1 Overview

In traditional data compression, measurements are first taken, and then the information is compressed by discarding components with small coefficients, ensuring the overall vector remains largely unchanged. Compressive sensing, however, asks a fundamental question: *Could we have directly sampled only this significant portion of the initial signal?* This approach is particularly valuable in scenarios where minimizing the number of measurements is crucial—for example, in MRI scans for infants, where reducing exposure to radiation is essential, or in quantum measurements, where limiting back-action effects is a priority.

At first glance, compressive sensing appears to challenge Nyquist's theorem, which states that the sampling rate must be at least twice the signal's highest frequency to avoid aliasing. However, compressive sensing ingeniously circumvents this requirement by leveraging an important assumption: the sparsity of the signal being measured. By exploiting this sparsity, compressive sensing achieves efficient sampling and reconstruction, even with fewer measurements than traditional methods demand.

Vector $\mathbf{s}$ *is k-sparse if it has at most k non-zero elements.*

This is a fair assumption as all natural images and signals are sparse in some domain, and if we can obtain the necessary information in this sparse domain we can easily transform back to the potentially non-sparse domain of interest via a Fourier, DFT or wavelet transform depending on the nature of the signal. for the rest of these notes, the following variables will be used :
$\mathbf{x} \in \mathbb{R}^n$ is the signal we are interested in
$\mathbf{s} \in \mathbb{R}^n$ is the sparse signal, tied to $\mathbf{x}$ simply by $\mathbf{x} = \Psi\mathbf{s}$ where Ψ is the transformation matrix.
$\mathbf{y} \in \mathbb{R}^m, m << n$ is the measurement vector obtained using the measurement matrix Θ

$$\mathbf{y} = \Phi\mathbf{x} = \Phi\Psi\mathbf{s} = \Theta\mathbf{s} \tag{1}$$

Θ is an $(m \times n)$ under-determined linear system, with infinite number of solutions. This is where the sparsity assumption comes in. We are looking for the sparsest $\mathbf{s}$ that satisfies $\mathbf{y} = \Theta\mathbf{s}$, or:

$$\mathbf{s} = \arg\min(||\mathbf{s}'||_0), \ \ satisfying \ \mathbf{y} = \Theta\mathbf{s}' \tag{2}$$

Note: $||\mathbf{x}||_p = (\sum_i^n |x_i|^p)^{\frac{1}{p}}$ *and thus the* l_0 *norm corresponds to the number of non-zero elements in* $\mathbf{x}$*, which is exactly the thing we want to minimise.*
However minimising the l_0 norm is no trivial task, it is actually an np-hard problem, so instead we minimise the l_1 norm, which is a convex problem and is computationally feasible whilst still providing the desired result with high accuracy.

Here we worked with the convex minimisation problem:

$$\min_{\mathbf{s}}(||\mathbf{y} - \Theta s||_2 + j||\mathbf{s}||_1) \tag{3}$$

where j is a Lagrange multiplier. The first term has to do with data-fitting whereas the second encourages sparsity. The optimal value of j depends on the sensing matrix used.

2 Coherence

Coherence of matrix Θ *is defined as:* $\mu(\Theta) = \max_{1 \leq i \leq j \leq n} \frac{|\langle \theta_\mathbf{i}, \theta_\mathbf{j} \rangle|_1}{||\theta_\mathbf{i}||_2 ||\theta_\mathbf{j}||_2}$

(where θ are coloumns of matrix Θ)

A sensing matrix is an over-complete dictionary, an under-determined linear system. The aim is to collect the maximum amount of information whilst sampling as little as possible. The amount of information obtained by a row in the matrix depends on the inner product of the rows of the sensing matrix. If two rows are linearly dependent that means they pick up the same information from the signal so the second time we sample along the direction we obtain no new information. We want to span as large a subspace of the signal space as possible, and coherence is a measure of that. Therefore minimising the coherence of the sensing matrix is crucial for performance.

2.1 SVD

SVD can be used in the following way: After SVD is applied, the singular values of the matrix can be artificially brought close to unity and the new sensing matrix can be recalculated. This will ensure the norm doesn't change much, satisfying RIP to a lower value of δ. This is particularly useful if the sensing matrix is formed with a random process, but could inhibit the learning abilities of a Bayesian matrix. I will experiment to see if such concern is indeed reasonable. Another idea is for $\Theta = U\Sigma V^t$ let V^t be the basis transformation to sparse domain, set singular values to unity as before and construct U from learned data .This idea was tested and did not turn out to have a significant effect on the accuracy of the signal reconstruction. Since theory shows that this definitely does imply RIP, the conclusion ,is that the accuracy of our estimations were not limited by RIP (or other words the sparsity).

3 Signal reconstruction of 5 tone signal

A random vector of length 128 with 5 non-trivial entries under noise from Gaussian distribution about 0 with standard deviation 0.05 was selected as original semi-sparse input signal.

$\mathbf{x} = \mathbf{s} + \mathbf{e}$ where $\mathbf{x}$ is the sensed signal, $\mathbf{s}$ is a 5-sparse signal and
$e \sim_{i.i.d} N(0, 0.05)$

Following graphs were constructed to visualise the performances of individual matrices for the same input signal, all with 4 times compression (sensed vector had length 32.)

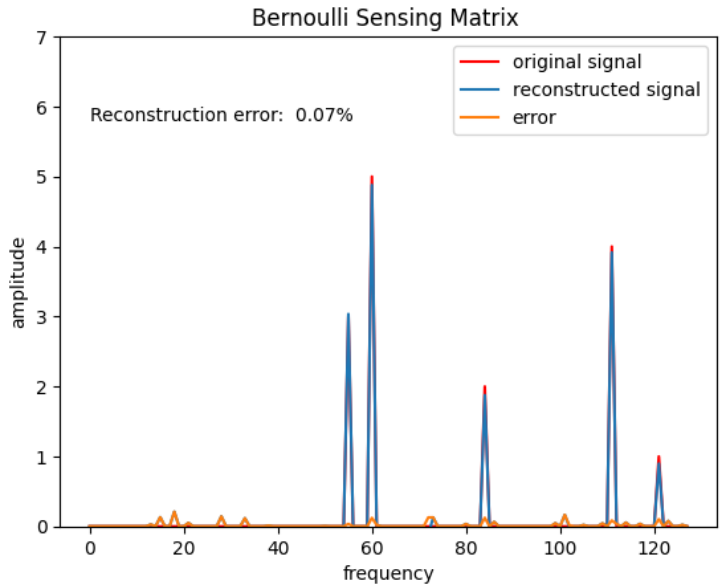


Figure 1: Matrix with random 0 and 1 entries

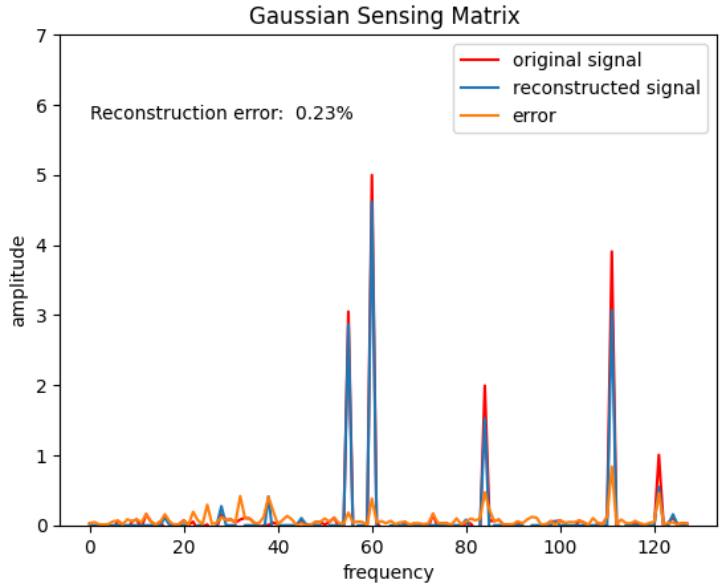


Figure 2: Matrix with input values pulled from a Gaussian about 0, with standard deviation 0.5

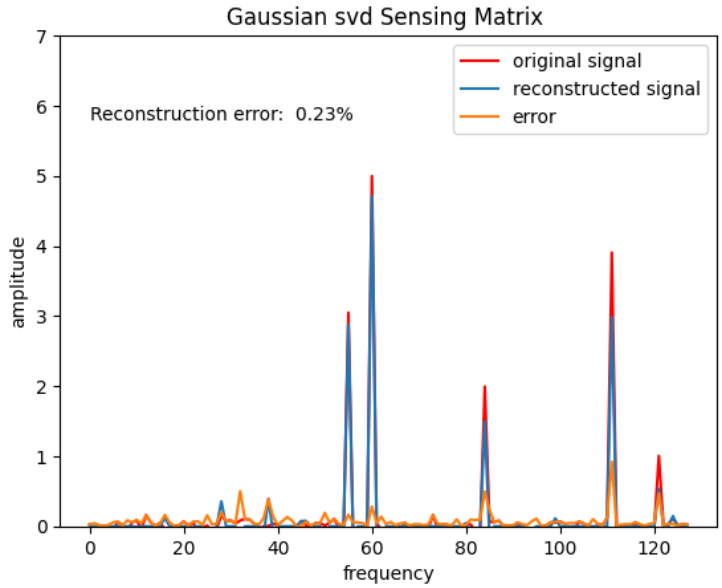


Figure 3: Gaussian matrix modified by setting singular values to $1 + \epsilon$, $\epsilon \sim_{i.i.d} N(0, 0.001)$

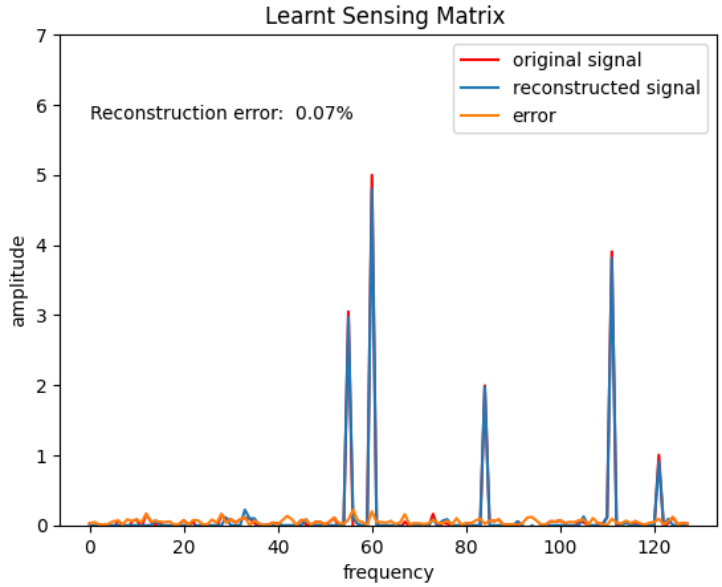


Figure 4: Matrix with learnt data as coloumns

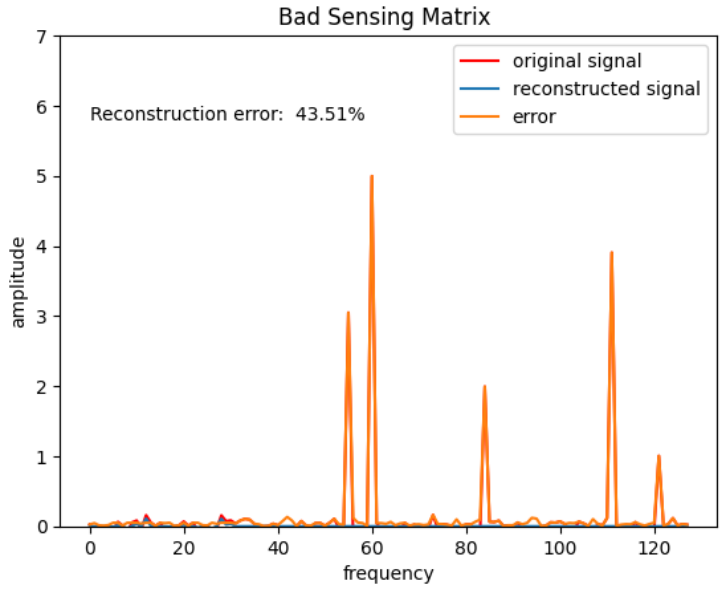


Figure 5: Identity matrix with zero padding.

Remark: Reconstruction values change considerably within the range $< 1\%$

for different input signals, it is hard to clearly compare the performances of individual matrices.
The Bayesian-learnt matrix (Figure 4) overall has the best performance, and the modified Gaussian (Figure 3) with singular values set to $1 + \epsilon$ tended to perform slightly better than the original Gaussian (Figure 2) but the difference was very small, which is expected since the singular values of the Gaussian were mostly within this range already.

4 Effect of degree of compression on the error

Figure 6 shows how the size of the measurement vector influences the error in the signal reconstructed by different sensing matrices. In this case the original signal was of length 128. As expected, the smaller the size of the measurement vector, the less information obtained about the original signal, but remarkably at sizes much smaller than the original signal the convex minimisation returns a reliable signal, and beyond about 4 times compression the reconstructed signal seems to converge to the original signal for all sensing matrices.
Notably, the sensing was done on a noisy signal (outlined in the python file) convergence was observed in much shorter measurement vectors when there is only trivial noise present.

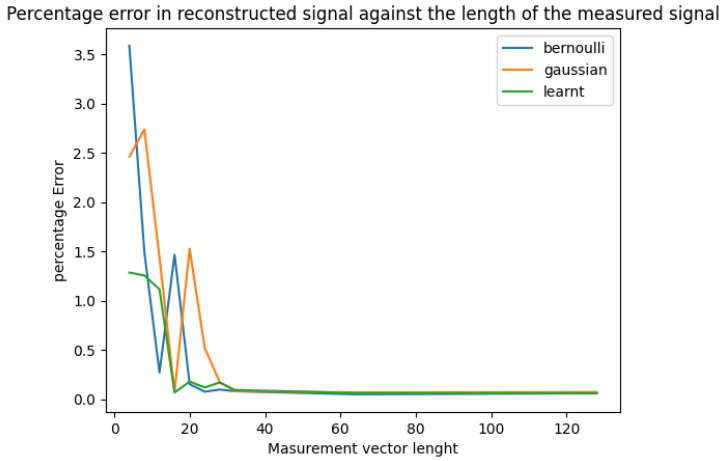


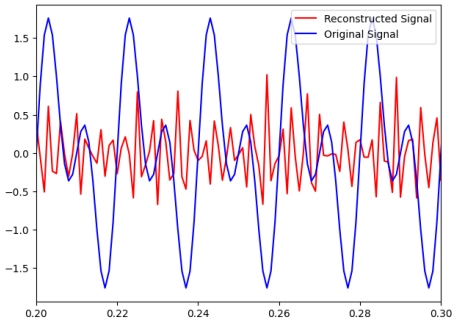
Figure 6: Effect of number of measurements on error

The following is the python code to produce the given graph.

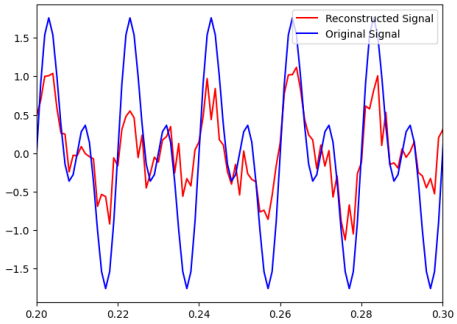
```
#comparing_effect_of_compression
#functions and matrices as defined in appendix

ms=[4,8,12,16,20,24,28,32,64,128] #dimensions of measured vector

learnt_matrices=[]
bernoulli_matrices=[]
gaussian_matrices=[]
for m in ms:                      #creating sets of measurement matrices with the desired dimensions
 b_m = np.random.choice([1, 0], size=(m, len(x)))
 bernoulli_matrices.append(b_m)
 g_m = np.random.normal(0, 0.5, size=(m, len(x)))
 gaussian_matrices.append(g_m)
errors_g=[]
errors_b=[]
errors_l=[]

for m in range(len(bernoulli_matrices)):
 learning_data=[]
 for _ in range(128):                          #128 samples of x
  learning_data.append(add_noise(x,0.05))
  learnt_ys=[]
 for i in range(128):
  learnt_ys.append(bernoulli_matrices[m] @ learning_data[i])   #using the bernoulli matrix to measure all samples

 learnt_k=np.column_stack(learnt_ys)                           #measured samples are used as learnt data to construct the new sensing matrix
 learnt_matrices.append(learnt_k)




for i in range(len(bernoulli_matrices)):
 sol_b= solve(bernoulli_matrices[i],bernoulli_matrices[i] @ x_noisy,1)
 sol_g= solve(gaussian_matrices[i],gaussian_matrices[i] @ x_noisy,10)
 sol_l= solve(learnt_matrices[i],learnt_matrices[i] @ x_noisy,10)

 errors_l.append(error_value(sol_l,x))
 errors_b.append(error_value(sol_b,x))
 errors_g.append(error_value(sol_g,x))

plt.plot(ms,errors_b,label='bernoulli')
plt.plot(ms,errors_g,label='gaussian')
plt.plot(ms,errors_l,label='learnt')
plt.legend()
plt.xlabel('Masurement vector lenght')
plt.ylabel('percentage Error')
plt.title('Percentage error in reconstructed signal against the length of the measured signal')
plt.show()
```

5 Fourier transformed signal

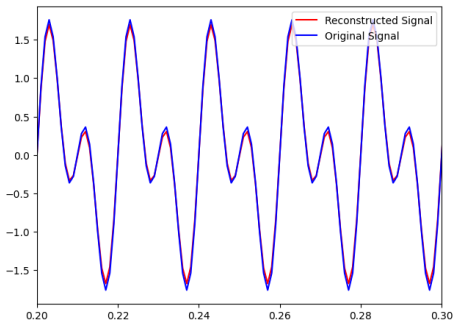
When the originial signal is not sparse in the given domain, it can be transformed into a domain that is sparse for the convex minimisation problem.
Here we considered a two tone signal, which is not sparse ,in time domain but is in frequency domain.



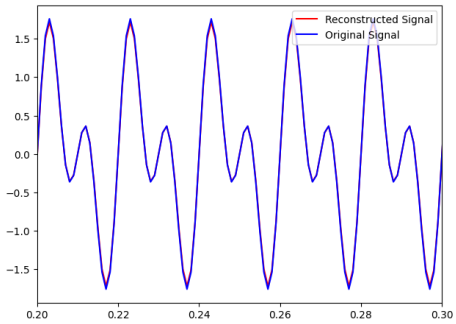
(a) x100 compression. SNR = -0.58 dB



(b) x50 compression. SNR = 4.00 dB



(c) x40 compression. SNR = 26.61 dB



(d) x20 compression. SNR = 32.33 dB

```

import numpy as np             #import necessary libraries
import scipy.fft
import matplotlib.pyplot as plt
import cvxpy as cp

# Parameters
fs = 1000  # Sampling frequency (Hz)
T = 1.0    # Duration (seconds)
t = np.linspace(0, T, int(fs * T), endpoint=False)  # Time vector

# Two-tone signal
f1, f2 = 50, 100  # Frequencies of the tones (Hz)
signal = np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)            #original
     signal we are interested in

# Compute FFT of the signal
s = scipy.fft.fft(signal)                    # fourier trasnform of the signal.IDeally
     two delta peaks, hence a 2 sparse vector

# Compute frequency bins
freqs = scipy.fft.fftfreq(len(signal), d=1/fs)

# Plot the original signal
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, signal)
plt.title("Two-Tone Signal in Time Domain")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()

# Plot the FFT (magnitude spectrum)
plt.subplot(2, 1, 2)
plt.stem(freqs[:len(freqs)//2], np.abs(s[:len(s)//2]), basefmt=" ")
```

```
plt.title("FFT of the Signal (Frequency Domain)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.grid()
plt.tight_layout()
plt.show()


sensing_matrices=[]
measurements=[]
reconstructed_signals=[]
Ms= ['10','15','20','50']               #different lenghts of measured vector. the nyquist limit is                                    #at 200, so all of these are sub-nyquist

for M in [10,15,20,50]:
 sensing_matrix = np.random.randn(M, len(signal))   #gaussin sensing matrix
 sensing_matrices.append(sensing_matrix)
# Compressed measurements
 y = np.dot(sensing_matrix, signal)                 #y = Phi @ x = Phi @ Psi @ s
 measurements.append(y)
F = np.fft.fft(np.eye(len(signal)))                 # Fourier transform matrix
F_inv = np.linalg.inv(F)


for i in range(3):
 s = cp.Variable(len(signal), complex=True)      #sparse matrix we are interested in
 x_reconstructed = cp.real(F_inv @ s)            # x will be the inverse fourier transform of s
 objective = cp.Minimize(
 cp.norm1(s) + cp.norm2(sensing_matrices[i] @ x_reconstructed - measurements[i])**2)
  #equal weight has been given the data fitting and sparsity in this case


# Define and solve the optimization problem
 problem = cp.Problem(objective)
 problem.solve()

# Recovered sparse vector
 s_recovered = s.value



# Reconstruct the signal
 reconstructed_signal = np.real(np.fft.ifft(s_recovered))
 reconstructed_signals.append(reconstructed_signal)
 plt.plot(t,reconstructed_signal,color='red')
 plt.plot(t,signal,color='blue')
 plt.xlim(0.2,0.3)
 plt.title(f'Compression To {Ms[i]}'),
 plt.tight_layout()
 plt.xlabel('Time (s)')
 plt.ylabel('Amplitude')
 plt.show()



 mse = np.mean((signal - reconstructed_signal) ** 2)
 snr = 10 * np.log10(np.sum(signal ** 2) / np.sum((signal - reconstructed_signal) ** 2))
 print(f"M={Ms[i]}: MSE={mse:.5f}, SNR={snr:.2f} dB")
```

Notice the sampling is done on the original signal, and the minimisation problem uses no knowledge of the original signal beyond the sample measurement, and it aims to obtain sparse signal $\mathbf{s}$,which should be the fourier transform of the original signal. After obtaining $\mathbf{s}$, one simply uses inverse fourier transform to reconstruct the original signal.

Remark: The Nyquist limit for the signal at hand was 200 Hz, and all the measurements were well beyond this limit, demonstrating the capabilities of compressive sensing.

The limitations of compressive sensing is also portrayed in this experiment, the cases with 100 and 50 fold compression clearly give sub-optimal results,

while slightly higher sampling rate of 40 returns a reconstructed signal perfectly acceptable for most purposes.
Lastly this experimentation focused on the transformation process rsther than abilites of matrices, hence it is likely that the accuracy of reconstruction could be improved with the use of tailored Bayesian matrices.

A Appendix : Construction of necessary functions, matrices and the original signal

```
import numpy as np                    #import necessary libraries
import matplotlib.pyplot as plt
import cvxpy as cvx

def solve(theta,y,j):            # the function to solve for sparse vector s given sensing
      matrix theta, sensed vector y, and constant j
 lenght=theta.shape[1]
 sol=cvx.Variable(lenght)
 objective = cvx.Minimize((j*cvx.norm(y-(theta @ sol),2)**2+(cvx.norm(sol,1))))
 prob = cvx.Problem(objective)
 result = prob.solve(verbose=True)
 return sol.value

def error_value(x,sol):                #function to return percentage error
 return np.linalg.norm(sol-x) /(np.linalg.norm(x))

def error_vector(x,sol):
  return abs(sol-x)

def plot(x,sol,error_vector,error):
  plt.plot(x,color='red',label='original signal')
  plt.plot(sol,label='reconstructed signal')
  plt.plot(error_vector, label ='error')
  plt.xlabel('frequency')
  plt.ylabel('amplitude')
  plt.legend(loc='upper right')
  plt.text(0.01,5.8,f"Reconstruction error:  {error:.2f}%")
  plt.ylim(0,7)
  plt.show()

def analyse(theta,j,x):
 y= theta @ x
 sol= solve(theta,y,j)
 error= error_value(x,sol)
 er_vector= error_vector(x,sol)
 plot(x,sol,er_vector,error)
  print(f"Reconstruction error: {error:.6f}")

x = np.concatenate((np.arange(1,6), np.zeros(128-5)))
x = x[np.random.permutation(128)]                        # original signal without noise
def add_noise(x, sd):
    noise = np.random.normal(0, sd, size=x.shape)        # Generate noise
    return x + noise                                     # Add noise

x_noisy= add_noise(x,0.05) # noisy semi-sparse vector

#different sensing matrices: bernoulli, gaussian, gaussian svd, learnt
bernoulli = np.random.choice([1, 0], size=(32, len(x)))     #noisy error =0.12%,
     noiseless error= 0.02%
gaussian = np.random.normal(0, 0.5, size=(32, len(x)))     #noisy error =0.21%   ,
     noiseless error= 0.05%
# Perform SVD
def svd_modification(matrix):
    U, Sigma, Vt = np.linalg.svd(matrix)
    for i in range(len(Sigma)):
        Sigma[i] = 1 +np.random.normal(1,0.001)
    Sigma_modified = np.hstack((np.diag(Sigma), np.zeros((matrix.shape[0], matrix.shape[1]
         - Sigma.shape[0]))))
    return np.dot(U, np.dot(Sigma_modified, Vt))

gaussian_svd= svd_modification(gaussian)

learning_data=[]
for _ in range (128):
 learning_data.append(add_noise(x,0.05))
learnt_ys=[]
for i in range(128):
  learnt_ys.append(bernoulli @ learning_data[i])

learnt = np.column_stack(learnt_ys)       #noisy error = 0.085% ,noiseless error= 0.009%
identity= np.eye(32)
bad_m=np.hstack((identity,np.zeros((32,128-32))))         #noisy error = 2.7% ,noiseless
     error= 2.7%
```

B Appendix : Optimising the Lagrange Multiplier

I found the following to be the optimal Lagrange multiplier j for the convex minimisation problem for the sensing matrices produced.

Matrix	**Optimal Lagrange Multiplier order of magnitude**
Bernoulli	1
Gaussian	10
Modified Gaussian	100
Learnt	10

Table 1: Optimal orders of magnitude for Lagrange Multiplier for different sensing matrices when dealing with the noisy signal

Remark : These values are specific to the matrices and the signal we examined, and not necessarily the optimal values in all situations. This is aimed only as a rough guide if one pleases to experiment with the code here.
As would be expected, in the case with a noiseless signal, larger the Lagrange multiplier for the data fitting term, better solution is obtained.