fitness function F_1 given in equation (4.8). The final result of this is an array of fitness values that was demonstrated to have the form (4.7).

4.3.2 Second Fitness Function

The second fitness function was implemented in an attempt to get faster results as opposed to the previous function. The second fitness is given as follows:

$$F_2 = \text{nonzero}\left(Q^TQ\right) - m \geq 0. \tag{4.10}$$

Equation (4.10) was implemented using property the first property from Section 1.1.1. That is, If Q is an Hadamard matrix, it has the following form:

$$Q^TQ = \begin{pmatrix} m & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & m \end{pmatrix} = mI_m. \tag{4.11}$$

If Q is an $m \times m$ matrix with a balanced number of $+1$ and -1 entries then,

$$Q^T Q = \begin{pmatrix} m & * & \cdots & * \\ * & \ddots & & \vdots \\ \vdots & & \ddots & * \\ * & \cdots & * & m \end{pmatrix} \tag{4.12}$$

The idea of the fitness function F_2 given in equation (4.10) is if Q is an Hadamard matrix, then the number of non-zero entries in $Q^T Q$ is the same as the size of the matrix, that is, nonzero$(Q^T Q) = m$. This was easily derived from the property given in equation (4.11). Moreover, if Q is an $m \times m$ matrix with a balanced number of $+1$ and -1 entries in each column and is not an Hadamard matrix, then nonzero$(Q^T Q) > m$. This results in equation (4.10) and is coded as follows:

```
F=cp.count_nonzero(cp.matmul(cp.transpose(Pop, axes=(0,2,1)), Pop),
   axis=(1,2))-m
```

Here, the matrix operations `cp.matmul(cp.transpose(...))` was explained in Section 4.3.1. The CuPy function `cp.count_nonzero(...)` counts the non-zero entries of each of the matrices in the population. As mentioned previously, if nonzero$(Q^T Q) = m$, then Q is an Hadamard matrix. The final result of this is an array of fitness values that was demonstrated to have the form (4.7).

4.3.3 Third Fitness Function

The third fitness function was implemented in an attempt to get faster results as opposed to the previous functions. The third fitness is given as follows:

$$F_3 = m^{m/2} - |\det(\text{Population})| \geq 0. \tag{4.13}$$

This function was created using the second property of Hadamard matrices that was mentioned in Section 1.1.1. The property states that, if H is an $m \times m$ Hadamard matrix, then $\det(H) = \pm m^{m/2}$.

From Section 1.1.1, we reduced this property using the Hadamard Inequality, that is, if Q is an $m \times m$ matrix with a balanced number of $+1$ and -1 in each column, then

$$|\det(Q)| \leq m^{m/2}. \tag{4.14}$$

If Q is an $m \times m$ Hadamard matrix, then

$$|\det(Q)| = m^{m/2}. \tag{4.15}$$

This results in equation (4.13) and is coded as follows:

```
F = md-cp.absolute(cp.linalg.det(Pop))
```

The variable `md` $= m^{m/2}$. The CuPy function `cp.linalg.det(Pop)` returns an array of determinants from each of the matrices in the population. Then, the CuPy function `cp.absolute(...)` takes the absolute value of each determinant in the array. The final result of this is an array of fitness values that was demonstrated to have the form (4.7).

4.3.4 Fourth Fitness Function

The fourth fitness function was introduced to compare the results of using a minimum fitness value and a maximum fitness value. The first 3 fitness functions were implemented in search of a minimum fitness value, the fourth fitness function searches for a maximum fitness value. The fourth fitness is given as follows:

$$F_4 = \frac{1}{\text{nonzero}\,(Q^T Q) - m + 1} \leq 1. \tag{4.16}$$

Notice that equation (4.16) is similar to equation (4.10) such that

$$F_4 = \frac{1}{F_2 + 1} \leq 1. \tag{4.17}$$

where $\min(F_2) = 0$ and so $\max(F_4) = 1$. For this function, a matrix Q is an Hadamard matrix if and only if $F_4 = 1$. This results in equation (4.16) and is coded as follows:

```
F=1/(cp.count_nonzero(cp.matmul(cp.transpose(Pop, axes=(0,2,1)),Pop),axis =
    (1,2))-m+1)
```

The line `cp.count_nonzero(...)` was described in Section 4.3.2. The other operations are quite obvious and follow that of equation (4.16). Unfortunately, after testing this maximizing function, it was found that the minimizing functions produced better results.

4.4 Selection

The purpose of the selection process is to select matrices that are more likely to become Hadamard matrices. The selected matrices will take the parent position of the population and will be of the form

$$\text{Population} = [P_1, \ldots, P_{2N}, Q_1, \ldots, Q_{2N}] \tag{4.18}$$

where $P_1, \ldots, P_{2N}$ are the selected parent matrices and $Q_1, \ldots, Q_{2N}$ are matrices that were not selected. Note that in first iteration of the selection process $Q_1, \ldots, Q_{2N}$ consist of matrices with all $+1$ entries and they are guaranteed not to be selected, this was discussed in Section 4.2. The selected parent matrices will be paired up to create a pair of offspring matrices which will be discussed in Section 4.5

In this section, we developed two selection processes. The first process selects matrices by using the direct fitness values from equation (4.7). The second process selects matrices by creating a probability function using the fitness values from equation (4.7).

4.4.1 Selection Without Probabilities

The selection process without probabilities is coded as follows:

```
Pop[0:2*N,:,:] = Pop[cp.random.permutation(cp.argsort(F)[:2*N]),:,:]
```

The CuPy function `cp.argsort(F)` returns the indices of the fitness array F given in equation (4.7) that correspond directly to the population in (4.2). The order of the indices returned depend on the fitness values of each matrix in the population. The indices will be sorted

from the least fitness value to the greatest fitness value. For the selection part, we only care to use the most fit individuals, that is, we only want the matrices that are closest to becoming Hadamard matrices. The indices of the selected matrices that we want are located in the first half of `cp.argsort(F)`, which will be defined as the indices of the parent matrices. This is accomplished using `cp.argsort(F)[:2*N]`. Then, the CuPy function `cp.random.permutation(...)` randomly permutes these indices to guarantee a random order of the parent matrices. Finally, using `Pop[0:2*N,:,:]=Pop[...,:,:]`, this moves the matrices with the best fitness value to the parent position in random order which is from 0 to $2N$. This results in the population of the form (4.18).

Consider the following example, let $N = 2$ and $m = 20$ such that the population has 8 matrices of sizes 20×20 with a balances number of $+1$ and -1 entries in each column. Assume that the fitness values are calculated by F_2 given in equation (4.10) and is given as follows:

$$\texttt{F} = [238, 220, 234, 236, 234, 220, 228, 212]. \tag{4.19}$$

To sort the fitness value indices from least to greatest, we use `cp.argsort(F)` such that

$$\texttt{cp.argsort(F)} = [7, 1, 5, 6, 2, 4, 3, 0]. \tag{4.20}$$

To save only the most fit half into the parent position index wise, we use `cp.argsort(F)[:4]` such that

$$\texttt{cp.argsort(F)[:4]} = [7, 1, 5, 6]. \tag{4.21}$$

Although we have the indices of the most fit matrices, it is in increasing order, we want to randomize the order of this using `cp.random.permutation(...)` such that

$$\texttt{cp.random.permutation(...)} = [6, 7, 5, 1]. \tag{4.22}$$

Then, using `Pop[0:4,:,:]=Pop[...,:,:]`, we have defined 4 matrices with that have superior fitness at the parent position. We now have a population that is of the form (4.18).

The next selection process uses a probability function to select the parent matrices. We will discuss and compare the two methods in the next section.

4.4.2 Selection With Probabilities

The selection process with probabilities is coded as follows:

```
idx = np.arange(4*N)
p = F/cp.sum(F)
C = np.random.choice(idx,2*N,replace=False,p=p.get())
Pop[0:2*N,:,:] = Pop[C,:,:]
```

The CuPy function `C = np.random.choice` generates a random sample from a one-dimensional array. The output array is `C` and the inputs are `idx`, `size`, `replace`, and `p`. In this case:

- The input `idx` is a one-dimensional array. Using `np.arange(4*N)`, it returns evenly spaced integer values which results in the following array

$$\text{idx} = [0, 1, 2, \ldots, 4N-1]\,. \tag{4.23}$$

- The input `size` describes the shape of the output. This implies the output `C` is a $2N$ array.
- The input `replace` determines whether the sample is with or without replacement. In this case, we are choosing without replacement.
- The input `p` is an array of probabilities that are associated with each entry in the `idx` given by equation (4.23). We define that probability function p as

$$p = \frac{F_i}{\sum_{i=1}^{4N} F_i} = \left[\frac{F_1}{\sum_{i=1}^{4N} F_i}, \ldots, \frac{F_{4N}}{\sum_{i=1}^{4N} F_i}\right] \tag{4.24}$$

  where the p is an array whose probabilities correspond to the fitness form (4.7).

The line `Pop[0:2*N,:,:]=Pop[C,:,:]` randomly selects the parent matrices with some probability based on equation (4.24) and results in the population of the form (4.18). Comparing the two selection processes, there was no improvement when using the probabilities. The hope of this probability method was to give matrices, with higher fitness values, the opportunity to create offspring matrices and hopefully the offspring will have an improved fitness value. Nonetheless, the probabilities had no impact on whether or not an Hadamard matrix was found.

4.5 Crossover

The purpose of the crossover is to create a pair of offspring matrices from a pair of parents matrices, these parents were selected using the selection process in section 4.4. The crossover process is coded as follows:

```
col = cp.random.random_integers(1,m-1,N)
Crossover(grids, blocks, (Pop, col, m, N))
```

The CuPy function `col = cp.random.random_integers(1,m-1,N)` returns an array of N random integers between the values of 1 and $m-1$. Each integer in this array will be known as the crossover points. Notice that there are $2N$ parent matrices and N crossover points. So each pair of parent matrices will be assigned a crossover point. A crossover point splits each pair of parent matrices into two parts and uses parts from each parent matrix to create a pair of offspring matrices.

For example, consider a pair of 4×4 parent matrices with a randomly generate crossover point `col = 3`. The parent matrices with the appropriate crossover point is given as follows:

$$P_1 = \left(\begin{array}{ccc|c} - & + & + & + \\ - & + & - & - \\ + & - & + & - \\ + & - & - & + \end{array}\right), \quad P_2 = \left(\begin{array}{ccc|c} - & + & + & + \\ + & - & + & + \\ - & - & - & - \\ + & + & - & - \end{array}\right) \tag{4.25}$$

Using the crossover point `col = 3`, we create the following offspring matrices as follows

$$O_1 = \left(\begin{array}{ccc|c} - & + & + & + \\ - & + & - & + \\ + & - & + & - \\ + & - & - & - \end{array}\right), \quad O_2 = \left(\begin{array}{ccc|c} - & + & + & + \\ + & - & + & - \\ - & - & - & - \\ + & + & - & + \end{array}\right) \tag{4.26}$$

This example demonstrates what the raw kernel function `Crossover(grids, blocks, (Pop, col, m, N))` accomplishes. However, the `Crossover` function does this for a pair of N parent matrices, each with an assigned crossover point, and it will create a pair of N offspring matrices. The result is a population with $2N$ parent matrices and $2N$ offspring matrices that hold characteristics from the parents. The `Crossover` is a kernel function that is given as follow:

```
Crossover = cp.RawKernel(r'''
extern "C" __global__
void Crossover(char *Q, const int* col, const int m, const int N)
{
    int k = blockDim.x*blockIdx.x+threadIdx.x;
    int i = blockDim.y*blockIdx.y+threadIdx.y;
    int j = blockDim.z*blockIdx.z+threadIdx.z;
    if (k<N){
        if((j<col[k]) && (i<m)){
            Q[m*m*(k+2*N)+m*i+j]=Q[m*m*k+m*i+j];
            Q[m*m*(k+3*N)+m*i+j]=Q[m*m*(k+N)+m*i+j];
       }
        if((j>=col[k]) && (j<m) && (i<m)){
            Q[m*m*(k+3*N)+m*i+j]=Q[m*m*k+m*i+j];
            Q[m*m*(k+2*N)+m*i+j]=Q[m*m*(k+N)+m*i+j];
        }
```

```
    }
}
''', 'Crossover')
```

The `Crossover` kernel function creates $2N$ offspring matrices using $2N$ parent matrices and returns an array `Pop` with size $4N$ such that

$$\texttt{Pop} = [P_1, \ldots, P_N, P_{N+1}, \ldots, P_{2N}, O_1, \ldots, O_N, O_{N+1}, \ldots, O_{2N}] \tag{4.27}$$

and has the following inputs:

- The input `char *Q` is the `Pop` consisting of $4N$ matrices with balanced $+1$ and -1 entries in each column (except the first).
- The input `const int* col` is given by `cp.random.random_integers(1,m-1,N)` and returns an array of N random integers between the values of 1 and $m-1$. As discussed earlier, this is the crossover point.
- The input `const int m` is an integer that is represented by the size of the matrices in `Pop`.
- The input `const int N` is an integer such that `Pop` has $4N$ matrices.

In the raw kernel function we define `int k` as the index of matrices, `int i` as the index of rows, and `int j` as the index of columns.

We use an `if (k<N)` statement since the indexing of the parent matrices are given by $P_1, \ldots, P_N$ and $P_{N+1}, \ldots, P_{2N}$, both of size N. This is also done in order to avoid going out of bound with the indexing, this was discussed in Section 2.3. To create the offspring matrices, we have to consider the crossover point. As demonstrated in the previous example, the crossover point splits the parent matrices into two parts. Within the previous if statement, we use two more if statements to denote the parent matrices into two parts using the crossover point and create the offspring matrices.

- Consider the `if((j<col[k])&& (i<m))` statement, this specifies that the index of columns are before the crossover point. We also must set a boundary for the index of rows to be less than the size of the matrices, this is used so the index does not exceed the number of rows for each matrix. Using these conditions, the following is executed:

```
Q[m*m*(k+2*N)+m*i+j]=Q[m*m*k+m*i+j];
Q[m*m*(k+3*N)+m*i+j]=Q[m*m*(k+N)+m*i+j];
```

  In this case, the crossover point is used to denote the first parts of the parent matrices. In the first line, we copy the first part of $P_1,\ldots,P_N$ into $O_1,\ldots,O_N$. In the second line, we copy the first part of $P_{N+1},\ldots,P_{2N}$ into $O_{N+1},\ldots,O_{2N}$.

- Consider the `if((j>=col[k])&& (j<m)&& (i<m))` statement, this specifies that the index of columns are at the crossover point or after the crossover point. We also must specify a boundary for the index of columns and rows to be less than the size of the matrices. Using these conditions, the following is executed:

```
Q[m*m*(k+3*N)+m*i+j]=Q[m*m*k+m*i+j];
Q[m*m*(k+2*N)+m*i+j]=Q[m*m*(k+N)+m*i+j];
```

  In this case, the crossover point is used to denote the second parts of the parent matrices. In the first line, we copy the second part of $P_1,\ldots,P_N$ into $O_{N+1},\ldots,O_{2N}$. In the second line, we copy the first part of $P_{N+1},\ldots,P_{2N}$ into $O_1,\ldots,O_N$.

It is obvious to see that each offspring matrices inherited parts from each of the parent matrices which is exactly what we wanted to achieve. We now have a population that consist of $2N$ parent matrices and $2N$ offspring matrices as shown in the population form (4.2). In the next section, we will discuss the mutation of the offspring matrices.

4.6 Mutation

The purpose of the mutation process is for each offspring matrix to have some kind of variation to them. The mutation of these offspring matrices is done by switching a pair of $+1$ and -1 entries while maintaining the balance of $+1$ and -1 entries in each column. In this section, three methods were implemented to achieve variation among offspring matrices.

4.6.1 First Mutation Function

This method flips a pair of $+1$ and -1 entries by randomly selecting a column (except the first) and randomly selecting two different row indices. For every offspring matrix, the randomly selected column and row indices are the same. The Mutation is coded as follows:

```
rowindx1 = np.random.randint(m)
rowindx2 = np.random.randint(m)
colindx = np.random.randint(1,m)
while rowindx1 == rowindx2: rowindx2 = np.random.randint(m)
Mutation1(grids, blocks, (Pop, rowindx1, rowindx2, colindx, m, N))
```

For `rowindx1` and `rowindx2`, the CuPy function `np.random.randint(m)` returns a random integer between the values 0 and $m-1$, this is the indexing for the two rows. Similarly, the Cupy function `colindx = np.random.randint(1,m)` returns a random integer between the values 1 and $m-1$, meaning the first column (which has index 0) cannot be selected. A while loop is then used anytime the row indices are the same, while this is true, we recalculate the second row index. Finally, the `Mutation1` kernel function is called and is given as follows:

```
Mutation1 = cp.RawKernel(r'''
extern "C" __global__
void Mutation1(char *Q, const int row1, const int row2, const int col,
   const int m, const int N)
{
```

```
    int k = blockDim.x*blockIdx.x+threadIdx.x;
    int i = blockDim.y*blockIdx.y+threadIdx.y;
    int j = blockDim.z*blockIdx.z+threadIdx.z;
    if ((k>=2*N) && (k<4*N)){
        if(((i==row1) || (i==row2)) && (j==col) && (Q[m*m*k+m*row1+col] !=
           Q[m*m*k+m*row2+col])){
            Q[m*m*k+m*row1+col] *= -1;
            Q[m*m*k+m*row2+col] *= -1;
        }
    }
}
''', 'Mutation1')
```

For $2N$ offspring matrices, the `Mutation1` kernel function flips the sign of two entries with opposite sign in a column. The column and row indices are randomly generated and are the same for each offspring matrix. The inputs are given as follows:

- The input `char *Q` is the `Pop` consisting of $4N$ matrices with balanced $+1$ and -1 entries in each column.
- The input `const int row1` is the first row index that is an integer between 0 and $m-1$.
- The input `const int row2` is the second row index that is an integer between 0 and $m-1$.
- The input `const int col` is the column index that is an integer between 1 and $m-1$.
- The input `const int m` is an integer that is represented by the size of the matrices in `Pop`.
- The input `const int N` is an integer such that `Pop` has $4N$ matrices.

In the raw kernel function we define `int k` as the index of matrices, `int i` as the index of rows, and `int j` as the index of columns. The `if ((k>=2*N)&& (k<4*N))` statement sets a

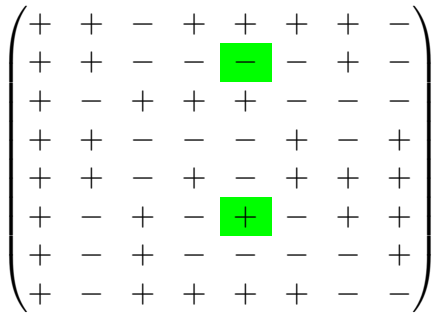
boundary for k such that only the offspring matrices are mutated.. For the actual mutation process, an if statement is given as follows:

```
if(((i==row1) || (i==row2)) && (j==col) && (Q[m*m*k+m*row1+col] !=
   Q[m*m*k+m*row2+col])){
           Q[m*m*k+m*row1+col] *= -1;
           Q[m*m*k+m*row2+col] *= -1;
       }
```

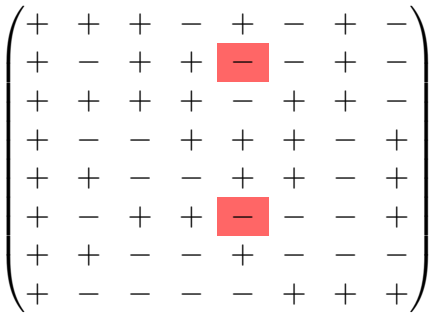
This if statement checks that `row1` or `row2` is in the index `i` and checks if `col` is in the index `j`. For the condition `Q[m*m*k+m*row1+col] != Q[m*m*k+m*row2+col]`, this checks that the entries have opposite sign given `row1`, `row2`, and `col`. If these conditions are met, then `Q[m*m*k+m*row1+col] *= -1` and `Q[m*m*k+m*row2+col] *= -1`. This multiplies the two entries by -1, in other words, we flip the sign. For each matrix, there is no mutation if this condition is not met.

To demonstrate how the `Mutation1` kernel function works, lets consider an example. Let $m = 8$ and $N = 2$ such that the we have a population of 8 matrices that have the sizes 8×8 with a balanced number $+1$ and -1 in each column. This implies there are 4 offspring matrices that will be mutated. Let `rowindx1 = 1`, `rowindx2 = 5`, and `colindx = 4` be the randomly generated indices. The mutation of the 4 offspring matrices are given in Figure 4.1. In parts (a), (c), and (d) the green highlights represent that a mutation occurred since the signs are opposite. In part (b) the red highlights represent that a mutation does not occur since the signs are not opposite.

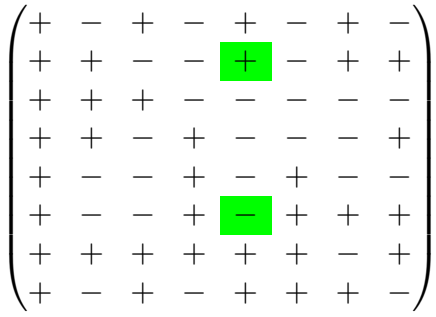
The disadvantage of this method is it mutates the same column and row indices for every offspring matrix given the restrictions. Implying that each offspring matrix was either not mutated at all or they have the same mutation as another offspring matrix. Additionally, when working with larger matrices, the flipping of just two entries may not be good enough, the convergence may be too slow. The largest Hadamard matrix found using this method was of the size 12×12. To improve upon creating variation among each offspring matrix, we implemented a second mutation function



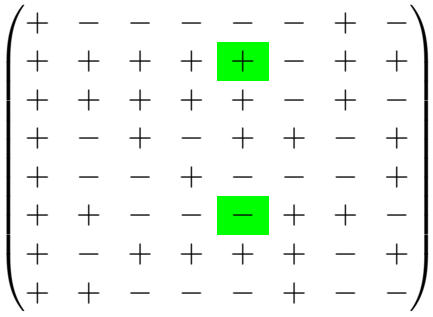
(a) First Offspring Matrix



(b) Second Offspring Matrix



(c) Third Offspring Matrix



(d) Fourth Offspring Matrix

Figure 4.1: Example of the First Mutation Method

that achieves this.

4.6.2 Second Mutation Function

The previous mutation function either mutated the same column and pair of row indices or no mutation took place. This method flips a pair of $+1$ and -1 entries by randomly selecting a column array of indices (except the first) and randomly selecting two row arrays of indices. In this section, we achieve variation among each offspring matrix. This mutation method is coded as follows:

```
rowindx1 = cp.random.random_integers(0,m-1,size=(2*N))
rowindx2 = cp.random.random_integers(0,m-1,size=(2*N))
colindx = cp.random.random_integers(1,m-1,size=(2*N))
Mutation2(grids, blocks, (Pop, rowindx1, rowindx2, colindx, m, N))
```

For `rowindx1` and `rowindx2`, the CuPy function `cp.random.random_integers(0, m-1, size=(2*N))` returns a $2N$ array with entries containing the integer values between 0 and

$m-1$. For `colindx`, the CuPy function `cp.random.random_integers(1,m-1,size=(2*N))` returns a $2N$ array with entries containing the integer values between 1 and $m-1$. Finally, the `Mutation2` kernel function is called and is given as follows:

```
Mutation2 = cp.RawKernel(r'''
extern "C" __global__
void Mutation2(char *Q, const int* row1, const int* row2, const int* col,
   const int m, const int N)
{
    int k = blockDim.x*blockIdx.x+threadIdx.x;
    int i = blockDim.y*blockIdx.y+threadIdx.y;
    int j = blockDim.z*blockIdx.z+threadIdx.z;
    if ((k>=2*N) && (k<4*N)){
        if((i==0) && (j==col[k-2*N])){
            if (Q[m*m*k+m*row1[k-2*N]+j] != Q[m*m*k+m*row2[k-2*N]+j]){
                Q[m*m*k+m*row1[k-2*N]+j] *= -1;
                Q[m*m*k+m*row2[k-2*N]+j] *= -1;
            }
        }
    }
}
''', 'Mutation2')
```

For $2N$ offspring matrices, the `Mutation2` kernel function flips the sign of two entries with opposite sign in a column. The column and row indices are randomly generated and are highly likely to be different for each offspring matrix. The inputs are given as follows:

- The input `char *Q` is the `Pop` consisting of $4N$ matrices with balanced $+1$ and -1 entries in each column.

- The input `const int row1` is a $2N$ array whose entries are integer values between 0 and $m-1$ and define the first set of row indices.
- The input `const int row2` is a $2N$ array whose entries are integer values between 0 and $m-1$ and define the second set of row indices.
- The input `const int col` is a $2N$ array whose entries are integer values between 1 and $m-1$ and define the set of column indices.
- The input `const int m` is an integer that is represented by the size of the matrices in `Pop`.
- The input `const int N` is an integer such that `Pop` has $4N$ matrices.

In the raw kernel function we define `int k` as the index of matrices, `int i` as the index of rows, and `int j` as the index of columns. The `if ((k>=2*N)&& (k<4*N))` statement sets a boundary for `k` such that only the offspring matrices are mutated. For the actual mutation process, an if statement is given as follows:

```
if((i==0) && (j==col[k-2*N])){
    if (Q[m*m*k+m*row1[k-2*N]+j] != Q[m*m*k+m*row2[k-2*N]+j]){
        Q[m*m*k+m*row1[k-2*N]+j] *= -1;
        Q[m*m*k+m*row2[k-2*N]+j] *= -1;
    }
}
```

The `if((i==0)&& (j==col[k-2*N]))` statement fixes the index of rows `i` so all rows are not worked on and fixes the index of columns `j` to assure we work with the appropriate index of columns for each offspring matrix. The last if statement checks that the sign of the two entries in each offspring matrix is opposite, if so, the signs are flipped. There is no mutation if this condition is not met.

To demonstrate how the `Mutation2` kernel function works lets consider an example. Let $m = 8$ and $N = 2$ such that the we have a population of 8 matrices that have the sizes 8×8 with a

balanced number $+1$ and -1 in each column. This implies there are 4 offspring matrices that will be mutated. Let `rowindx1 = [3 1 2 6]`, `rowindx2 = [0 5 6 0]`, and `colindx = [1 7 5 3]` be the randomly generated array of indices. The mutation of the 4 offspring matrices are given in Figure 4.2 such that

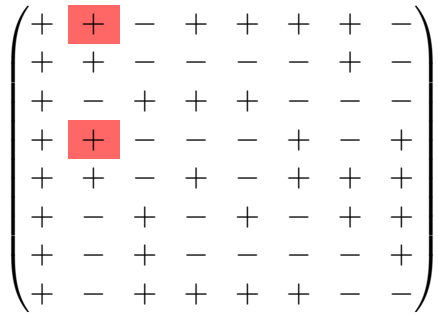
- In part (a), we use the indices `rowindx1 = 3`, `rowindx2 = 0`, and `colindx = 1`.
- In part (b), we use the indices `rowindx1 = 1`, `rowindx2 = 5`, and `colindx = 7`.
- In part (c), we use the indices `rowindx1 = 2`, `rowindx2 = 6`, and `colindx = 5`.
- In part (d), we use the indices `rowindx1 = 6`, `rowindx2 = 0`, and `colindx = 3`.

Additionally, In part (a), the red highlights represent that a mutation did not occur since the signs are the same. In parts (b), (c), and (d), the green highlights represent that a mutation did occur since the signs are opposite. Each offspring matrix has a randomly generates set of row and column indices. This provides variation between each offspring matrix as opposed to the example in Figure 4.1 where all the offspring matrices had the same mutation.

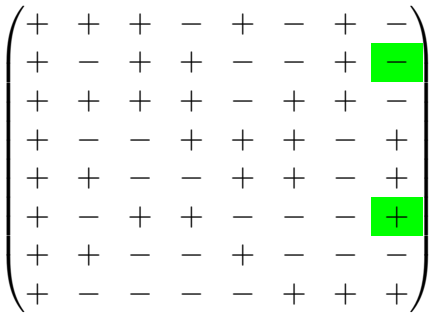
The largest Hadamard matrix found is of the size 12×12, which is the same result as the last method. Although this method achieves variation between each offspring matrix. The same issue stands, that is, when working with larger matrices, the flipping of just two entries may not be good enough as the convergence may be too slow. To improve upon this, a third mutation function was implemented that allows us to choose the number of random columns and random pair of row indices that we want to mutate..

4.6.3 Third Mutation Function

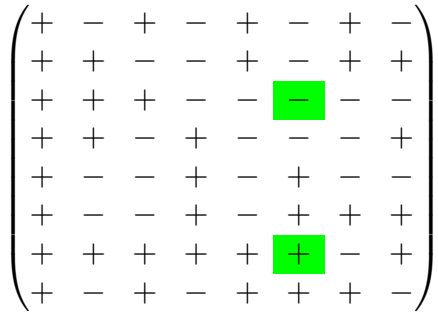
The previous mutation function achieved variation between each offspring matrix. This method builds upon that by controlling the number of random columns and pair of random rows that we want to mutate for each matrix. This allows us to have greater variation amongst the offspring matrices and in theory, will give us a better chance of finding an Hadamard matrix. This method



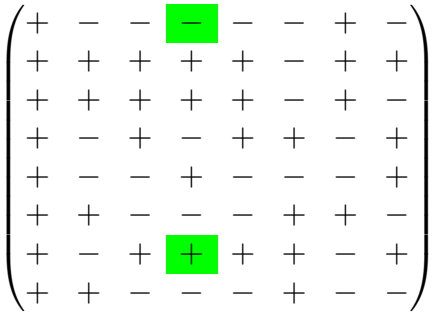
(a) First Offspring Matrix



(b) Second Offspring Matrix



(c) Third Offspring Matrix



(d) Fourth Offspring Matrix

Figure 4.2: Example of the Second Mutation Method

flips a pair of $+1$ and -1 entries by randomly selecting a column matrix of indices (except the first) and randomly selecting two row matrices of indices. This method is coded as follows:

```
rowindx1 = cp.random.random_integers(0,m-1,size=(2*N,NR))
rowindx2 = cp.random.random_integers(0,m-1,size=(2*N,NR))
colindx = cp.random.random_integers(1,m-1,size=(2*N,NC))
Mutation3(grids, blocks, (Pop, rowindx1, rowindx2, colindx, NC, NR, m, N))
```

For `rowindx1` and `rowindx2`, the CuPy function `cp.random.random_integers(0, m-1, size=(2*N,NR))` returns a $2N \times NR$ matrix with entries containing the integer values between 0 and $m-1$. Similarly, the CuPy function `colindx = cp.random.random_integers(1, m-1, size=(2*N,NC))` returns a $2N \times NC$ matrix with entries containing the integer values between 1 and $m-1$. Finally, the `Mutation3` kernel function is called and is given as follows:

```
Mutation3 = cp.RawKernel(r'''
extern "C" __global__
```

```
void Mutation3(char *Q, const int* row1, const int* row2, const int* col,
   const int NC, const int NR, const int m, const int N)
{
    int k = blockDim.x*blockIdx.x+threadIdx.x;
    int i = blockDim.y*blockIdx.y+threadIdx.y;
    int j = blockDim.z*blockIdx.z+threadIdx.z;
    int jc, ir, coljc, rowir1, rowir2;
    if ((k>=2*N) && (k<4*N)){
        if((i==0) && (j==0)){
            for(jc=0; jc<NC; jc++){
                coljc=col[(k-2*N)*NC+jc];
                for(ir=0; ir<NR; ir++){
                    rowir1=row1[(k-2*N)*NR+ir];
                    rowir2=row2[(k-2*N)*NR+ir];
                    if (Q[m*m*k+m*rowir1+coljc] != Q[m*m*k+m*rowir2+coljc]){
                        Q[m*m*k+m*rowir1+coljc] *= -1;
                        Q[m*m*k+m*rowir2+coljc] *= -1;
                    }
                }
            }
        }
    }
}
''', 'Mutation3')
```

For $2N$ offspring matrices, the `Mutation3` kernel function can flip multiple pairs $+1$ and -1 entries in multiple columns. The inputs are given as follows:

- The input `char *Q` is the `Pop` consisting of $4N$ matrices with balanced $+1$ and -1 entries in each column.
- The input `const int row1` is a $2N \times NR$ matrix whose entries are integer values between 0 and $m-1$ and define the first set of row indices.
- The input `const int row2` is a $2N \times NR$ matrix whose entries are integer values between 0 and $m-1$ and define the second set of row indices.
- The input `const int col` is a $2N \times NC$ matrix whose entries are integer values between 1 and $m-1$ and define the set of column indices.
- The input `const int NC` is an integer value between 1 and m that denotes the number of columns we want to mutate for each matrix.
- The input `const int NR` is an integer value between 1 and $m/2$ that denotes the half the number of rows we want to mutate for each matrix.
- The input `const int m` is an integer that is represented by the size of the matrices in `Pop`.
- The input `const int N` is an integer such that `Pop` has $4N$ matrices.

In the raw kernel function we define `int k` as the index of matrices, `int i` as the index of rows, and `int j` as the index of columns. We define some integer values such that `int jc, ir, coljc, rowir1, rowir2`. The `if ((k>=2*N)&& (k<4*N))` statement sets a boundary for `k` such that only the offspring matrices are mutated. The line `if((i==0)&& (j==0))` fixes the row index `i` and column index `j`, this is used for indexing purposes. Now, the actual mutation step is executed as follows:

```
for(jc=0; jc<NC; jc++){
    coljc=col[(k-2*N)*NC+jc];
    for(ir=0; ir<NR; ir++){
        rowir1=row1[(k-2*N)*NR+ir];
```

```
        rowir2=row2[(k-2*N)*NR+ir];
        if (Q[m*m*k+m*rowir1+coljc] != Q[m*m*k+m*rowir2+coljc]){
            Q[m*m*k+m*rowir1+coljc] *= -1;
            Q[m*m*k+m*rowir2+coljc] *= -1;
        }
    }
}
```

The `for(jc=0; jc<NC; jc++)` loop creates iteration steps for `jc` that increases by 1 every loop, it starts at 0 and ends at $NC-1$. In this for loop, the line `coljc=col[(k-2*N)*NC+jc]` defines `coljc` to be the column indices that correspond to the column indices of the offspring matrices that we want mutate. Within this for loop, we have another for loop that is given by `for(ir=0; ir<NR; ir++)`, notice that this line does the same thing as the previous for loop, but for the row indices `rowir1` and `rowir2`. Finally, the `if (Q[m*m*k+m*rowir1+coljc] != Q[m*m*k+m*rowir2+coljc])` statement checks that the sign of the two entries are opposite, if so, the signs are flipped. There is no mutation if this condition is not met. Note that the first for loop works on one column for each offspring matrix at a time. So if we choose to mutate multiple columns, each column will be done one-by-one, but for all offspring matrices at a time. The second for loop works in a similar manner but with row indices.

To demonstrate how the `Mutation3` kernel function works lets consider an example. Let $m = 8$ and $N = 2$ such that the we have a population of 8 matrices that have the sizes 8×8 with a balanced number $+1$ and -1 in each column. This implies there are 4 offspring matrices that will be mutated. Let $NC = 3$ be the number of columns mutated and $NR = 2$ be the number of row pairs

mutated such that we have the randomly generated column and row indices given by

$$\texttt{rowindx1} = \begin{bmatrix} 6 & 5 \\ 4 & 2 \\ 0 & 6 \\ 1 & 6 \end{bmatrix}, \quad \texttt{rowindx2} = \begin{bmatrix} 3 & 1 \\ 7 & 0 \\ 1 & 2 \\ 2 & 5 \end{bmatrix}, \quad \texttt{colindx} = \begin{bmatrix} 6 & 5 & 2 \\ 5 & 2 & 4 \\ 3 & 1 & 6 \\ 2 & 7 & 6 \end{bmatrix} \tag{4.28}$$

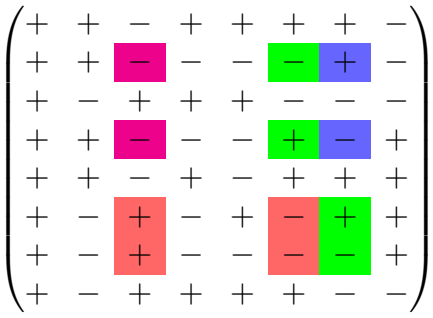
The mutation of the 4 offspring matrices are given in Figure 4.3 such that

- In part (a), the indices are $\texttt{rowindx1} = [6,\ 5]$, $\texttt{rowindx2} = [3,\ 1]$, and $\texttt{colindx} = [6,\ 5,\ 2]$.
- In part (b), the indicies are $\texttt{rowindx1} = [4,\ 2]$, $\texttt{rowindx2} = [7,\ 0]$, and $\texttt{colindx} = [5,\ 2,\ 4]$.
- In part (c), the indicies are $\texttt{rowindx1} = [0,\ 6]$, $\texttt{rowindx2} = [1,\ 2]$, and $\texttt{colindx} = [3,\ 1,\ 6]$.
- In part (d), the indicies are $\texttt{rowindx1} = [1,\ 6]$, $\texttt{rowindx2} = [2,\ 5]$, and $\texttt{colindx} = [2,\ 7,\ 6]$.

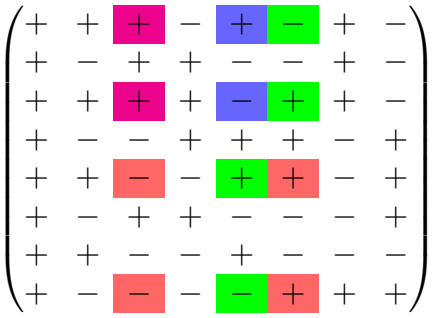
In each column, a successful mutation is represented in green and blue (if there is more than one successful), an unsuccessful mutation is represented in red and magenta (if there is more than one unsuccessful).

In analyzing Figure 4.3, although we do choose the number of columns and pair of rows we want to mutate, it is not guaranteed to mutate all of them. In part (a), we saw the `colindx=2` have no mutation, the `colindx=5` have one mutation, and the `colindx=6` have two mutations. So, the maximum number of mutations for each column is *NR*.

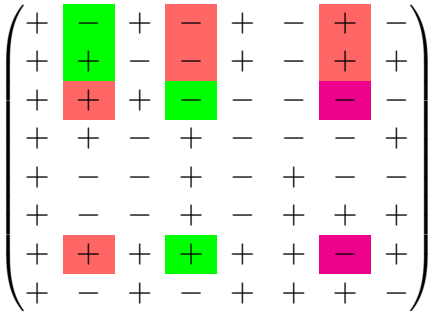
This method improves upon the issue of working with larger matrices, we have achieved greater variation among offspring matrices and the hope is the convergence to be faster when searching for Hadamard matrices. Using this method, the largest Hadamard matrix found so far is of the size 32×32, this is a major improvement from the other two methods.



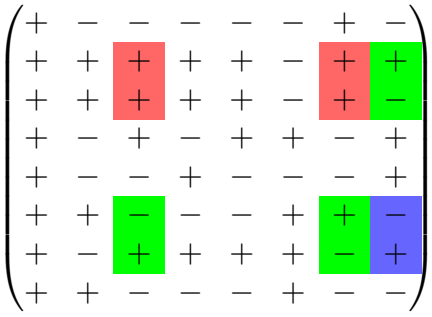
(a) First Offspring Matrix



(b) Second Offspring Matrix



(c) Third Offspring Matrix



(d) Fourth Offspring Matrix

Figure 4.3: Example of the Second Mutation Method

CHAPTER V

COMPUTATIONAL RESULTS

5.1 Local Minimum

Genetic Algorithms also suffer from finding local minimums instead of global minimum similarly to the Simulated Annealing Algorithm. Figure 5.1 shows the minimum of the fit function as a function of the iteration number for matrix size 20×20 with $NC = NR = 4$. It is not just that the minimum of the fit function stays constant (gets stuck) without converging to zero, but the whole parent population becomes homogeneous. One way to try to prevent stalling at a local minimum is to use a selection process with some probability. This approach does not seem to result in improved convergence. We had some success to speed up convergence by selecting low number of columns for the mutations. This is discussed in Section 5.3.

5.2 Fitness Function Comparisons

The functions tested in this section are the minimizing fitness functions discussed in Section 4.3 and are given as follows:

$$F_1 = \sum_{i,j} \left| Q^T Q \right| - m^2 \geq 0. \tag{5.1}$$

$$F_2 = \text{nonzero}\left(Q^T Q\right) - m \geq 0. \tag{5.2}$$

$$F_3 = m^{m/2} - |\det(\text{Population})| \geq 0. \tag{5.3}$$

Table 5.1 shows the average speed (in seconds) for the first three fitness functions while working with 1000 matrices at a time and over 10000 iterations. Through 10 runs, The average speed was taken and was done for each matrix size in the table. The parameter are $N = 250$, $T = 10^4$, $NC = 4$, and $NR = 2$ for several matrix sizes. The purpose of this table was to compare the speed

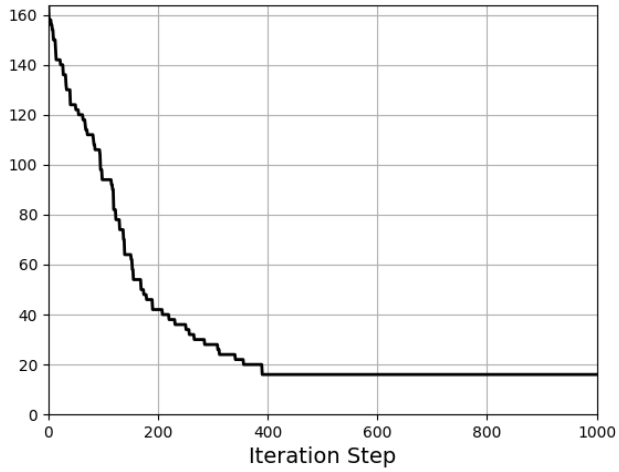


Figure 5.1: Minimum of the Fit function

of each fitness function while working with larger matrices. Note that none of these runs resulted in any Hadamard matrices. For the F_3 function, it was unable to run with 40×40 and larger matrices. This is due to the determinants of the matrices being too large and as a result we see an overflow error in the python code. Over 10000 iterations, comparing F_1 to F_2 we saw that:

- For 20×20, F_2 was 0.16 seconds slower on average than F_1.
- For 40×40, F_2 was 0.26 seconds faster on average than F_1.
- For 100×100, F_2 was 0.75 seconds faster on average than F_1.
- For 200×200, F_2 was 1.86 seconds faster on average than F_1.
- For 400×400, F_2 was 14.26 seconds faster on average than F_1
- For 668×668, F_2 was 58.48 seconds faster on average than F_1.

The results were relatively similar for sizes 200×200 and smaller. For the sizes 400×400 and 668×668, we saw a larger difference in time between the two functions. This indicates that F_2 performs better than F_1 as the size of the matrices increase. Now, what can be said if we use a larger amount of matrices. In this table, we only work with 1000 matrices at a time.

Table 5.2 shows the average speed (in seconds) for the first three fitness functions while working with 40000 matrices at a time and over 10000 iterations. Through 10 runs, The average

Table 5.1: Average Speed in seconds for each Fitness Function using 1000 matrices

Matrix sizes	F_1	F_2	F_3
20×20	17.41	17.57	18.61
40×40	18.70	18.44	–
100×100	26.28	25.53	–
200×200	62.49	60.63	–
400×400	271.80	257.54	–
668×668	873.16	814.68	–

speed was taken and was done for each matrix size in the table. The parameter are $N = 10^4$, $T = 10^4$, $NC = 4$, and $NR = 2$ for several matrix sizes. The purpose of this table was to compare the speed of each fitness function while working with large number of matrices. Note that none of these runs resulted in any Hadamard matrices. Similar to Table 5.1, F_3 is unable to run with 40×40 and larger matrices. Over 10000 iterations, comparing F_1 to F_2 we saw that:

- For 20×20, F_2 was 1.12 seconds faster on average than F_1.
- For 40×40, F_2 was 3.31 seconds faster on average than F_1.
- For 60×60, F_2 was 6.08 seconds faster on average than F_1.
- For 80×80, F_2 was 19.33 seconds faster on average than F_1.
- For 100×100, F_2 was 35.3 seconds faster on average than F_1.

So from the Tables 5.1 and 5.2, we can clearly see that F_2 is the superior fitness function. These results were showed for only 10^4 iterations. Typically when finding larger Hadamard matrices, 10^7 iterations and greater is usually needed to find them. Consequently, for a larger number of iterations, F_2 will perform significantly better. For these reasons, when computing results in Section 5.4, we use the fitness function F_2 given by

$$F_2 = \text{nonzero}\left(Q^T Q\right) - m \geq 0. \tag{5.4}$$

Table 5.2: Average Speed in seconds for each Fitness Function using 40000 matrices

Matrix sizes	F_1	F_2	F_3
20×20	60.50	59.38	64.79
40×40	103.15	99.84	–
60×60	176.28	170.20	–
80×80	275.92	256.59	–
100×100	416.68	381.38	–

Table 5.3: Average Iteration steps for a 12×12 matrix using Mutation3 kernel function

Average Iterations		NC			
		1	2	3	4
NR	1	34.2	35.8	37.4	35.8
	2	32.3	38.8	40.5	43.4
	3	35.2	42.6	47.5	50.7
	4	36.3	46.0	49.9	65.7

5.3 Mutation Comparisons

Table 5.3 shows the average number of iterations required to find a 12×12 Hadamard matrix given different *NC* and *NR* values. For each case 10 runs were made. The numbers in the table suggests that mutating two pairs ($NR = 2$) in one column ($NC = 1$) takes the least amount of steps to find an Hadamard matrix. In addition, the best results occurred when $NC = 1$ with any amount of row pairs.

Table 5.4 shows the average number of iterations required to find a 16×16 Hadamard matrix given different *NC* and *NR* values. For each case 10 runs were made. We found that the smallest numbers for $NR = 1$ and $NC = 2, 3, 4$ were obtained with the majority runs not finding Hadamard matrix. This suggest that either an Hadamard matrix was found fast or it wasn't found at all. The best results occurred when $NC = 1$ with any amount of row pairs. Although it took slightly more iterations to find the Hadamard matrices, it finding them more consistently. For the case of larger matrices, this table summarizes those results. Therefore when finding Hadamard matrices, we use a low number of columns $(NC = 1)$ or $(NC = 2)$ and a larger number of row pairs depending on the size of the matrix.

Table 5.4: Average Iteration steps for a 16×16 matrix using Mutation3 kernel function

Average Iterations		NC			
		1	2	3	4
NR	1	130.1	119.4*	116.0*	115.5*
	2	121.0	181.3	135.0	168.7
	3	120.0	158.1	165.9	202.7
	4	120.2	166.2	236.2	301.6

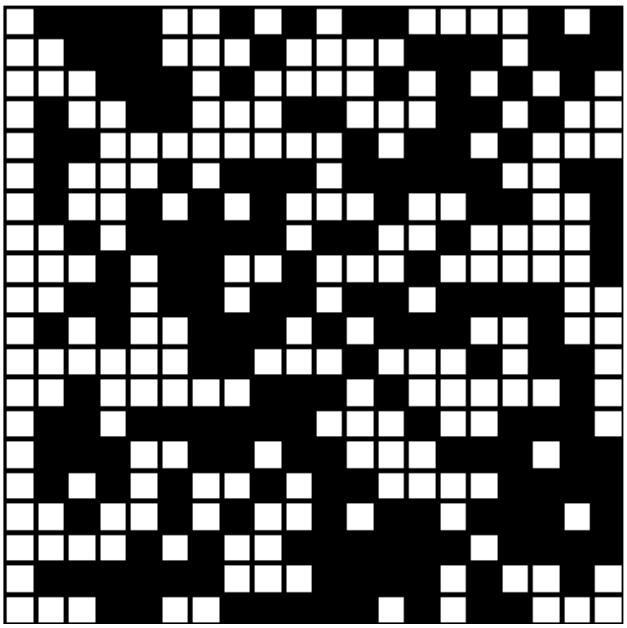


Figure 5.2: 20×20 Hadamard matrix

5.4 Results

The following are Hadamard matrices found using the fitness function F_2 and the mutation kernel function `Mutation3`.

- In Figure 5.2, we show a 20×20 Hadamard matrix using the parameters $k = 5$, $N = 10^3$, $T = 10^7$, $NC = 2$, and $NR = 8$. We acquired this result after 517 seconds and 258875 iterations.

- In Figure 5.3, we show a 24×24 Hadamard matrix using the parameters $k = 6$, $N = 10^3$, $T = 10^7$, $NC = 2$, and $NR = 10$. We acquired this result after 19997 seconds and 9576814 iterations.

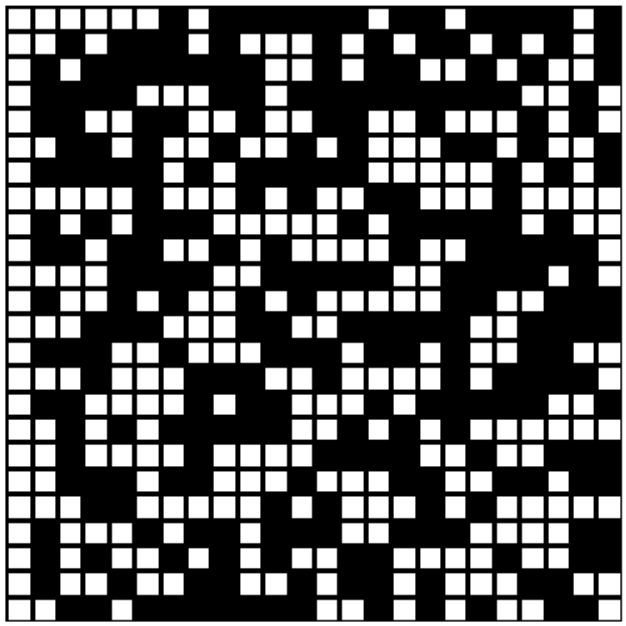


Figure 5.3: 24×24 Hadamard matrix

- In Figure 5.4, we show a 28×28 Hadamard matrix using the parameters $k = 7$, $N = 10^4$, $T = 10^8$, $NC = 1$, and $NR = 10$. We acquired this result after 3054 seconds and 428082 iterations.

- In Figure 5.5, we show a 32×32 Hadamard matrix using the parameters $k = 8$, $N = 2(10^4)$, $T = 10^7$, $NC = 1$, and $NR = 12$. We acquired this result after 94923 seconds and 7472853 iterations.

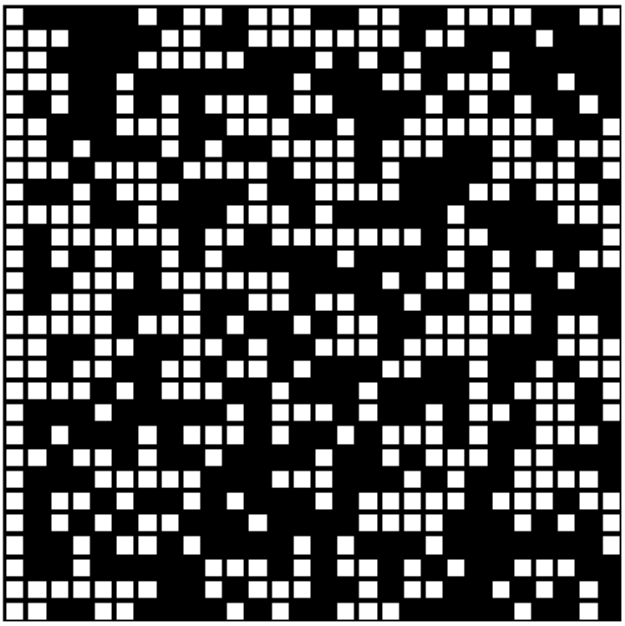


Figure 5.4: 28×28 Hadamard matrix

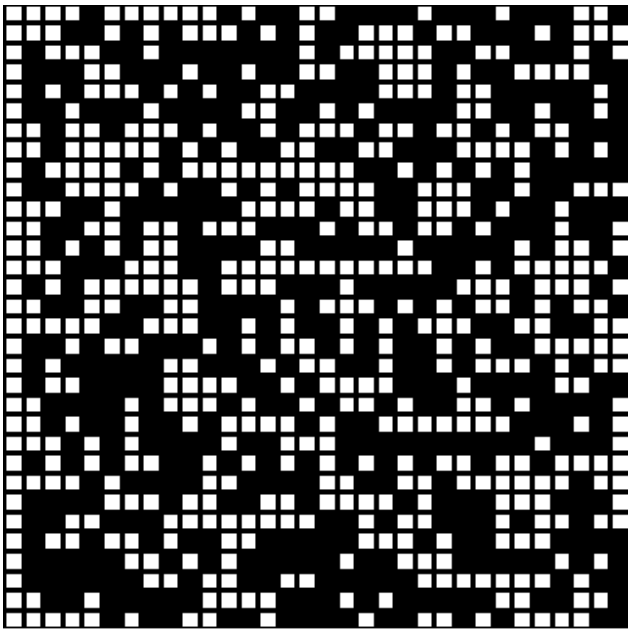


Figure 5.5: 32×32 Hadamard matrix

REFERENCES

Bridges, C.L. and D.E. Goldberg (1987). "An Analysis of Reproduction and Crossover in a Binary-Coded Genetic Algorithm." In: *ICGA*. Ed. by John J. Grefenstette. Lawrence Erlbaum Associates, pp. 9–13. ISBN: 0-8058-0158-8.

Browne, P. et al. (2021). "A Survey of the Hadamard Maximal Determinant Problem". In: *The Electronic Journal of Combinatorics* 28.4. DOI: 10.37236/10367.

Craigen, R. (1995). "Signed groups, sequences, and the asymptotic existence of Hadamard matrices". In: *J. Combin. Theory Ser. A* 71.2, pp. 241–254. ISSN: 0097-3165. DOI: 10.1016/0097-3165(95)90002-0.

Craigen, R., J. Seberry, and X.M. Zhang (1992). "Product of four Hadamard matrices". In: *J. Combin. Theory Ser. A* 59.2, pp. 318–320. ISSN: 0097-3165. DOI: 10.1016/0097-3165(92)90073-4.

Durstenfeld, R. (July 1964). "Algorithm 235: Random Permutation". In: *Commun. ACM* 7.7, p. 420. ISSN: 0001-0782. DOI: 10.1145/364520.364540. URL: https://doi.org/10.1145/364520.364540.

Fisher, R.A. and F. Yates (1938). *Statistical Tables for Biological, Agricultural and Medical Research*. Edinburgh, UK; London, UK: Oliver and Boyd, pp. viii + 90 + 1.

Goethals, J.M. and J.J. Seidel (1970). "A skew Hadamard matrix of order 36". In: *J. Austral. Math. Soc.* 11, pp. 343–344. ISSN: 0263-6115.

Hadamard, J. (1893). “Résolution d’une question relative aux déterminants”. In: *Bulletin des Sciences Mathématiques* 17, pp. 240–246.

Han, A. and K. Rochford (2020). *EVGA GeForce RTX 3080 XC3 BLACK GAMING, 10G-P5-3881-KL, 10GB GDDR6X, iCX3 Cooling, ARGB LED, LHR*. https://www.evga.com/products/product.aspx?pn=10G-P5-3881-KL.

Harris, C.R. et al. (Sept. 2020). “Array programming with NumPy”. In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press.

Jayathilake, A.A.C.A., A.A.I. Perera, and M.A.P. Chamikara (Apr. 2014). “A New Set of 32 In-equivalent Hadamard Matrices of Order 404 of Goethals- Seidel Type”. In: *Elixir* 69, pp. 23266–23272.

Jayathilake, C., A.A.I. Perera, and M.A.P. Chamikara (Jan. 2013). “Discrete Walsh-Hadamard Transform in Signal Processing”. In: *International Journal of Research in Information Technology* 1, pp. 80–89.

Kharaghani, H. and B. Tayfeh-Rezaie (2005). “A Hadamard matrix of order 428”. In: *J. Combin. Des.* 13.6, pp. 435–440. ISSN: 1063-8539. DOI: 10.1002/jcd.20043. URL: https://doi.org/10.1002/jcd.20043.

Mohammadian, A. and B. Tayfeh-Rezaie (2019). “Hadamard matrices with few distinct types”. In: *Linear and Multilinear Algebra* 67.8, pp. 1596–1605. DOI: 10.1080/03081087.2018.1464113. eprint: https://doi.org/10.1080/03081087.2018.1464113. URL: https://doi.org/10.1080/03081087.2018.1464113.

NVIDIA, P. Vingelmann, and F.H.P. Fitzek (2020). *CUDA, release: 10.2.89*. URL: https://developer.nvidia.com/cuda-toolkit.

Okuta, R. et al. (2017). "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

Paley, R. (1933). "On orthogonal matrices". In: *J. Math. Phys.* 12, pp. 311–320.

Plotkin, M. (1972). "Decomposition of Hadamard matrices". In: *J. Combinatorial Theory Ser. A* 13, pp. 127–130. ISSN: 0097-3165. DOI: 10.1016/0097-3165(72)90015-5. URL: https://doi.org/10.1016/0097-3165(72)90015-5.

Seberry, J. (2017). *Orthogonal designs*. Hadamard matrices, quadratic forms and algebras, Revised and updated edition of the 1979 original [MR0534614]. Springer, Cham, pp. xxiii+453. ISBN: 978-3-319-59031-8. DOI: 10.1007/978-3-319-59032-5. URL: https://doi.org/10.1007/978-3-319-59032-5.

Seberry, J. and M. Yamada (1992). "Hadamard matrices, sequences, and block designs". In: *Contemporary design theory*. Wiley-Intersci. Ser. Discrete Math. Optim. Wiley, New York, pp. 431–560.

Stanton, R.G. and D.A. Sprott (1958). "A family of difference sets". In: *Canadian J. Math.* 10, pp. 73–77. ISSN: 0008-414X. DOI: 10.4153/CJM-1958-008-5. URL: https://doi.org/10.4153/CJM-1958-008-5.

Suksmono, A. (2016). "Finding a Hadamard Matrix by Simulated Annealing of Spin-Vectors". In: *Journal of Physics: Conference Series* 18(3), pp. 66–70.

Suksmono, A.B. and Y. Minato (Oct. 2019). “Finding Hadamard Matrices by a Quantum Annealing Machine”. In: *Scientific Reports* 9.14380. DOI: 10.1038/s41598-019-50473-w. URL: https://doi.org/10.1038/s41598-019-50473-w.

Sylvester, J. (1867). “Thoughts on inverse orthogonal matrices, simultaneous sign successions, and tessellated pavements in two or more colours, with applications to Newton’s rule, ornamental tile-work, and the theory of numbers”. In: *Philosophical Magazine* 34, pp. 461–475.

Walsh, J.L. (1923). “A Closed Set of Normal Orthogonal Functions”. In: *American Journal of Mathematics* 45.1, pp. 5–24. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2387224.

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems*. Packt Publishing. ISBN: 9781838559182. URL: https://books.google.com/books?id=A0vODwAAQBAJ.

Xia, M.J. (1992). “Some infinite classes of special Williamson matrices and difference sets”. In: *J. Combin. Theory Ser. A* 61.2, pp. 230–242. ISSN: 0097-3165. DOI: 10.1016/0097-3165(92)90020-U.

BIOGRAPHICAL SKETCH

The author, Raven I. Ruiz was born November 28, 1997, in Mcallen, Texas. He has two siblings, Rigoberto Ruiz Jr. and Andrew A. Ruiz with parents Rigoberto Ruiz Sr. and Araceli Ruiz. To contact him, his email address is ruiziraven@gmail.com.

From 2012-2016, Raven was enrolled at Robert Vela High School in Edinburg, Texas. After graduating in 2016, he enrolled at the University of Texas Rio Grande Valley in Edinburg, Texas. He was enrolled as an undergraduate student from 2016-2020. In 2020, he became a magna cum lauda graduate and received a Bachelor of Science in Applied Mathematics.

In 2020, he continued his education at the University of Texas Rio Grande Valley as a graduate student and was selected to receive the College of Sciences Dean's Graduate Assistantship Award. He was enrolled as a graduate student from 2020-2022. In 2022, he received a Master of Science in Applied Mathematics.