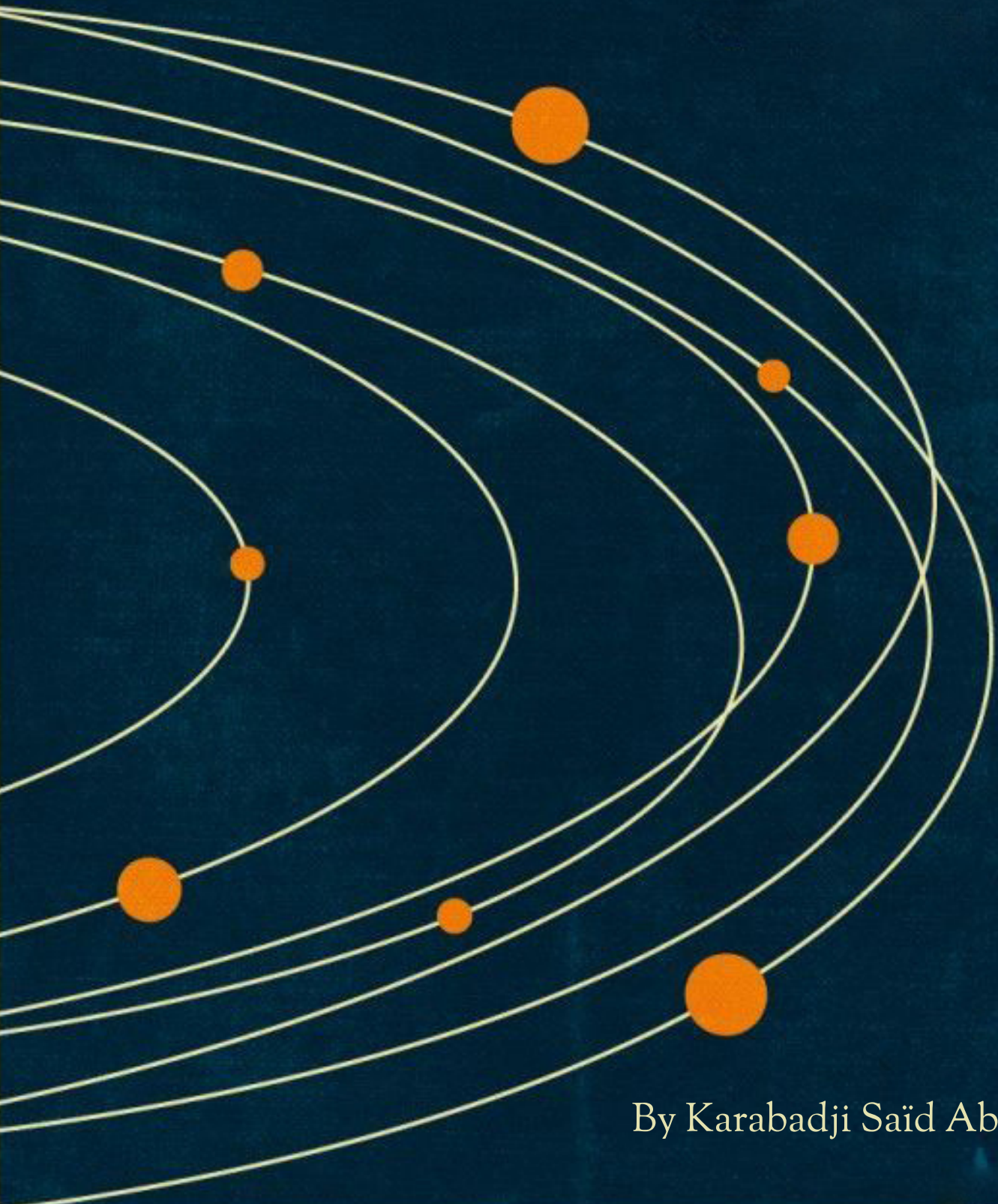


National Higher School Of Mathematics

Cosmic Encryption

About the protection of cosmic snapshots and Data
recovery



By Karabadji Saïd Abd-Allah

NATIONAL HIGHER SCHOOL OF MATHEMATICS

Algebra And Coding

Cosmic Encryption

About the protection of cosmic snapshots and data recovery

Karabadji Saïd Abd-Allah

Abstract

No transmission channel is perfect, that is, no matter the information we send in any of them, there will always be a certain proportion of that information that will be corrupted at best and lost at worst. To solve this problem, engineers developed the theory of error correcting codes that treats of the developments of languages with certain properties that preserve structure of the information with different degrees of protection.

In this paper we treat of a particular Code called Hadamard Codes that were used in the Mariner programs by NASA in the 1960's to recover the spatial snapshots taken by the Mariner 2 satellite beginning with an explanation of the code itself right before diving into a programing application and testing.

Contents

1	Introduction	3
2	What Are Hadamard Codes	4
2.1	Hadamard Matrices:	4
2.2	Distances On A Hadamard Matrix	4
2.3	Encoding and Decoding Processes Using Hadamard Matrices	5
3	Python Code Review	7
3.1	Imports	8
3.2	Useful Functions	8
3.3	Hadamard Code/Decode Generator	10
3.4	Image Encoder	11
3.5	Error Introducing Functions	13
3.5.1	Correctable Error	13
3.5.2	Detectable Error	14
3.5.3	Non Detectable Error	15
3.6	Channel's Noise	16
3.7	Decoder	17
4	Testing On Mariner Missions spacial Snapshots	18

5	Testing on Colored Images From Mariner 10	20
6	Managerial Report	22

1 Introduction

In 1962, the American National Aeronautics and Space Administration (NASA) launched its second satellite mission on the Mariner program. After a devastating failure that was the Mariner 1 mission and after many improvements regarding the launching as well as control procedures; the Mariner 2 mission was a total success that sent back pictures of Venus' surface in addition to many information about the Evening's Star atmospheric composition, CO₂ levels, pressure levels etc...

In this paper, we treat of the very important problem that NASA faced when dealing with the transmission of those spatial snapshots and information as well as the ingenious way that were developed in order to recover them.

Due to the solar winds in space, all messages, files, pictures that we may send or receive are object to perturbation, thus stripping us of all reliable Data we may need in order to study extraterrestrial entities.

However thanks to the theory of error correcting codes, we can equip our spatial machinery with specific languages that encode the information right before sending it which can resist the perturbation of outer space. In the Mariner program, NASA opted for the use of Hadamard Codes which are codes based solely on the use of Hadamard Matrices.

We note that for all the different languages we may develop, we are always limited by the compromise of choosing between the time it takes us to encode and decode our information as well as the resources we may need to do that VS the reliability and security of our data.

In other words, we have to choose between a language that is reliable (recovers a big proportion of our information) but the time and resources it takes to perform an encoding/decoding procedure may grow fast with respect to the quantity of information we want to process. Or, in the counter part, a language that takes lesser time and consumes lesser resources however the information we send with it would be in more danger of corruption.

To solve that Dilemma, NASA went with the first strategy that is to use a language that is reliable no matter the time and resources it will take them to recover the data since the cost of the mission itself was far more important then the cost of recovery.

2 What Are Hadamard Codes

2.1 Hadamard Matrices:

A Hadamard Matrix H of order n is an $n \times n$ matrix of 1 and -1 such that

$$\forall n \in \mathbb{N} : HH^t = nI_n$$

Proposition 1. *If H is a Hadamard matrix of order n then the matrix:*

$$\tilde{H} = \begin{bmatrix} H & H \\ H & -H \end{bmatrix}$$

Is a Hadamard Matrix of order $2n$

We can define Hadamard matrices of orders that are powers of 2 in a recursive way like such:

$$\begin{cases} H_0 = 1 \\ H_{2^{n+1}} = \begin{bmatrix} H_{2^n} & H_{2^n} \\ H_{2^n} & -H_{2^n} \end{bmatrix} \text{ for all } n \geq 1 \end{cases}$$

In this manner we have:

$$H_0 = 1$$

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

2.2 Distances On A Hadamard Matrix

Let $n \in \mathbb{N}$ and let H be the hadamard matrix of order 2^n constructed in the manner stated before.

We recall that the Hamming distance " d_H " between two codewords represents the number of different entries in each codeword. That is:

for $\mathbf{x} = x_1 x_2 \dots x_n$, $\mathbf{y} = y_1 y_2 \dots y_n$

$$d_H(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n \tilde{\delta}_{x_i, y_i} \text{ where } \tilde{\delta}_{x_i, y_i} = 1 \iff x_i \neq y_i$$

We will later on use this distance to give a justification for the reason this Code was originally chosen for the Mariner mission.
Next, let us consider the $2^{n+1} \times 2^n$ matrix:

$$\tilde{H} = \begin{bmatrix} H \\ -H \end{bmatrix}$$

we have the following

Proposition 2. *for $1 \leq i, j \leq 2^{n+1}$ let R_i, R_j respectively denote the i -th and the j -row of the matrix \tilde{H} then we have:*

$$1) d_H(R_i, R_j) = 2^n \quad \text{if } j = i + 2^n$$

$$2) d_H(R_i, R_j) = 2^{n-1} \quad \text{otherwise}$$

This tells us that the rows agree in no position if we are treat the two matrices seperately and pick the same row in each one, otherwise the rows always agree in $\frac{2^n}{2} = 2^{n-1}$ positions

2.3 Encoding and Decoding Processes Using Hadamard Matrices

1) Encoding:

In order to Encode an information of length n using the Hadamard coding method, the following steps must be followed:

1) Generate the Hadamard matrix of order 2^k such that $n \leq 2^{k+1}$ and to do so we must solve for k that is it suffices to take

$$k \geq \frac{\ln(\frac{n}{2})}{\ln(2)} \text{ that is } k \geq \log_2(\frac{n}{2})$$

2) Assign to each letter/bit of our information a unique row of the matrix \tilde{H} which will be considered as the codeword encoding that unique information which will then be sent through the channel.

example:

Let us say we want to encode the word "Venus" then we first see that it contains five letters. Accordingly, we must use the matrix H_4 in which case we would have:

$$\widetilde{H}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix}$$

after that we inject the letters of our word into the set of rows of the above matrix giving us:

$$\begin{aligned} \text{"V"} &\rightarrow [1, 1, 1, 1] \\ \text{"e"} &\rightarrow [1, -1, 1, -1] \\ \text{"n"} &\rightarrow [1, 1, -1, -1] \\ \text{"u"} &\rightarrow [1, -1, -1, 1] \\ \text{"s"} &\rightarrow [-1, -1, -1, -1] \end{aligned}$$

We then send in the channel the 4 bit codewords:

$$(1 \ 1 \ 1 \ 1) \ (1 \ -1 \ 1 \ -1) \ (1 \ 1 \ -1 \ -1) \ (1 \ -1 \ -1 \ 1) \ (-1 \ -1 \ -1 \ -1)$$

• **Remark :** Depending on the need, we may choose larger dimensions of the Hadamard matrix even when the latter suffices to encode the information. We will later on show that this is intrinsically motivated by the amount of protection we want form our code.

2) Decoding:

Supposing we received signals in a channel that first originated from a Hadamard coded information. The next step would be to Decode that information to reveal the message it held (in the best case scenarios) or potentially reveal some errors that could have happened during transmission, part of which will be recoverable and the other possibly lost.

The decoding process in Hadamard codes is based upon the concept of least Hamming distance, that is, for each received "packet" of information (in the exemple above the length of a packet were 4 bits of 1 and -1) we must look for the row of the Hadamard matrix (that was used to encode the message in the first place) that is strictly closer to the received information then all other rows.

We must note that by row we mean the rows of the extended Hadamard matrix \widetilde{H} seen as a vector (just as it was originally used to encode the information) and by strictly closer we mean that the Hamming distance between the received vector and the row is the minimum between all the other distances.

As seen above, the minimal distance of the Hadamard code is $d_c = 2^{n-1}$ where the matrix were originally of dimension $2^{n+1} \times 2^n$ thus we have the following

Proposition 3. *let $\mathbf{C} = \{c_1, c_2, \dots, c_n\}$ be the set of all received information and $\mathbf{R} = \{r_1, r_2, \dots, r_{2^{n+1}}\}$ be the set of all the rows of the extended Hadamard matrix then:*

1) *for $c_j \in \mathbf{C}$ if there exist $1 \leq i \leq 2^{n+1}$ such that : $d_H(r_i, c_j) \leq 2^{n-2} - 1$ then we decode c_j as r_i*

2) *if $\min_{r_i \in \mathbf{R}} d_H(c_j, r_i) \leq 2^{n-1} - 1$ then we know that at least 2 of the rows of the matrix are tied to being the closest to the vector hence we can only detect errors within this received vector but we cannot decode it properly*

Proof. since the minimal distance of Hadamard code is $d = 2^{n-1}$ then a know result in coding theory tells us that we can detect for up to $d - 1$ errors and we can correct for up to $\lfloor \frac{d-1}{2} \rfloor$.

In particular for Hadamard codes we can correct for up to $\lfloor \frac{2^{n-1}-1}{2} \rfloor = \lfloor 2^{n-2} - \frac{1}{2} \rfloor = 2^{n-2} - 1$ errors. And we can detect for up to $2^{n-1} - 1$ errors, hence the bounds proposed in the proposition. \square

• **Remark :**

Since the Hadamard matrices used have their dimensions as powers of 2 then the growth of the length of Hadamard codes is exponential which reveals itself useful when dealing with noisy channels since those codes permit us to correct and detect a large number of errors hence motivating their choice in the Mariner mission.

for exemple: with a Hadamard code of length 256 bits we can detect for up to 127 errors and correct for up to 63 errors.

3 Python Code Review

In this section we will take a deep dive and analyse a python program that permits us to encode images using Hadamard matrices right before introducing some controllable noise which will be considered as the channel's perturbation then try to decode those images back and see the potential differences that occur on the image.

3.1 Imports



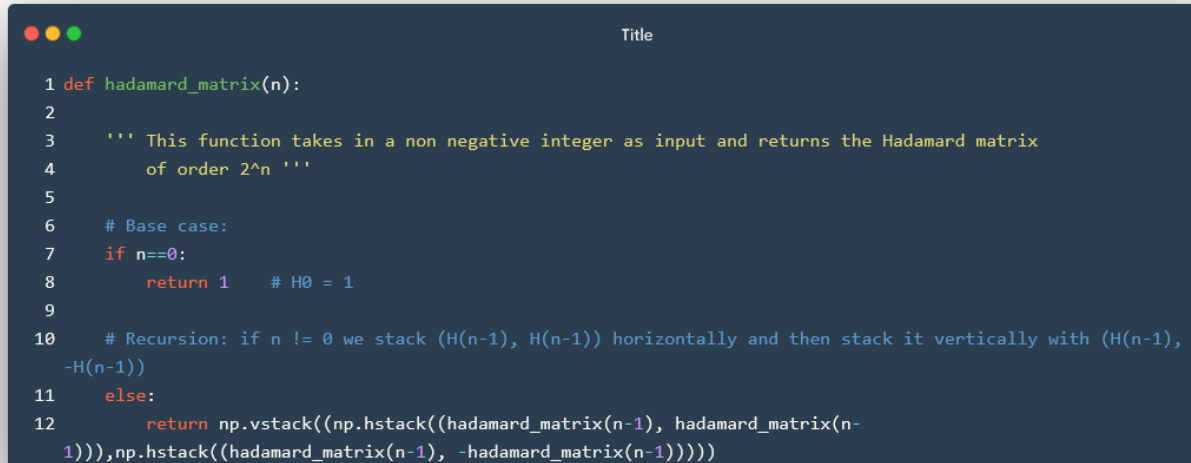
```
1 import numpy as np          # for array manipulation
2 from matplotlib import pyplot as plt    # to plot images
3 from PIL import Image as im  # to manipulate images
4 from random import randint as r        # to generate random numbers
5 from random import sample as s         # to generate random samples
6 from random import shuffle as sh       # to shuffle lists
7 from random import choice as c         # to choose randomly from a list
8 from itertools import product as p     # to construct cartesian products
```

Figure 1: Necessary Imports

We will use the numpy library to manipulate images as arrays and do all computations with it before redisplaying it using matplotlib.

We imported some functions from the random library in order to make the noise function which will be explained later on which will introduce errors randomly in the picture.

3.2 Useful Functions



```
1 def hadamard_matrix(n):
2
3     ''' This function takes in a non negative integer as input and returns the Hadamard matrix
4         of order 2^n '''
5
6     # Base case:
7     if n==0:
8         return 1    # H0 = 1
9
10    # Recursion: if n != 0 we stack (H(n-1), H(n-1)) horizontally and then stack it vertically with (H(n-1),
    -H(n-1))
11    else:
12        return np.vstack((np.hstack((hadamard_matrix(n-1), hadamard_matrix(n-
13        1))),np.hstack((hadamard_matrix(n-1), -hadamard_matrix(n-1))))))
```

Figure 2: Hadamard matrix generator

This function generates Hadamard matrices recursively starting from $H_0 = 1$ then stacks matrices vertically and horizontally in the same manner it was explained in section 2.1

```

1 def construct_sublists(input_list, proportions):
2     ''' This Function takes in a list of elements and a list of proportions then cuts the elements of the list
    respectively, returning a list of lists containing the elements proportionally '''
3
4     total_elements = len(input_list)
5
6     # Calculate the number of elements for each sublist based on proportions
7     sublist_sizes = [int(prop * total_elements) for prop in proportions]
8
9     # Ensure that the sum of sizes is equal to the length of the list
10    sublist_sizes[-1] += total_elements - sum(sublist_sizes)
11
12    # Create a list of unique indices
13    indices = list(range(total_elements))
14
15    # Shuffle the indices
16    sh(indices)
17    sh(input_list)
18
19    # Use the sizes and shuffled indices to construct sublists without repetitions
20    sublists = [input_list[i:i + size] for i, size in zip(range(0, total_elements, sublist_sizes[0]),
    sublist_sizes)]
21
22    return sublists

```

Figure 3: Proportional list generator

This function takes a list of elements, shuffles it and another list of specified proportions that will be used to partition the first into multiple sublists each containing their respective percentage of elements.

This function will be used later on in the noise function in order to introduce mixed types of errors in the picture (correctable errors, detectable errors, lost errors). the following figure illustrates a test of the function.

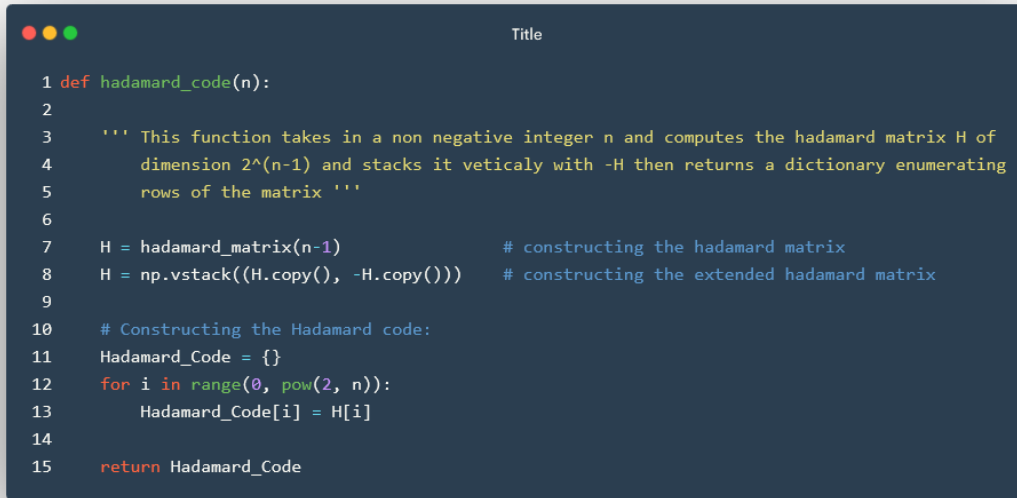
```

1 test_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 proportions = [.2, .5, .3]
3
4 proportional_lists = construct_sublists(test_list, proportions)
5 print(proportional_lists)
6
7 # answer: [[8, 9], [7, 6, 10, 3, 2], [10, 3, 2]]

```

Figure 4: Illustration of the function's work

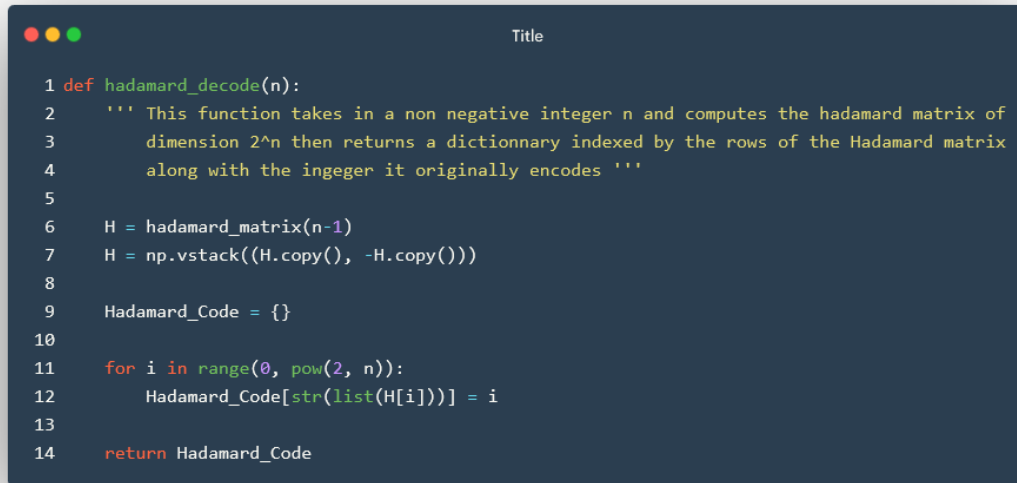
3.3 Hadamard Code/Decode Generator

A screenshot of a code editor window titled "Title" with a dark blue background. It contains Python code for a function named `hadamard_code`. The code defines the function, includes a docstring, constructs a Hadamard matrix `H` and an extended matrix, and then builds a dictionary `Hadamard_Code` mapping integers from 0 to $2^n - 1$ to the rows of the extended matrix.

```
1 def hadamard_code(n):
2
3     ''' This function takes in a non negative integer n and computes the hadamard matrix H of
4         dimension 2^(n-1) and stacks it vertically with -H then returns a dictionary enumerating
5         rows of the matrix '''
6
7     H = hadamard_matrix(n-1)          # constructing the hadamard matrix
8     H = np.vstack((H.copy(), -H.copy())) # constructing the extended hadamard matrix
9
10    # Constructing the Hadamard code:
11    Hadamard_Code = {}
12    for i in range(0, pow(2, n)):
13        Hadamard_Code[i] = H[i]
14
15    return Hadamard_Code
```

Figure 5: Hadamard Code Generator

This function encodes all the non negative integers from 0 to $2^n - 1$ with the rows of the extended Hadamard matrix \tilde{H} of dimension $2^n \times 2^{n-1}$ and returns it in a dictionary as such $\{0: (1, 1, \dots, 1), \dots, 2^n - 1: (-1, 1, \dots, (-1)^{n+1})\}$

A screenshot of a code editor window titled "Title" with a dark blue background. It contains Python code for a function named `hadamard_decode`. The code defines the function, includes a docstring, constructs the same extended Hadamard matrix as in Figure 5, and then builds a dictionary `Hadamard_Code` mapping the string representation of the rows of the matrix back to the integers 0 to $2^n - 1$.

```
1 def hadamard_decode(n):
2
3     ''' This function takes in a non negative integer n and computes the hadamard matrix of
4         dimension 2^n then returns a dictionary indexed by the rows of the Hadamard matrix
5         along with the integer it originally encodes '''
6
7     H = hadamard_matrix(n-1)
8     H = np.vstack((H.copy(), -H.copy()))
9
10    Hadamard_Code = {}
11
12    for i in range(0, pow(2, n)):
13        Hadamard_Code[str(list(H[i]))] = i
14
15    return Hadamard_Code
```

Figure 6: Hadamard Decoding Dictionary Generator

This function is similar to the one before it however it returns the dictionary by inverting the indexes and the values that is because when decoding we are going to look for the color that were assigned to the vector thus we will compare the received vector with the rows of the matrix.

3.4 Image Encoder

```
Title

1 def image_encoder(image_path):
2
3     ''' This function takes a string as input (image_path) which represents the path to the
4     image we wish to encode using Hadamard coding technique and returns the Hadamard encoded
5     picture's array '''
6
7     Hadamard_code = hadamard_code(8) # Generating the Hadamard code of length 128 to encode all the gray scale
8
9     image = im.open(image_path) # Opening the image
10
11     if image.mode == 'L': # This case treats the gray scaled images
12
13         image_array = np.array(image) # The array of the image
14         coded_image_array = np.array([Hadamard_code[image_array[i, j]] for i in range(0, image_array.shape[0])
15         for j in range(0, image_array.shape[1])]) # Creating the array of the encoded image via a list of the image'
16         entries as a hadamard coded vector
17         coded_image_array = coded_image_array.reshape(image_array.shape[0], image_array.shape[1] * 128) #
18         Reshaping the array to the original size of the image
19
20     else:
21
22         R_array = np.array(image)[: , : , 0]
23         G_array = np.array(image)[: , : , 1]
24         B_array = np.array(image)[: , : , 2]
25
26         arrays_list = [R_array, G_array, B_array]
27         coded_arrays_list = []
28
29         for array in arrays_list:
30
31             coded_image_array = np.array([Hadamard_code[array[i, j]] for i in range(0, array.shape[0]) for j
32             in range(0, array.shape[1])]) # Creating the array of the encoded image via a list of the image' entries as a
33             hadamard coded vector
34             coded_image_array = coded_image_array.reshape(array.shape[0], array.shape[1] * 128)
35             coded_arrays_list.append(array)
36
37     return coded_arrays_list
```

Figure 7: Image Encoding Function

This function starts by taking a path to an image before proceeding to its encoding. it first generates a Hadamard Code of length 128bits that will be used to encode all the non negative integers ranging from 0 to 255. That is, for each integer, there is a unique row of the Hadamard matrix that encodes it then it will be stored in the dictionary. We note that this range represents both the gray and the colors scales but for RGB pictures, the case will be treated by itself.

Next if the picture is gray scaled then we start by generating a list of all the coded image entries that is for each entry of the original image (which is a non negative integer

between 0 and 255) we look for the vector that encodes that number and append it to the list. Doing so for all the image's entries, we will end up with a list of vectors (that is a list of lists) such that each vector is a row of the Hadamard matrix, each one of it encoding a unique number.

After that, the list is given to the `np.array()` method which transforms it to a `numpy` array. We will have to reshape the array to "look like" the original picture. Since in the encoded array, each entry of the original image has been replaced by a vector of length 128, we reshape it such that it has the same number of lines as the original image and the number of its columns is 128 times the number of columns of the original image that is given by the method `.reshape((image array.shape[0], image array.shape[1]*128))`.

In The second case where the image is RGB, then upon transforming the image to a `numpy` array, we find that the latter contains 3 layers, that are the RED, GREEN, BLUE layers.

For computational and logical reasons, we adopted the strategy to generate only one code that we will use for all three layers, that is, each layer of color will be treated separately in the same way as the gray scaled images then upon sending the image as a whole, it will be sent in 3 separate messages.

The reason behind this choice is motivated by the time and resource complexity of the algorithm if we decide to use one Hadamard code for all 3 layers, that is because:

for each entry we have 256 possibilities of choosing a color off the RED layer and for each such choice we have 256 possibilities to choose a color off the BLUE layer for which itself has 256 possibilities to choose off the GREEN layer ending up with:

$$256^3 = 16,777,216 \text{ distinct combinations}$$

for which generating the hadamard code itself demands over 553.24 GIB to be stored. On the other hand, while decoding the image, we will later see that for each received vector we must check its Hamming distance with all the rows of the matrix in order to find the closest codeword. This task will reveal itself particularly heavy if we decide to encode all layers at once since for each entry of the received coded image, we need to check it with all 253^3 triplets of numbers which as you may have guessed is absolutely cumbersome.

3.5 Error Introducing Functions

3.5.1 Correctable Error

```
1 def correction_type_error(coded_array, list_of_coordinates):
2
3     ''' This function takes in 3 arguments and returns an array with some Correctable noise
4     introduced in the respective places listed in the (list_of_coordinates) variable
5
6     1- coded_array: type = np_array. represents the array with its entries being the
7     hadamard coded colors of the original image ie vectors of the hadamard matrix each
8     representing a particular color
9
10    2- list_of_coordinates: type = list. represents a list of randomly generated tuples with
11    entries ranging in the between the dimension of the image that is they represent coordinates
12    of individual pixels in the image that will be perturbed in the transmission '''
13
14    noised_coded_array = coded_array.copy() # Creating a copy of the given array
15
16
17    for coordinate in list_of_coordinates:
18        number_of_bit_flip = r(1, 31) # Amount of noise introduced in an individual coded pixel
19        index_of_noise = s(range(128), number_of_bit_flip) # A list containing the indexes of the elements to
20        flip in a single coded pixel
21        noised_coded_pixel_i = coordinate[0]
22        noised_coded_pixel_j = coordinate[1] * 128
23        noised_coded_pixel = coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j +
24        128] # Coded pixel to corrupt
25
26        # Corruption of the coded_pixel by flipping its entries from 1 to -1 and conversly:
27        for index in index_of_noise:
28            if noised_coded_pixel[index] == 1:
29                noised_coded_pixel[index] = -1
30            else:
31                noised_coded_pixel[index] = 1
32
33        # Exchanging the image's coded pixel with the corrupted pixel:
34        noised_coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j + 128] =
35        noised_coded_pixel
36
37    return noised_coded_array
```

Figure 8: Correctable error introducing function

This function takes in a list of random coordinates in the image for which it will apply a random noise that we will later be able to correct. That is, respecting the bound given by the theory above, and since our code is of length 128 then this function introduces an number of bit flips that is at most 31.

3.5.2 Detectable Error

```
Title

1 def detection_type_error(coded_array, list_of_coordinates):
2
3     ''' This function takes in 2 arguments and returns an array with some Detectable noise
4     introduced in the respective places listed in the (list_of_coordinates) variable
5
6     1- coded_array: type = np_array. represents the array with its entries representing the
7     hadamard coded colors of the original image
8
9     2- list_of_coordinates: type = list. represents a list of randomly generated tuples with
10    entries ranging in the between the dimension of the image that is they represent coordinates
11    of individual pixels in the image that will be perturbed in the transmission '''
12
13    noised_coded_array = coded_array.copy()
14
15    for coordinate in list_of_coordinates:
16        number_of_bit_flip = r(1, 63) # Amount of noise introduced in an individual coded pixel
17        index_of_noise = s(range(128), number_of_bit_flip) # A list containing the indexes of the elements to
18        flip in a single coded pixel
19        noised_coded_pixel_i = coordinate[0]
20        noised_coded_pixel_j = coordinate[1] * 128
21        noised_coded_pixel = coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j +
22        128] # Coded pixel to corrupt
23
24        #Corruption of the coded_pixel:
25        for index in index_of_noise:
26            if noised_coded_pixel[index] == 1:
27                noised_coded_pixel[index] = -1
28            else:
29                noised_coded_pixel[index] = 1
30
31        # Exchanging the image's coded pixel with the corrupted pixel:
32        noised_coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j + 128] =
33        noised_coded_pixel
34
35    return noised_coded_array
```

Figure 9: Detectable error introducing function

This function takes in a list of random coordinates in the image for which it will apply a random noise that we will later be able to detect. That is, it introduces an number of bit flips that is at most 63.

3.5.3 Non Detectable Error

```
Title

1 def non_detectable_error(coded_array, list_of_coordinates):
2     ''' This function takes in 2 arguments and returns an array with some NON-Detectable noise
3         introduced in the respective places listed in the (list_of_coordinates) variable
4
5     1- coded_array: type = np_array. represents the array with its entries representing the
6         hadamard coded colors of the original image
7
8     2- list_of_coordinates: type = list. represents a list of randomly generated tuples with
9         entries ranging in the between the dimension of the image that is they represent coordinates
10        of individual pixels in the image that will be perturbed in the transmittion '''
11
12    noised_coded_array = coded_array.copy()
13
14    for coordinate in list_of_coordinates:
15        number_of_bit_flip = r(64, 128) # Amount of noise introduced in an individual coded pixel
16
17        if number_of_bit_flip == 128: # If the numbers of bits to flip is exactly the length of the code it
18            means that we will flip all the vector's entries from 1 to -1 and conversly which is excatly the same as
19            multiplying by -1
20            noised_coded_pixel_i = coordinate[0]
21            noised_coded_pixel_j = coordinate[1] * 128
22            noised_coded_pixel = coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j
23            + 128]
24            noised_coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j + 128] = -
25            noised_coded_pixel
26
27        index_of_noise = s(range(128), number_of_bit_flip) # A list containing the indexes of the elements to
28        flip in a single coded pixel
29        noised_coded_pixel_i = coordinate[0]
30        noised_coded_pixel_j = coordinate[1] * 128
31        noised_coded_pixel = coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j +
32        128] # Coded pixel to corrupt
33
34        #Corruption of the coded_pixel:
35        for index in index_of_noise:
36            if noised_coded_pixel[index] == 1:
37                noised_coded_pixel[index] = -1
38            else:
39                noised_coded_pixel[index] = 1
40
41        # Exchanging the image's coded pixel with the corrupted pixel:
42        noised_coded_array[noised_coded_pixel_i, noised_coded_pixel_j : noised_coded_pixel_j + 128] =
43        noised_coded_pixel
44
45    return noised_coded_array
```

Figure 10: Caption

This function takes in a list of random coordinates in the image for which it will apply a random noise that we will later NOT be able to detect. That is, it introduces an number of bit flips that is greater or equal to 64. However, we note that there is a non zero probability to flip the bits of a vector in such a way that it becomes another vector of the code, in which situation, the received vector is decoded as if no error has occurred thus making the latter undetectable.

3.6 Channel's Noise

```
1 def noise(color_coded_array, image_shape, noise_amount= .60, type_of_noise = "Mixed"):
2
3     ''' This function takes in 3 arguments a color_coded_array, the amount of noise,
4     length_of_code and type_of_noise
5
6     1 - color_coded_array: type = numpy-array. represents the array of a color-coded gray-scaled
7     image
8
9     2 - image_shape: type = tuple. represents the shape of the image
10
11     3 - noise amount: type = float. represents the amount of pixels that will be corrupted
12     in the picture. the function generates as much random tuples with entries ranging in between
13     the dimensions of the array and those represent the coordinates of the pixels that will be
14     corrupted.
15
16     4 - type_of_noise: type = str in {Correction, Detection, Mixed}. This parameter determines
17     the type of the noise that will be applied to the picture. '''
18
19     number_of_noised_pixels = int(image_shape[0]*image_shape[1]*noise_amount)
20
21     # Generating a list of coordinates for pixels to be corrupted:
22     coordinate_list = s(list(p*[list(range(image_shape[0])), list(range(image_shape[1]))])),
23     number_of_noised_pixels) # Creating a list of random tuples with entries ranging in the array's dimension
24
25     # Introducing noise in the array:
26     # Correction type:
27     if type_of_noise == "Correction":
28         return correction_type_error(color_coded_array, coordinate_list)
29
30     # Detection type:
31     elif type_of_noise == "Detection":
32         return detection_type_error(color_coded_array, coordinate_list)
33
34     # Lost information type:
35     elif type_of_noise == "Lost":
36         return non_detectable_error(color_coded_array, coordinate_list)
37
38     # Mixed type:
39     elif type_of_noise == "Mixed":
40
41         while True:
42             distribution = np.random.uniform(size=3)
43             distribution /= np.sum(distribution)
44             sublists = []
45             try:
46                 sublists = construct_sublists(list(range(1, 11)), distribution)
47                 if (len(sublists) == 2) or ([ ] in sublists):
48                     next
49             except (ValueError):
50                 next
51
52         correction_coordinate_list = sublists[0] # List of correction typed errors in the image
53         detection_coordinate_list = sublists[1] # List of Detection typed errors in the image
54         non_correctable_error = sublists[2] # List of non detectable errors in the image
55
56         return non_detectable_error(detection_coordinate_list(correction_type_error(color_coded_array,
57         correction_coordinate_list), detection_coordinate_list), non_correctable_error)
```

Figure 11: Noise function

This Function represents the noise of the transmission channel, that is, it simulates the noise caused by physical phenomenon. It does that by introducing random bit flips at random pixels of the picture.

The user has the possibility to choose the type of noise that will be introduced in the picture, the possibilities are among: correctable type of noise, detectable type of noise, lost information type of noise and the Mixed type.

We note that by default the type of noise is mixed and the way it works is the following:

the program starts by taking a list of random coordinates of pixels then generates a sample of proportions (a list of 3 numbers summing to 1). After that it partitions the list of coordinates with respect to that sample making use of the `construct` `sublists` function.

3.7 Decoder

```

1 def decoder(color_coded_array, image_shape):
2
3     H_code = hadamard_decode(8)
4     codes = [eval(code) for code in list(H_code.keys())]
5
6     computing_array = color_coded_array.copy()
7     computing_array = computing_array.reshape(image_shape[0]*image_shape[1], 128)
8
9     decoded_vectors_list = []
10    for i in range(0, computing_array.shape[0]):
11        received_vector = computing_array[i]
12
13        if list(received_vector) in codes:
14            decoded_vectors_list.append(H_code[str(list(received_vector))])
15
16        else:
17            distances_dictionary = dict([(str(index), list(np.array(index) + received_vector).count(0)) for
18index in codes])
19            min_distance = min(distances_dictionary.values())
20
21            if min_distance <= 31:
22                min_vector = [vector for vector in list(distances_dictionary.keys()) if
23distances_dictionary[vector] == min_distance][0]
24                decoded_vectors_list.append(H_code[min_vector])
25
26            elif 32 <= min_distance <= 63:
27                list_of_possibilities = [vectors for vectors in list(distances_dictionary.keys()) if
28distances_dictionary[vectors] == min_distance]
29                decoded_vector = c(list_of_possibilities)
30                decoded_vectors_list.append(H_code[decoded_vector])
31
32            else:
33                decoded_vectors_list.append(r(0, 255))
34
35    decoded_array = np.array(decoded_vectors_list)
36    decoded_array = decoded_array.reshape(color_coded_array.shape[0], color_coded_array.shape[1] // 128)
37
38    return decoded_array

```

Figure 12: Caption

This function takes in an array of the transmitted picture and decodes it in the following way:

First we generate the Hadamard Decoding dictionary as well as a list of all the rows

of the Hadamard matrix that we will use to compare with the received vector.

Secondly, We calculate the received vector's Hamming distance with all the rows of the matrix and store the latter to a dictionary in order to be able to recover the closest vector later on.

After going through all the vectors we simply return the minimum of the list containing all the distances and use that as a determinant of the type of the error that had occurred that is:

If the minimum distance is at most 31 then we know this error is recoverable then we decode it to the vector it gave this distance with.

If the minimum distance is at most 63 then we know that at least 2 vectors will be equally spaced of the received vector. Thus, the approach we take is to generate a list of all those vectors that are tied and randomly choose one from them and use it to decode. Finally, if the minimum distance exceeds 64 then we know we cannot recover this information to which we decided to generate a random number and decode the received vector randomly.

4 Testing On Mariner Missions spacial Snapshots

In 1952, The Mariner 2 Mission to Venus sent back very promising pictures about the planet's geography, ground and Terrain. The process to bring back those spacial snapshot back to earth's laboratories was based on Hadamard Codes in the following, we will take some pictures from NASA's spacial imaging data base, encode them, noise them then we'll see the effect of the decoding process and how much of the image had been recovered.

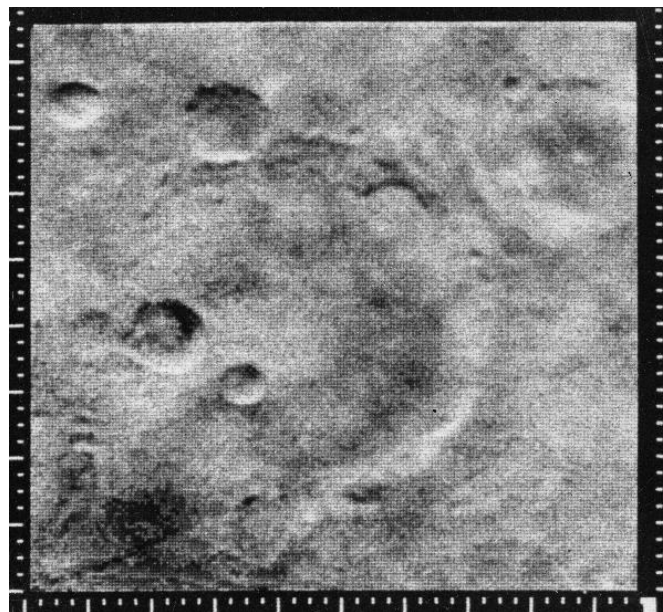


Figure 13: Venus Surface taken by Mariner 4

In this case we will noise 50% of the image and to have a look at the places where the image had been corrupted where we will highlight correctable errors with white pixels, detectable error with grey pixels and lost error with black pixels. in the latter exemple we have:

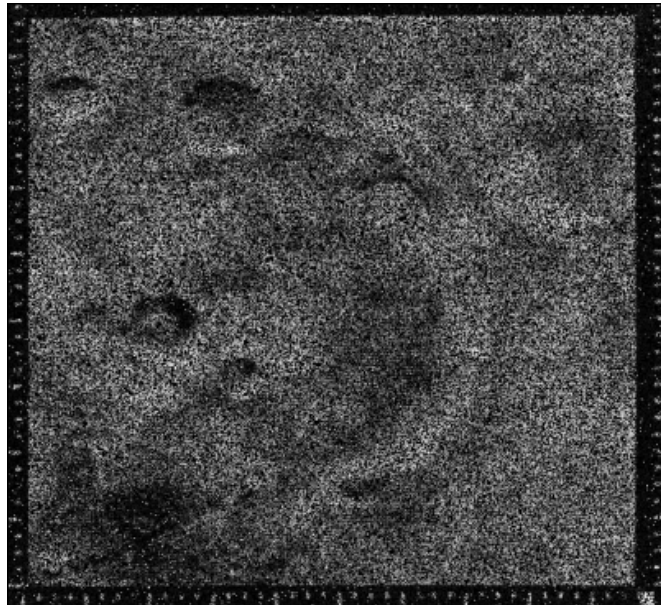


Figure 14: Venus surface at 50% mixed noise

To further illustrate some places where the noise had different types the following figure shows 50 places of each color labeled as such: Green circle refers to a correctable type error, Orange refers to a detectable type error, Red refers to a lost information.

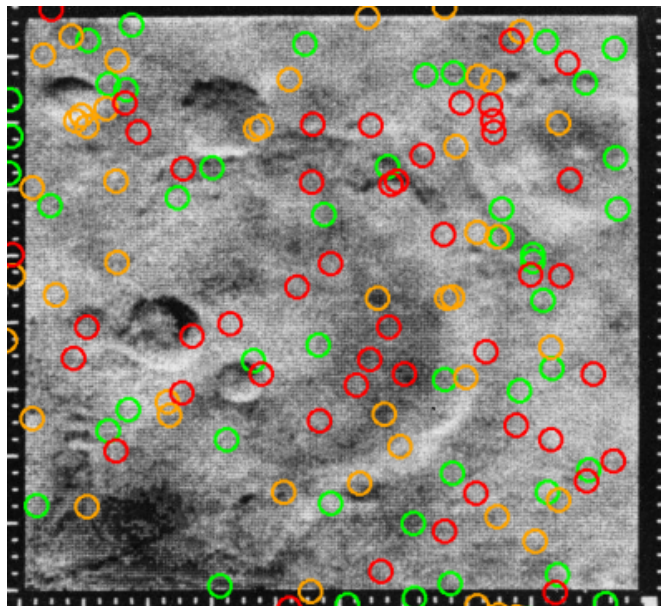


Figure 15: Venus surface with 150 errors circled

Finally, after the decoding process, we counted the percentage of the image that had been lost during the transmission and for 50% random noise added to the picture, only 12.168% of it were not recovered yielding the figure bellow:

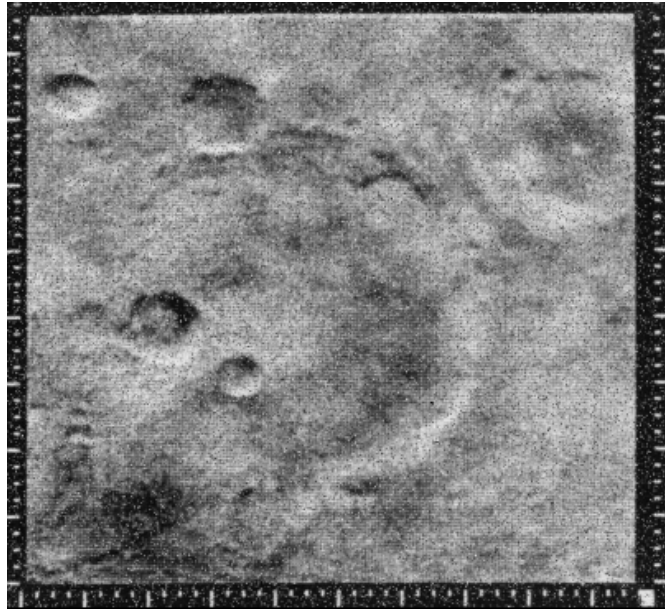
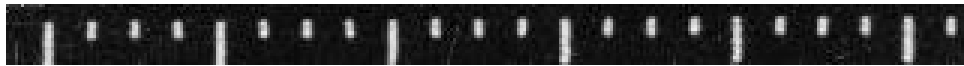


Figure 16: Decoded Image

for a closer look at the state of the received image we zoom on the boundaries of the picture yielding the figure bellow:



(a) Original Boundary



(b) Received boundary

5 Testing on Colored Images From Mariner 10

In this section we will test the algorithm on a colored picture that was sent in by the Mariner 10 mission for which we will first separate all three layers of color and encode each at a time right before noising the information and recovering the picture.

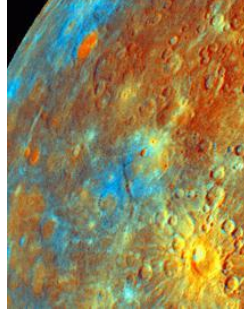
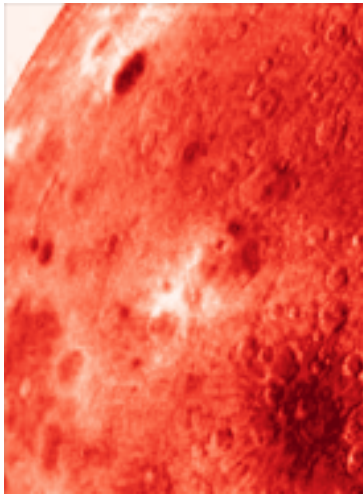
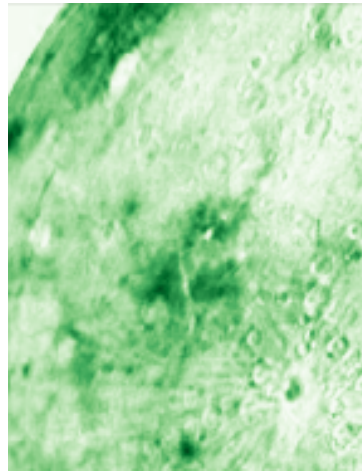


Figure 18: Mariner 10 colored picture of Mercury

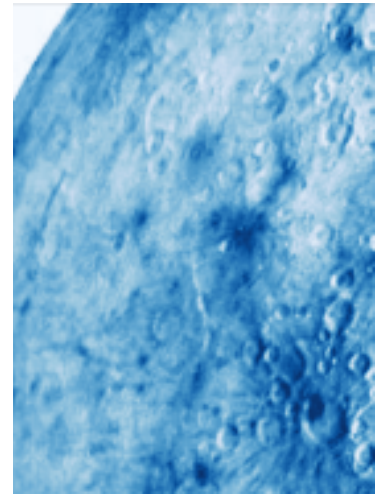
the colored layers of this picture are illustrated in the figure bellow:



(a) Red Layer



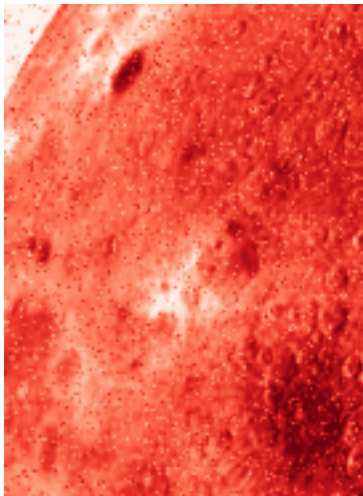
(b) Green Layer



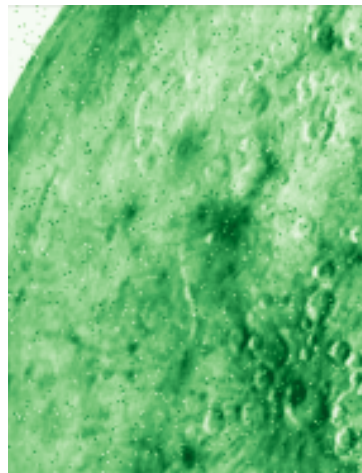
(c) Blue Layer

Figure 19: Color layers of the picture

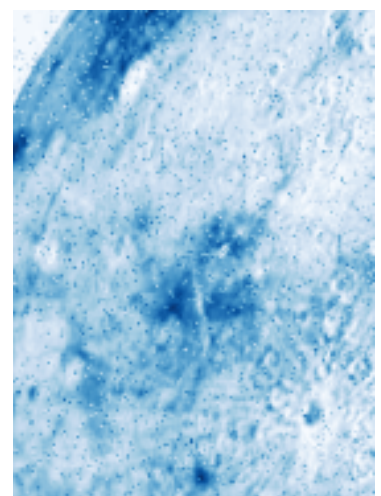
In this case, 30% of each of the color layers were corrupted and the recovery process yields the following figure



(a) Noised Red Layer



(b) Noised Green Layer



(c) Noised Blue Layer

Figure 20: Color layers after transmission

Finally, after stacking the image back together, the final received image is illustrated below yielding:

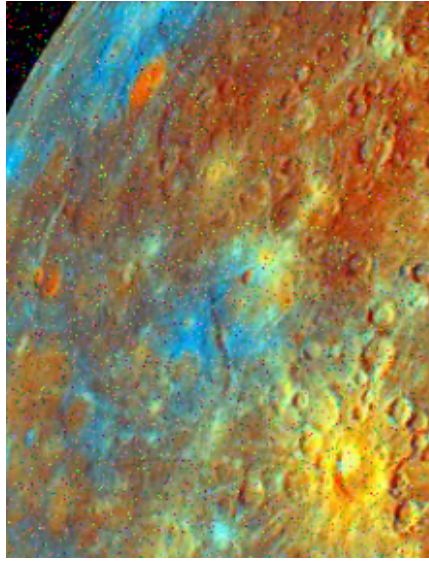


Figure 21: Received Mercury picture

6 Managerial Report

To resume all the work that has been presented, we were motivated by the need to receive images from outer space without them being noised by physical phenomenon, and for that matter, we have developed a code that is based on Hadamard matrices that permits us to catch errors when the latter happen in our image that is, if a pixel is perturbed during the transmission, we would be able to detect that error and correct it in some particular cases that are dictated by the length of the code we chose in the start. Hence making us able to at least recover the crucial parts of the picture and a general idea of the surface of the different planets the Mariner missions were distant to travel to.

References

- [1] .Chemlal, Algebra and Coding lectures, Higher National School of Mathematics (NHSM) <https://classroom.google.com/u/1/c/NjIy0TQ5MzkzNjQ4>
- [2] Jet Propulsion Laboratories, NASA Picture Database <https://www.jpl.nasa.gov/images?query=mariner&page=1>.

"The important achievement of Apollo was demonstrating that humanity is not forever chained to this planet and our visions go rather further than that and our opportunities are unlimited."

Neil Armstrong

