Compressed Sensing

University of North Texas Department of Electrical Engineering
EENG 5850.001 - Image and Video Communication

Nicholas Chiapputo - UNT ID: 11135627

Instructor: Kamesh Namuduri, PhD

April 30, 2021

CONTENTS

I. Introduction

With the increase in high quality sensors (in particular image, video, and audio), the storage capacity requirements have likewise increased. When capturing a 4K image with a resolution of 3,840 x 2,160 pixels, nearly 25 MB of data is required (assuming 24-bits per pixel, or 8-bits per color channel in an RGB image). However, despite this massive amount of data captured at the sensor, almost immediately the device throws away a significant portion of it in the process known as compression. Instead, why do we not simply capture or measure the portion of the data that we don't want to throw away?

When sampling a signal, the common rule is that we must sample at a rate of *at least* twice the frequency of the signal. This rule is known as the Nyquist-Shannon sampling theorem. As with most rules, however, there is an exception to this widely followed rule. The assumption that we must sample at twice the frequency only holds in the case where the signal we are sampling is a broadband signal. That is, all the frequencies within the bandwidth of the signal are present. However, if the signal we are measuring is "compressible", or can it can be sparsely represented in the frequency domain, then we can sample beneath the Nyquist rate and still achieve perfect reconstruction. To take advantage of this caveat in the theorem, we can utilize a technique known as compressive sensing (CS).

In CS, the original signal is subsampled at some sample value $M << N$ where N is the original number of samples. Reconstruction is done by finding a solution to an underdetermined linear system $\boldsymbol{y} = \boldsymbol{\Phi x}$. In this system, y is the Mx1 subsampled, or compressively sensed, signal and $\boldsymbol{\Phi}$ is the MxN sampling or sensing matrix. The Nx1 vector $\boldsymbol{x}$ is a K-sparse (only containing K non-zero values) representation of the signal in some domain where the signal is sparse (e.g., the spectral domain). While there are many methods of finding an optimal solution to the underdetermined linear system $\boldsymbol{y} = \Phi \boldsymbol{x}$, this report only focuses on the ℓ_1-norm minimization definition:

$$\begin{aligned} min \quad & \|\boldsymbol{x}\|_1 \\ s.t. \quad & \boldsymbol{y} = \boldsymbol{\Phi x} \end{aligned} \tag{1}$$

In this report, Section II introduces and explains the fundamental ideas behind CS and recovery using the ℓ_1-norm. Section III discusses the methods of implementation used during research for this report in both MATLAB and Python for 1D and 2D signals and explains the implementation source files attached in the Appendices and located publicly at [15]. This is followed by the experiments performed and the results obtained from these implementations in Section IV. Finally, the paper is concluded in Section V.

II. Compressed Sensing

The principle component of compressed sensing (CS) is the underdetermined linear system shown in Equation 2 below. The matrix $\boldsymbol{\Phi}$ is commonly substituted with A, or the entire equation is rewritten as $\boldsymbol{Ax} = \boldsymbol{b}$, depending on the field of publication.

$$\boldsymbol{y} = \boldsymbol{\Phi x} \tag{2}$$

Underdetermined linear systems of this form are characterized by having more equations than unknowns, meaning that there are an infinite number of solutions to the system. The Mx1 vector $\boldsymbol{y}$ contains the M randomly selected samples from the original signal of N samples where $M << N$. The MxN matrix $\boldsymbol{\Phi}$ is known as the "dictionary" or "sensing" matrix. The Nx1 vector

$\boldsymbol{x}$ contains the frequency components of the signal after reconstructing from the M samples. This vector is k-sparse, meaning that there are only k non-zero values (where $k << M << N$) and the remaining are at or close to zero. Figure 1 shows a visual representation of the system along with the colored k non-zero values in the $\boldsymbol{x}$ vector.

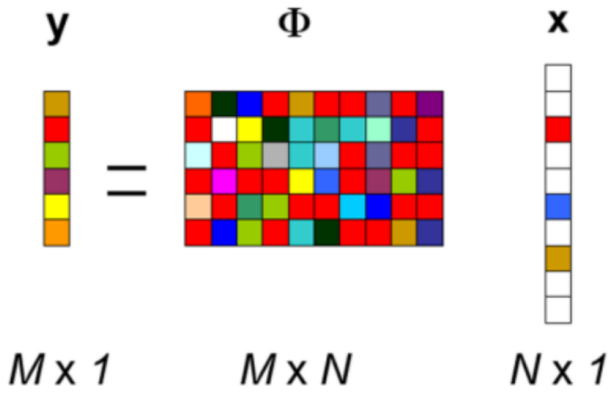


Fig. 1: Visual representation of the $\boldsymbol{y} = \boldsymbol{\Phi x}$ underdetermined linear system for compressive sensing. [3]

To construct the $\boldsymbol{y}$ vector, we first take M random samples from the target source (e.g., an audio signal or an image). Ideally, we would know *a priori* which measurements to take from the signal to best compress the image. However, since this is not possible to know, we instead take a random sample of the measurements to preserve the frequency components of the image.

The dictionary, $\boldsymbol{\Phi}$, is derived such that, when multiplied by a k-sparse matrix $\boldsymbol{x}$ made of components in the frequency domain, it will yield $\boldsymbol{y}$, the vector of M samples from the original signal. This means that the $\boldsymbol{\Phi}$ matrix performs both a transformational (transformation from spectral to temporal domains) role and a subsampling (from N to M) role in the computation. To derive $\boldsymbol{\Phi}$, we first let $\boldsymbol{f}$ be the Nx1 target signal and $\boldsymbol{\phi}$ be the sampling matrix that chooses the M random samples in $\boldsymbol{y}$. The equation is as follows:

$$\boldsymbol{y} = \boldsymbol{\phi f} \tag{3}$$

Next, we define the transformation equation. With $\boldsymbol{f}$ still the Nx1 target signal in the temporal domain, $\boldsymbol{\psi}$ is the NxN transformation matrix that converts the Nx1 sparse spectral data in $\boldsymbol{x}$ to the temporal domain. This equation is as follows:

$$\boldsymbol{f} = \boldsymbol{\psi x} \tag{4}$$

Plugging $\boldsymbol{f}$ from Equation 4 into Equation 3, we get:

$$\boldsymbol{y} = \boldsymbol{\phi\psi x}. \tag{5}$$

Comparing this with the original equation in Equation 2, we can see that $\boldsymbol{\Phi} \equiv \boldsymbol{\phi\psi}$. Then, to construct $\boldsymbol{\Phi}$, we need to construct the transformation ($\boldsymbol{\psi}$) and sampling ($\boldsymbol{\phi}$) matrices. The transformation matrix is constructed as the inverse DCT of $\boldsymbol{I}_N$, the NxN identity matrix. Then, $\boldsymbol{\psi}$ can simply be implemented by selecting the random indices as in the M subsampled signal

$\boldsymbol{y}$. Now that we have generated the $\boldsymbol{\Phi}$ matrix and random sampled to create the $\boldsymbol{y}$ vector, we must solve for an optimal $\boldsymbol{x}$.

To solve the underdetermined linear system in Equation 2, the ideal solution is to minimize $\|\boldsymbol{x}\|_0$ (the number of non-zero elements in $\boldsymbol{x}$). However, this is an NP-hard problem. That is, it is computationally inefficient and does not have a polynomial-time solution. Instead, we can perform an approximation using a higher-order ℓ_p-norm minimization such as ℓ_1 or ℓ_2. The ℓ_1-norm, or the least absolute deviations, is less easily calculated than the ℓ_2-norm, or least squares. However, the ℓ_1-norm attempts to find the most sparse solution. To truly compare which approximation is best suited for CS, we can perform a simple example.

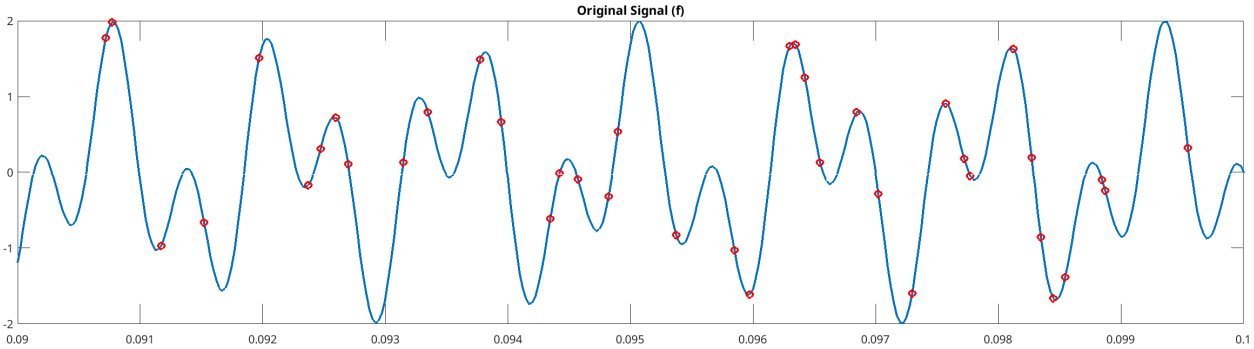


Fig. 2: Zoomed audio signal with $N = 5000$ samples of the equation $f(t)$ =sin$(1394 \cdot \pi \cdot t)$ + sin$(3266 \cdot \pi t)$ (blue line) and $M = 500$ subsamples (red circles).

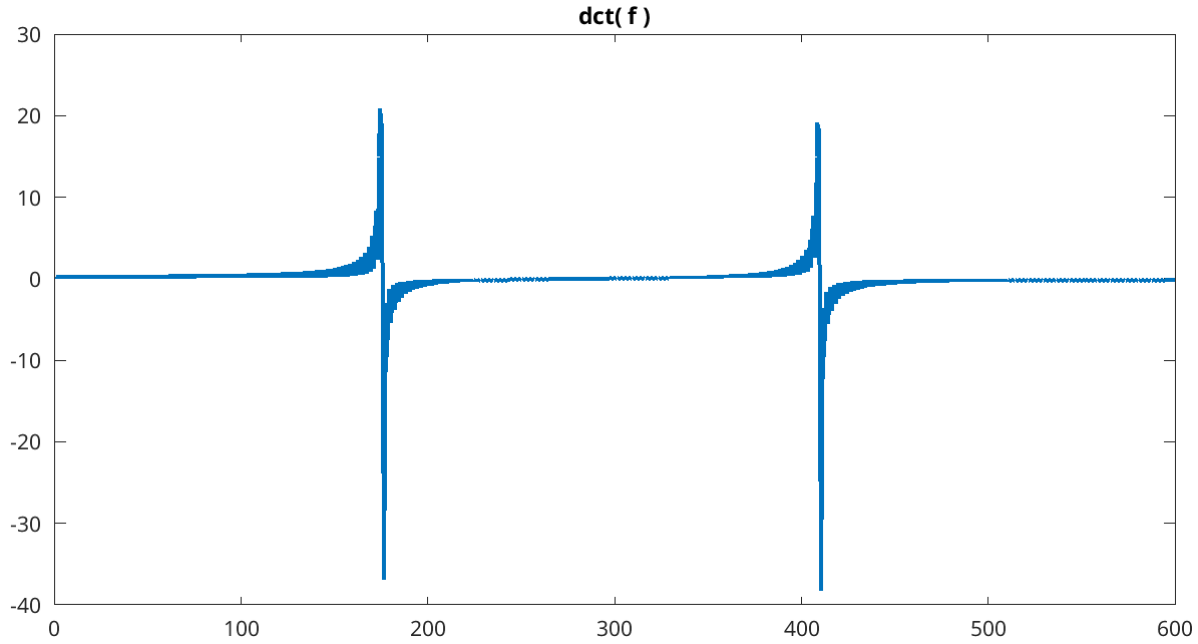
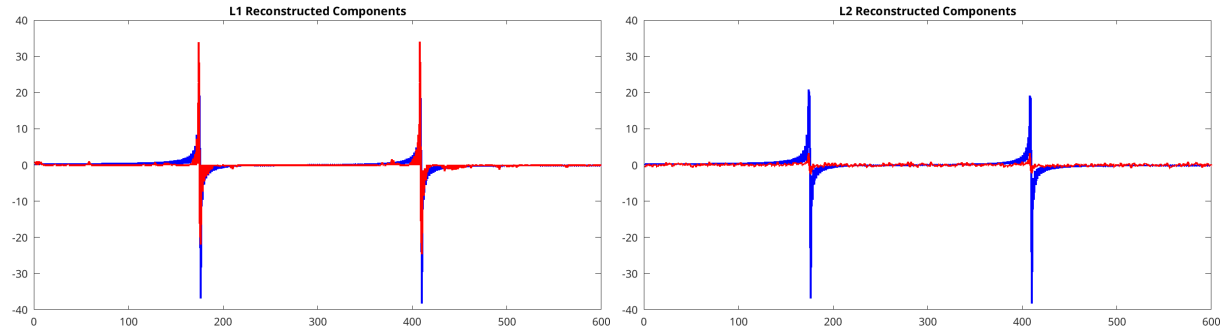


Fig. 3: DCT components of $f(t)$ =sin$(1394 \cdot \pi \cdot t)$ + sin$(3266 \cdot \pi t)$ zoomed in to focus on two main peaks.

The signal in Figure 2 shows $N = 5000$ samples of the equation $f(t)$ =sin$(1394 \cdot \pi \cdot t)$ + sin$(3266 \cdot \pi t$ in the blue curve with a 10% subsampling of $M = 500$ shown as red circles on

the line. The DCT components are then shown in Figure 3. This shows the two main frequency components of the signal, while the remaining components are at or near zero, meaning the signal is sparse in the spectral domain.



(a) (b)

Fig. 4: Comparison of the original frequency components (blue) using (a) L1-norm and (b) L2-norm (least squares) with the reconstructed values in red.

We can then use both the ℓ_1- and ℓ_2-norm and compare the resulting DCT components found using convex optimization in the Nx1 $\boldsymbol{x}$ vector. The results of this test can be seen in Figure 4. These results show that, because the ℓ_2-norm is more sensitive to extreme outliers due to the squared error, it results in DCT components that are all relatively small with no outlying peaks. However, since the ℓ_1-norm does not punish outliers, it returns more sparse results with only two main peaks. The rest of the DCT components are at or near zero, showing a sparse output in the spectral domain. From this, we can determine that it is best to minimize the ℓ_1-norm and use Equation 2 as the constraint to a convex optimization problem. The problem is then reformulated as:

$$\begin{aligned} min \quad & \|\boldsymbol{x}\|_1 \\ s.t. \quad & \boldsymbol{y} = \boldsymbol{\Phi x}. \end{aligned} \tag{6}$$

A. 2D Signals

Implementation of CS using 2D signals (e.g., images) requires only a simple reformulation. Since the $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors are one-dimensional, we can restructure the 2D signals as 1D vectors by flattening them. That is, restructure the matrices into vectors by stacking the columns of the 2D data.

However, this can pose a significant problem during implementation. Consider a 2D signal of size 512x512. Flattening this into a vector sets $N = 512 \cdot 512 = 262,144$. If, for example, we wanted to subsample at 50%, this sets $M = 0.50 \cdot 262,144 = 131,072$. Then, the MxN $\boldsymbol{\Phi}$ matrix contains 34.4 *billion* values. Assuming the data is stored as an 8-bit unsigned integer (the common data type for one channel of an image), this equates to over 34 Gigabytes of memory necessary just to store the information. Of course, matrix multiplication is also very memory intensive, so the computation will need a huge amount of memory, also increasing the processing cost.

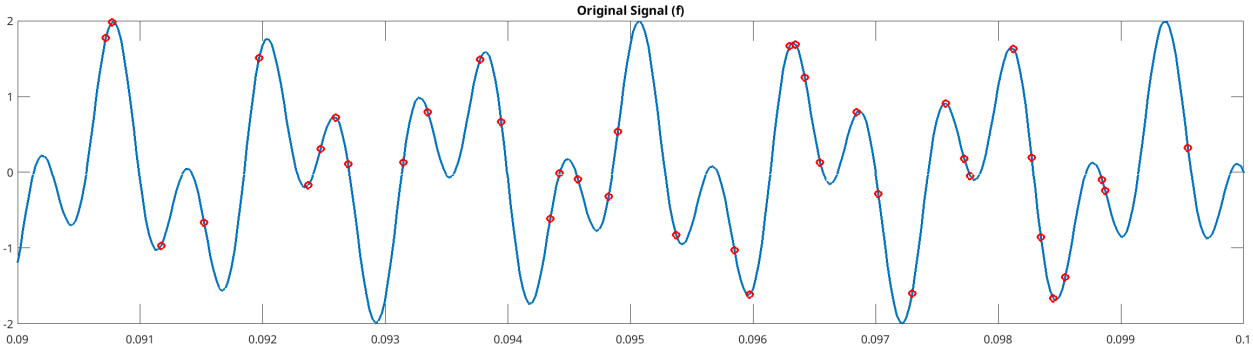
To solve this problem, we can utilize a variant of the Limited memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm called OWL-QN (Orthant-Wise Limited-Memory Quasi-Newton) developed by a Microsot Research time from [12]. The L-BFGS algorithm approximates

the BFGS algorithm which solves an unconstrained minimization problem if the objective function and its gradient are computable. The OWL-QN variant of L-BFGS minmizes a function $F(x)$ with the ℓ_1-norm using $F(x) + C\|\boldsymbol{x}\|_1$ where C is some constant.

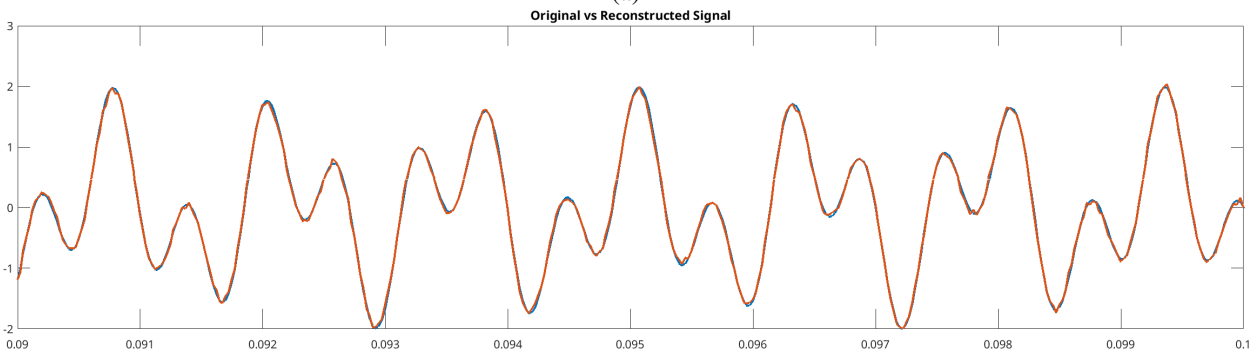
From the original FORTRAN implementation of L-BFGS, a C library (libLBFGS) is available from [13] and also includes the OWL-QN implementation among other features. The author of [1] used the Python package NumPy to write a wrapper for this C library for use in Python and published the open-source wrapper at [2]. The author also shows an example implementation of the library for CS and uses the least squares objective function ($\|\boldsymbol{\Phi x} - \boldsymbol{y}\|_2^2$). The gradient of this objective function is $2(\boldsymbol{\Phi}^T\boldsymbol{\Phi}x - \boldsymbol{\Phi}^T y)$.

III. Implementation

Through the process of performing the research for this report, the first implementation was written in MATLAB following the lecture from [7]. This implementation used the CVXR convex optimization library from [11] to perform the convex optimization problem. In addition, the implementations defined in the following section are hosted publicly at [15].



(a)

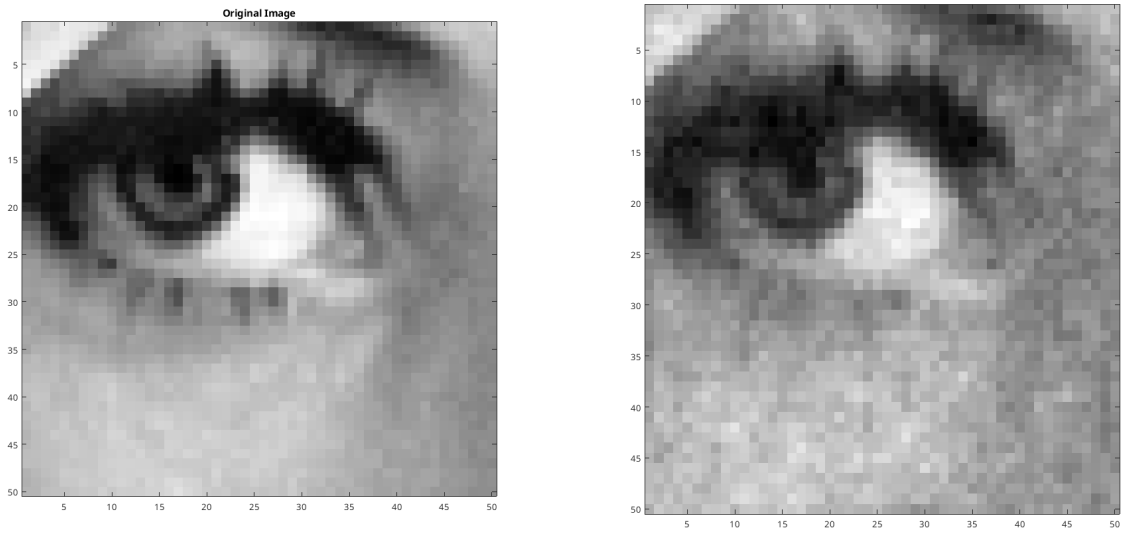


(b)

Fig. 5: (a) Original audio signal (blue line) ($f(t)$ =sin(1394 $\cdot$ π $\cdot$ t) + sin(3266 $\cdot$ πt)) of 5,000 samples with 500 subsamples (red dots) and (b) reconstructed signal (orange) over original signal (blue).

To test the one-dimensional implementation, an example CS problem was conducted using the audio signal $f(t)$ =sin(1394$\cdot\pi\cdot t$) + sin(3266$\cdot\pi t$) (the tone for the 'A' key on a phone). Figure 5 shows the initial audio sample of $N = 5000$ samples and subsampled at 10% with $M = 500$ samples followed by the reconstructed signal in orange overlaid on top of the original audio signal in blue. These figure shows that the CS implmentation in MATLAB works as expected

with only very small deviances in the result despite only using 10% of the samples. These deviances are likely caused by the natural inaccuracies of the ℓ_1 approximation method along with the magnitude of the compression. This implementation is found in the `cs_example.m` MATLAB script attached to this report and in Section A.



(a) (b)

Fig. 6: (a) Original 50x50 pixel portion of the Lena image and (b) the result of the MATLAB compressed sensing implementation after 50% subsampling.

After implementing one-dimensional CS successfully in MATLAB, we explored options for two-dimensional implementations. The resource at [9] was used as a basis for the 2D image CS implementation in MATLAB. Minor changes were implemented such as moving to the use of the CVXR library and adding extra logic to more easily test multiple images. However, this implementation is very memory-constrained, leading to very high system resource utilization. Because of this, only small portions of an image could be used. The images in Figure 6 show the original 50x50 pixel portion of the well-known Lena image on the left and the result of the compressed sensing implementation on the right after a 50% subsampling. From this, it can be seen that the implementation does work well, and that a reasonable quality image can be obtained with only 50% of the original samples with only some distortion. This implementation is found in the `cs_image_test.m` MATLAB script attached to this report and in Section B.

Despite these good results, even just this small portion of an image with only 50% of samples removed takes up to 30 seconds (on a system with an AMD 5600x processor and 32 GB of RAM) to compute. Increasing the size of the image, or the number of samples removed, polynomially increases the computation time and resources required. Even using full-sized 512x512 Lena image causes MATLAB to crash as it requests much more than the available system memory. To optimize this, we implemented the Python solution from [1] using the author's pyLBFGS library from [2]. The Python script is attached to this report in the script `cs.py` and in Section C.

This implementation fixes many of the system resource (both memory and processing) requirement problems of the MATLAB implementation. The maximum memory usage noted during

the processing approached 3 GB for a single image of size 5,472x2,976 and was around 300 MB for a 512x512 image. Since the Python script is single threaded, it naturally maxed out the processing capability of a single core of the CPU. There is some room for optimization here for images with multiple color channels as, since the channels are computed separately, a multi-threaded implementation could improve compute time on capable processors. Computation times were also greatly improved with a maximum of four minutes for a single image of size 5,472x2,976 at a sample rate of only 1% (thus requiring significant time to converge during optimization). For a 512x512 image, the computation time was as low as 0.4 and 6.2 seconds with sample rates of 100% and 1%, respectively.

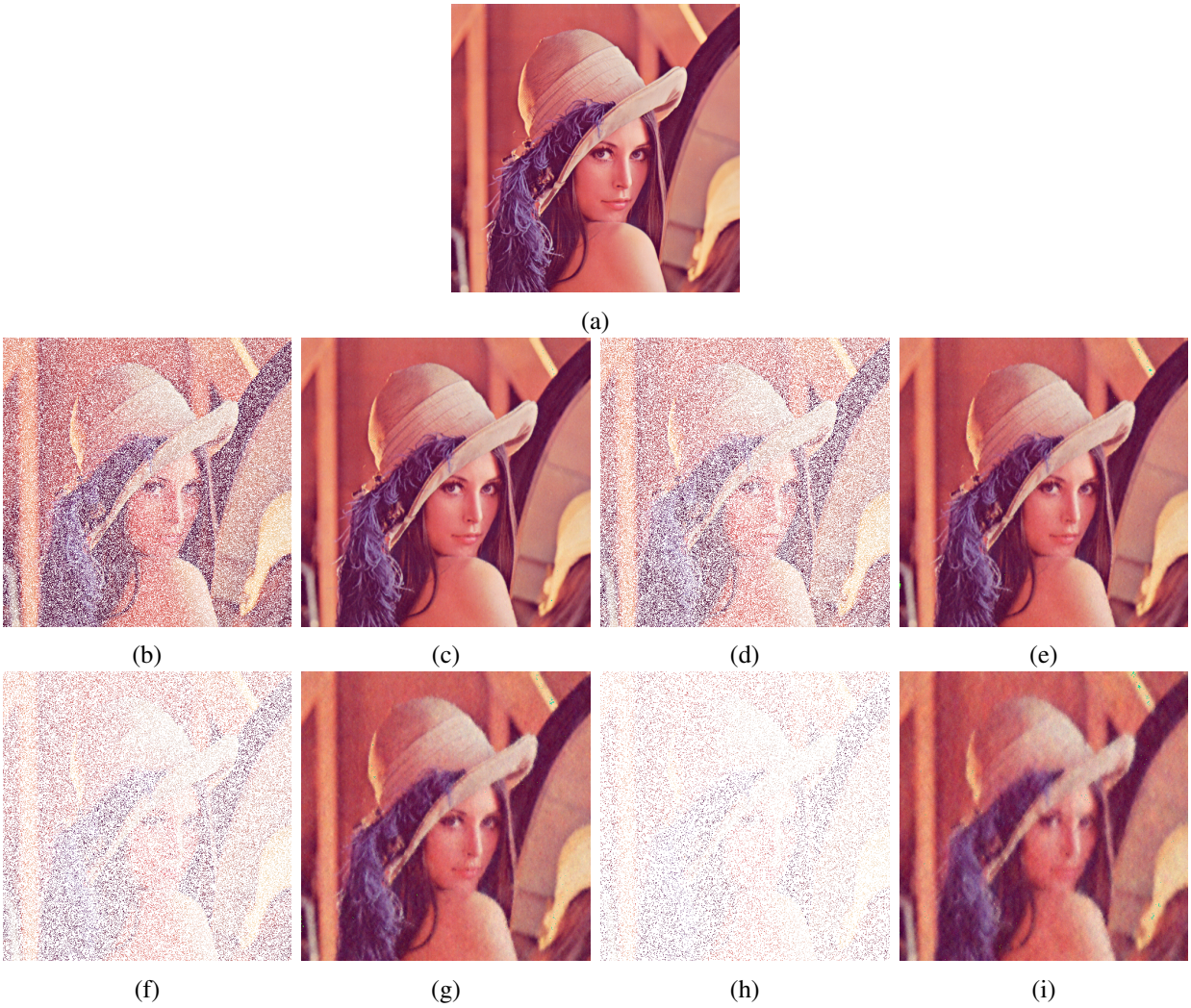
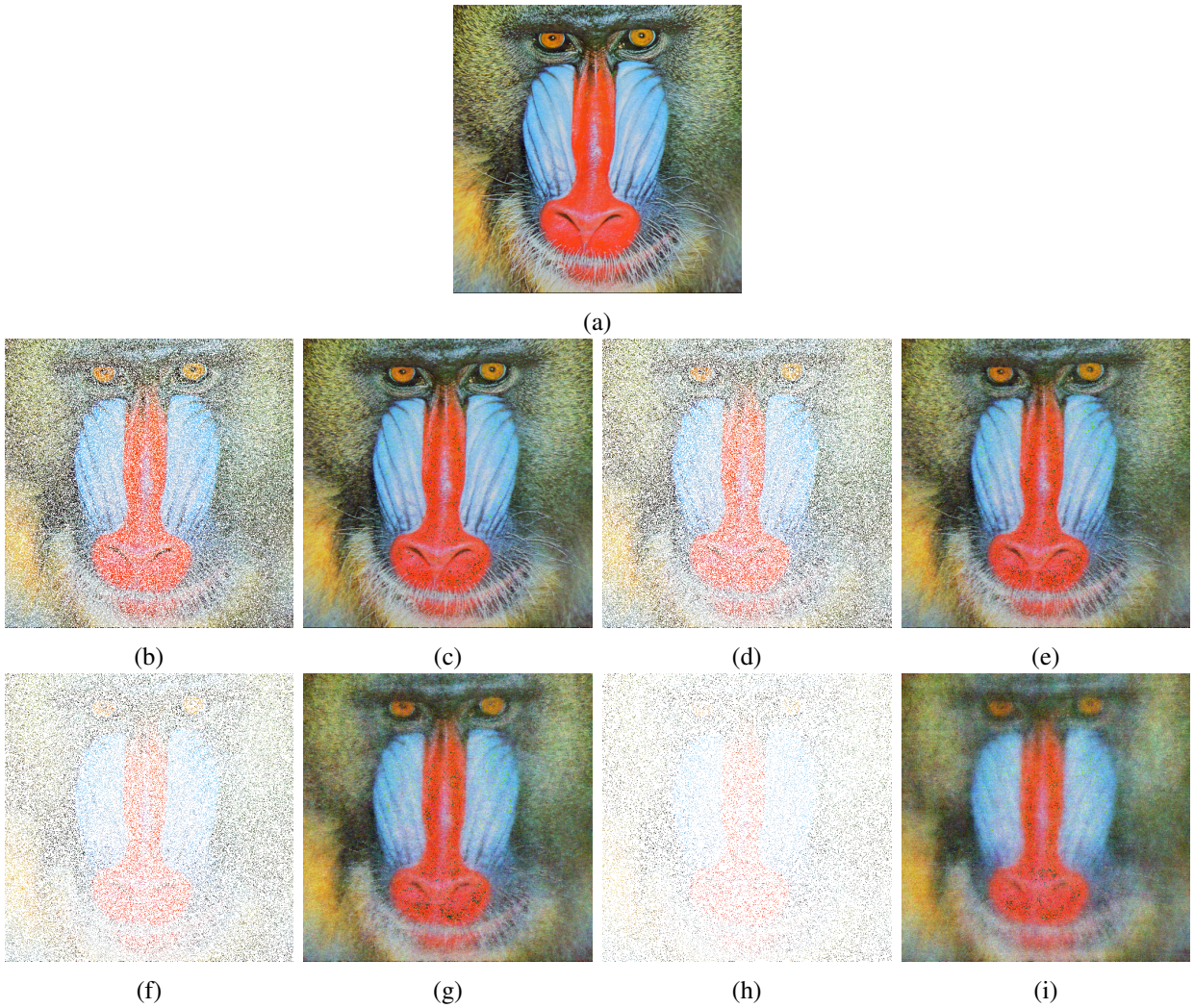


Fig. 7: Results of Python implementation using (a) original Lena image with (b) 60% mask, (c) 60% recovery, (d) 40% mask, (e) 40% recovery (f) 20% mask, (g) 20% recovery, (h) 10% mask, and (i) 10% recovery

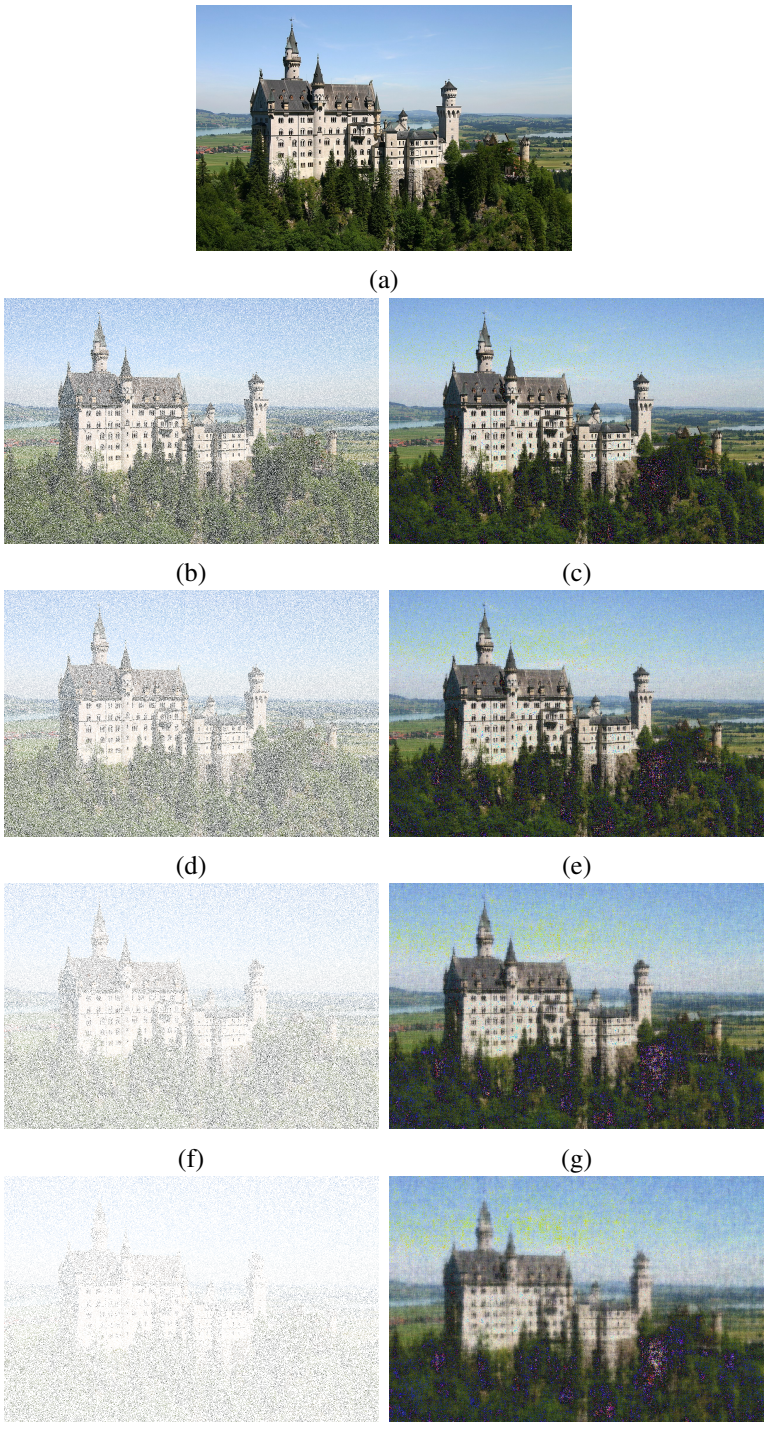
The results in Figure 7 show the original Lena image in Figure 7a followed by the masked images for 60%, 40%, 20%, and 10% of samples in Figure 7b, Figure 7d, Figure 7f, and Figure 7h, respectively. The constructed images are then shown in Figure 7c (60%), Figure 7e (40%), Figure 7g (20%), and Figure 7i (10%). These results show that even with a low sample

rate of 10%, we retrieve a very reasonable quality image. This experiment is repeated again below using the mandrill and Neuschwanstein images in Figure 8 and Figure 9, respectively.



(a) (b) (c) (d) (e) (f) (g) (h) (i)

Fig. 8: Results of Python implementation using (a) original mandrill image with (b) 60% mask, (c) 60% recovery, (d) 40% mask, (e) 40% recovery (f) 20% mask, (g) 20% recovery, (h) 10% mask, and (i) 10% recovery



(a) (b) (c) (d) (e) (f) (g) (h) (i)

Fig. 9: Results of Python implementation using (a) original Neuschwanstein image with (b) 60% mask, (c) 60% recovery, (d) 40% mask, (e) 40% recovery (f) 20% mask, (g) 20% recovery, (h) 10% mask, and (i) 10% recovery

IV. Results

Beyond visually identifying the quality of reconstructed images, we can also use objective measures to determine the quality of a reconstructed image against the ground truth (the original image with full samples). One such metric is the Structural Similarity Index Measurement (SSIM). The SSIM metric is a combination of measurements comparing luminance, contrast, and structure. The goal of this metric, and what makes it different from other metrics such as Mean Squared Error (MSE), is that it focuses on measuring the perceived change in structural information rather than simply measuring the absolute difference in pixel values. The goal of this metric is to identify the perceived quality of an image based on the ground truth original image. The SSIM value is a real value over the range $[0, 1]$ where a value of 1.0 indicates the image is very similar or the same to the ground truth.

Other metrics include MSE and Peak Signal-to-Noise Ratio (PSNR). The MSE metric is used to calculate the square of the error in pixel value between the test image and the ground truth. This metric is one of the most common used to measure image quality. The MSE value is a real value of 0 or greater where a lower value indicates greater image quality and less deviation from the ground truth.

PSNR is used to calculate the ratio of the maximal signal power (or the square of the maximum pixel value) to the MSE. The PSNR metric is measured in decibels (dB) and values generally range from 30 to 50 dB for average to good quality images where pixels are represented by eight bit integers. For this metric, a higher value indicates greater image quality as the MSE is lower while a lower value indicates greater deviation from the ground truth image.

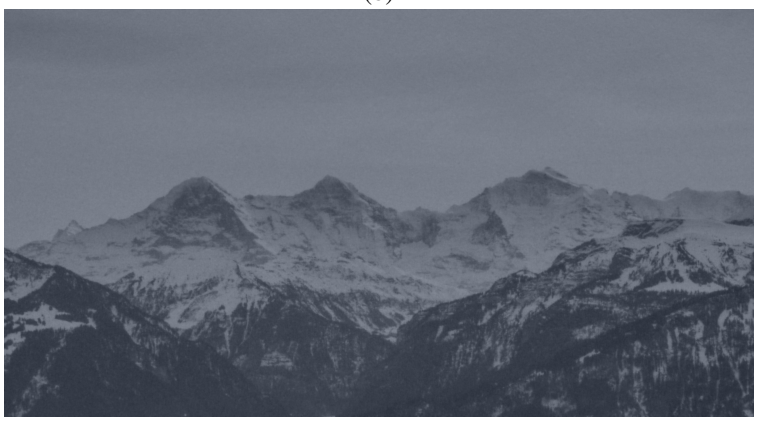
Sample	SSIM			MSE			PSNR		
Percentage	Lena	Mandrill	Neusch.	Lena	Mandrill	Neusch.	Lena	Mandrill	Neusch.
100%	0.978	0.993	0.933	5.226	6.991	157.099	40.949	36.685	26.169
90%	0.959	0.932	0.770	10.243	74.015	376.106	38.026	29.438	22.378
80%	0.937	0.870	0.648	15.987	154.363	617.950	36.093	26.245	20.221
70%	0.911	0.800	0.554	24.849	255.127	871.792	34.178	24.063	18.727
60%	0.879	0.725	0.476	34.154	368.031	1147.691	32.796	22.472	17.533
50%	0.840	0.643	0.407	50.981	497.685	1426.921	31.057	21.161	16.587
40%	0.792	0.555	0.344	70.938	642.209	1728.594	29.622	20.054	15.754
30%	0.726	0.457	0.281	102.855	790.105	2021.594	28.009	19.154	15.074
20%	0.640	0.349	0.217	157.736	966.865	2319.439	26.151	18.277	14.477
10%	0.506	0.227	0.149	275.905	1163.247	2599.152	23.723	17.474	13.982
1%	0.251	0.105	0.093	1067.362	1685.916	2760.950	17.848	15.862	13.720

TABLE I: Structural Similarity Index Measurement (SSIM), Mean Squared Error (MSE), and Peak Signal-to-Noise Ratio (PSNR) metric results for the three images shown in Figure 7 (Lena), Figure 8 (Mandrill), and Figure 9 (Neuschwanstein) with varying sample percentages using the Python implementation.

The results of the SSIM, MSE, and PSNR metrics with varying sample percentages, or M samples as a percentage of N original samples, for the Lena (Figure 7a), mandrill (Figure 8a), and Neuschwanstein (Figure 9a) images as the ground truth. These results first show some error found in the 100% sample scenario. This small error is due to the inaccuracies inherent with the ℓ_1 approximation method (due to it being an *approximation*). As expected, the image quality does decrease as the sample percentage goes down. It is notable, however, that the image quality increases much more rapidly once below approximately 30% samples. These results show that CS does perform very well for image compression down to 50% with decreasing image quality afterwards. At 10%, the images are still somewhat usable.

(a)

(b)



(c)

Fig. 10: CS reconstruction from (a) the original mountain image of size 5,472x2,976 with (b) a mask showing the retained 10% of samples and (c) the recovered image using the 10% of the original samples.

During testing, a low-contrast image was also used to test the quality of the CS compression with different types of images. The image shown in Figure 10a is the original ground truth source of size 5,472x2,976 with a low contrast ratio and the mask in Figure 10b shows the image with only 10% of the original pixels. This specific example shows that the CS implementation performs very well for low-contrast images, likely due spectral domain sparsity of the image.

V. Conclusion

Compressed (or Compressive) Sensing (CS) is a good tool for data compression that allows for the capture of significantly less data from the source (e.g., a sensor) for compression on the fly. This technique may be used instead of current techniques that capture very large amounts of data (for example, 24.8 MB for a 4K still image) before immediately compressing and throwing away a large portion of that data. This technique can be utilized in many fields such as data compression, medical imaging (e.g., MRIs), channel coding as an error correcting code technique, network routing, and many more.

In this report, we examined the ℓ_1-norm minimization (or Basis Pursuit) technique to solve the underdetermined linear system $\boldsymbol{y} = \boldsymbol{\Phi x}$ used to represent the CS problem. We first implemented the one-dimensional solution in MATLAB and tested it with an audio signal. After finding that the performance was very poor for two-dimensional data (i.e., images), we implemented a version in Python using a limited memory BFGS (L-BFGS) algorithm variant OWL-QN. This technique vastly increase the performance of the algorithm in regards to both memory and processing requirements, allowing me to test large images and have nearly all results complete in under 10 seconds. All implementations and the results are availably publicly and open-source at [15].

To objectively measure the quality of the reconstructed images at varying sample percentages, we used the Structural Similarity Index Measurement (SSIM), Mean Squared Error (MSE), and Peak Signal-to-Noise Ratio (PSNR) with three separate images at 11 different sample values. These results showed the reconstructed images retained good quality down to 30-50% sample percentages with still recognizable images at sample percentages as low as 1%. We also found that lower contrast images appear to have better resiliency to degradation and maintain their objective quality measurements at sample percentages of 1-10%.

References

[1] "Compressed Sensing in Python." Pyrunner. http://www.pyrunner.com/weblog/2016/05/26/compressed-sensing-python/ (retrieved Apr. 29, 2021).

[2] R. Taylor. "pylbfgs." bitbucket.org. https://bitbucket.org/rtaylor/pylbfgs/src/master/. (accessed Apr. 28, 2021).

[3] A. Hladnik, P. Saksida, "Compressed sensing and some image processing applications," in *Int. Symp. on Graphic Engineering and Design*, 2018, pp. 567-572. doi: 10.24867/GRID-2018-p68.

[4] P. C. Nahar, M. T. Kolte, "An introduction to compressive sensing and its applications," *Int. Jour. of Scientific and Research Publications*, vol. 4, no. 6, Jun. 2014. [Online]. Available: http://www.ijsrp.org/research-paper-0614/ijsrp-p3076.pdf

[5] P. Indyk. *Tutorial on compressed sensing (or compressive sampling, or linear sketching)* [PDF]. Available: https://people.csail.mit.edu/indyk/princeton.pdf

[6] W.-K. Ma. *Sparse optimization* [PDF]. Available: http://dsp.ee.cuhk.edu.hk/eleg5481/Lecture%20notes/13-%20compressive%20sensing/cs.pdf

[7] Nathan Kutz. *Compressive Sensing*. (Feb. 6, 2021). Accessed: Apr. 29, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=rt5mMEmZHfs

[8] L.-F. Cheng. (2014). *Compressive sensing* [PDF]. Available: http://3dvision.princeton.edu/courses/COS598/2014sp/slides/lecture20_Compressive_Sensing.pdf

[9] S. Gibson. "simple compressed sensing example." https://www.mathworks.com/matlabcentral/fileexchange/41792-simple-compressed-sensing-example (retrieved Apr. 29, 2021).

[10] Y. Chen. (2020). *Compressed sensing and sparse recovery* [PDF]. Available: https://www.princeton.edu/~yc5/ele520_math_data/lectures/compressed_sensing.pdf

[11] "CVX Research." CVX Research. http://cvxr.com/. (accessed Apr. 29, 2021).

[12] G. Andrew, J. Gao. "Scalable training of L1-regularized log-linear models," *Proceedings of the 24th International Conference on Machine Learning*, 2007, pp. 33-40, doi: 10.1145/1273496.1273501.

[13] "libLBFGS: a library of limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)." chokkan.org. http://chokkan.org/software/liblbfgs/. (accessed Apr. 29, 2021).

[14] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," In *IEEE Trans. on Image Proc.*, vol. 13, no. 4, pp. 600-612, Apr. 2004, doi: 10.1109/TIP.2003.819861.

[15] N. Chiapputo. "Compressive Sensing." github.com. https://github.com/NickChiapputo/compressive-sensing. (accessed Apr. 29, 2021).

APPENDIX A
MATLAB AUDIO SIGNAL CS SOURCE CODE

```
1  %clear all, close all, clc
2  clear_figs
3
4  % 'A' tone on a phone.
5  n = 5000;
6  m = 500;
7  t = linspace( 0, 1/8, n );
8  f = sin( 1394 * pi * t ) + sin( 3266 * pi * t );
9
10 figure( 1 );
11 plot( t, f, 'Linewidth', 2 );
12 title( 'Original Signal (f)' );
13
14 ft = dct( f);
15 figure( 2 );
16 plot( ft, 'Linewidth', [2] );
17 title( 'dct( f )' );
18
19
20 % Sub-sample the original signal.
21 temp = randperm( n );   % Random permutation of [n]
22 ind = temp( 1:m );      % Grab the first 500 random indexes.
23 tr = t( ind );           % Grab x-axis values at each index.
24 fr = sin( 1394 * pi * tr ) + sin( 3266 * pi * tr );
25
26
27
28
29 figure( 1 );
30 hold on
31 plot( tr, fr, 'ro', 'Linewidth', 2 );
32
33
34 % Figure out mapping from time to frequency domain.
35 D = dct( eye( n, n ) );
36 A = D( ind, : );
37
38
39 % Least Square fit.
40 % This won't fit properly. Plotting the DCT of this will show the
       Fourier
41 % components. They will all be very small because least square
      wants to
```

```
% make everything small. It won't want to make the proper
   coefficients
% large like they should be because that increases the "error".
%x = pinv( A ) * fr'; % Least square fit.


% Perform convex optimization to minimize the L1-norm (least
   absolute
% deviations) subject to the constraint Ax = fr'
cvx_begin
    variable x( n );
    minimize( norm( x, 1 ) );
    subject to
        A*x == fr.';
cvx_end

sig1 = dct( x );


figure( 3 );
plot( t, f, t, sig1, 'Linewidth', [2] );

% subplot( 2, 1, 1 );
% plot( t, f, 'Linewidth', [2] );
% subplot( 2, 1, 2 );
% plot( t, sig1, 'Linewidth', [2] );

title( 'Original vs Reconstructed Signal' );

% Show the Fourier components.
figure( 4 );
plot( ft, 'b', 'Linewidth', [2] );
hold on
plot( x, 'r', 'Linewidth', [2] );
title( 'Original vs Reconstructed Components' );
```

cs_example.m

APPENDIX B
MATLAB IMAGE CS SOURCE CODE

```
1  tic
2
3  % Read in the image.
4  I = imread( "lena.png" );
5
6  % Convert to grayscale if image is in RGB
7  % to convert from 3D matrix to 2D.
8  if ndims( I ) ~= 2
9      I = rgb2gray( I );
10 end
11
12
13 % Take a square portion of the image and convert to a 1-D double
       vector.
14 start_idx = 250;
15 end_idx = 299;
16 num_idxs = ( end_idx - start_idx ) + 1;
17 I = I( [ start_idx:end_idx ], [ start_idx:end_idx] );
18 x = double( I( : ) );
19 n = length( x );
20
21
22 % Generate pseudo-random Phi matrix of size mxn.
23 % m: Number of samples to take (e.g., m = n / 3 takes a third of
       the
24 % samples as the original image).
25 m = floor( n / 2 );
26 Phi = randn( m, n );
27
28
29 % We want to solve y = Phi * Psi * a for a, but we only know Phi
       and Psi.
30 % Since we also know that y = Phi * x, we can get y using this
       relation.
31 % Since Phi is an mxn matrix where m < n, we are also limiting
       the number
32 % of samples from the original image.
33 y = Phi*x;
34
35
36 % Calculate the product of the Phi and Psi matrices (named Theta)
       .
37 % This has to be done one column at a time to reduce system
       resource
```

```
% over-usage (the matrices can become very large and use a large amount of
% RAM).
Theta = zeros(m,n);
for idx = 1:n
%   fprintf( "Iteration %d\r", idx );

    tmp = zeros( 1, n );
    tmp( idx ) = 1;
    psi = idct( tmp )';
    Theta( :, idx ) = Phi * psi;
end
psi_1 = psi;

% Solve the convex optimization problem using the L1-norm with the
% constraint y = Theta * a.
cvx_begin
    variable a( n );
    minimize( norm( a, 1 ) );
    subject to
        Theta * a == y;
cvx_end


% Reconstruct the image.
psi = zeros( n );
for idx = 1:n
%   fprintf( "Iteration %d\r", idx );

    tmp = zeros( 1, n );
    tmp( idx ) = 1;
    psi( :, idx ) = idct( tmp )';
end

reconstructed_vector = psi * a;


%% Results
% Display the original image, Psi, Phi, and reconstructed result.
% Use 'axis image' for appropriately proportioned axes and a tight bounding.
figure( 2 )

subplot( 1, 2, 1 )
imagesc( reshape( x, num_idxs, num_idxs ) )
```

```
title( 'Original Image' )
axis image

subplot( 1, 2, 2 )
imagesc( reshape( reconstructed_vector, num_idxs, num_idxs ) )
title( 'L1 Minimization Result' )
axis image

colormap gray


toc
```

cs_image_test.m

APPENDIX C
PYTHON SOURCE CODE

```
1  import numpy as np
2  import matplotlib as mpl
3  import matplotlib.pyplot as plt
4  import scipy.optimize as spopt
5  import scipy.fftpack as spfft
6  import scipy.ndimage as spimg
7  import imageio
8  import cvxpy as cvx
9
10 from skimage.metrics import structural_similarity
11 from skimage.metrics import mean_squared_error
12 from skimage.metrics import peak_signal_noise_ratio
13
14 from pylbfgs import owlqn
15
16 from math import trunc
17 from time import time
18
19
20 def dct2( x ):
21   return spfft.dct( spfft.dct( x.T, norm='ortho', axis=0 ).T,
      norm='ortho', axis=0 )
22
23 def idct2( x ):
24   return spfft.idct( spfft.idct( x.T, norm='ortho', axis=0 ).T,
      norm='ortho', axis=0 )
25
26
27 def evaluate( x, g, step ):
28   # Goal is to calculate:
29   # (1) The norm squared of the residuals, sum( ( Ax - b ).^2 )
30   # (2) The gradient 2*A'*( Ax - b )
31
32   # Expand x columns first
33   x2 = x.reshape( ( nx, ny ) ).T
34
35   # Ax is just the inverse 2D dct of x2
36   Ax2 = idct2( x2 )
37
38   # Stack columns and extract samples
39   Ax = Ax2.T.flat[ ri ].reshape( b.shape )
40
41   # Calculate the residual Ax-b and its 2-norm squared
42   Axb = Ax - b
```

```
43    fx = np.sum( np.power( Axb, 2 ) )
44
45    # Project residual vector (k x 1) onto blank image (ny x nx)
46    Axb2 = np.zeros( x2.shape )
47    Axb2.T.flat[ ri ] = Axb # Fill columns first
48
49    # A'(Ax-b) is just the 2D dct of Axb2
50    AtAxb2 = 2 * dct2( Axb2 )
51    AtAxb = AtAxb2.T.reshape( x.shape ) # Stack columns
52
53    # Copy over the gradient vector
54    np.copyto( g, AtAxb )
55
56    return fx
57
58
59  def progress( x, g, fx, xnrom, gnorm, step, k, ls ):
60    print( f'Iteration {k}', end = '\r' )
61    return 0
62
63
64  # Compute a percentage of the total number of pixels and then
65  # generating a randomly permuted masked array of zeros and ones.
66  def generate_random_samples( total_samples, sample_percentage ):
67    k = round( total_samples * sample_percentage )
68    ri = np.random.choice( total_samples, k, replace = False )
69
70    return ri
71
72
73  def block_indexes( width, height, rows, cols ):
74    start_row = ( height // 2 ) - ( rows // 2 )
75    end_row   = start_row + rows
76
77    start_col = ( width // 2 ) - ( cols // 2 )
78    end_col   = start_col + cols
79
80    ri = []
81    for i in range( 1, cols + 1 ):
82      ri = np.hstack( [ ri, np.arange( ny * ( start_col - 1 + i ) +
       start_row, ny * ( start_col - 1 + i ) + start_row + rows + 1
      ) ] )
83    ri = ri.astype( 'int' )
84    ri = np.setdiff1d( np.array( np.arange( 0, width * height ) ),
      ri )
85
```

```
86    return ri
87
88
89  def create_mask( image, ri ):
90    # Extract a small sample of the signal.
91    b = image.T.flat[ ri ]
92
93    # Create a blank white image and add in the randomly sampled
       pixels.
94    Xm = 255 * np.ones( image.shape )
95    Xm.T.flat[ ri ] = image.T.flat[ ri ]
96
97    # Return the random indices, subsampled image, and the mask for
        visualization
98    return b, Xm
99
100
101 def owl_qn_cs( image, ri, eval_callback, progress_callback ):
102   # Get the dimensions of the image.
103   ny, nx = image.shape
104
105   # Take random samples of the image
106   b = image.T.flat[ ri ].astype( float )
107
108   # Perform the L1 minimization in memory.
109   # Result is in the frequency domain.
110   Xat2 = owlqn( nx * ny, eval_callback, progress_callback, 5 )
111   print( '' )
112
113   # Transform the output back into the spatial domain.
114   Xat = Xat2.reshape( nx, ny ).T # Stack columns.
115   Xa = idct2( Xat ).astype( 'uint8' )
116
117   # Return the reconstructed signal.
118   return Xa
119
120
121 if __name__ == '__main__':
122   # Define image options and select one.
123   image_paths = [ 'img/lena.png', 'img/mountain.jpeg', 'img/
       mandrill.png', 'img/neuschwanstein.jpg', '../van_gogh.jpg' ]
124   save_folders = [ 'lena/', 'mountain/', 'mandrill/', '
       neuschwanstein/', 'van_gogh/' ]
125   image_selection = 0
126   image_path = image_paths[ image_selection ]
127   save_folder = save_folders[ image_selection ]
```

```
128
129   # Define global parameters.
130   sample_percentages = [ 1.00, 0.90, 0.80, 0.70, 0.60, 0.50,
       0.40, 0.30, 0.20, 0.10, 0.01 ] # Percentage of pixel samples
       to keep [0.0, 1.0].
131   image_reduction = 1.00      # Ratio of new image size to
       original [0.0, 1.0].
132
133   # Define vectors to hold metric results.
134   ssim_results = np.zeros( len( sample_percentages ) )
135   mse_results  = np.zeros( len( sample_percentages ) )
136   psnr_results = np.zeros( len( sample_percentages ) )
137
138   # Read in image, resize, and calculate new size.
139   original_image = imageio.imread( image_path, as_gray=False )
140   zoomed_image = original_image
141   # zoomed_image = spimg.zoom( original_image, image_reduction )
142   ny, nx, n_channels = zoomed_image.shape
143
144
145   final_result = np.zeros( zoomed_image.shape, dtype = 'uint8' )
146   masks = np.zeros( zoomed_image.shape, dtype = 'uint8' )
147
148   # Iterate through each sample percentage value.
149   for i, sample_percentage in enumerate( sample_percentages ):
150     print( f'Samples = {100 * sample_percentage}%' )
151     start = time()
152
153     # Get random sample indices so they're the same for all
       channels
154     ri = generate_random_samples( nx * ny, sample_percentage )
155
156     # Iterate through each color channel
157     for j in range( n_channels ):
158       # Randomly sample from the image with the given percentage.
159       # Retrieve the samples (b) and the masked image.
160       b, masks[ :, :, j ] = create_mask( zoomed_image[ :, :, j ],
       ri )
161
162       # Compute results using OWL-QN
163       final_result[ :, :, j ] = owl_qn_cs( zoomed_image[ :, :, j
       ], ri, evaluate, progress )
164
165
166     # Compute Structural Similarity Index (SSIM) of
167     # reconstructed image versus original image.
```

```
168     ssim_results[ i ] = structural_similarity( zoomed_image,
        final_result, data_range = final_result.max() - final_result.
        min(), multichannel = True )
169     mse_results[ i ]  = mean_squared_error( zoomed_image,
        final_result )
170     psnr_results[ i ] = peak_signal_noise_ratio( zoomed_image,
        final_result, data_range = final_result.max() - final_result.
        min() )
171     # print( f'{ssim = }\n{mse = }\n{psnr = }' )
172
173
174     # Save images.
175     imageio.imwrite( f'results/{save_folder}mask_{trunc( 100 *
        sample_percentage )}.png', masks )
176     imageio.imwrite( f'results/{save_folder}recover_{trunc( 100 *
        sample_percentage )}.png', final_result )
177
178     print( f'Elapsed Time: {time() - start:.3f} seconds.\n' )
179
180   for i, sample_percentage in enumerate( sample_percentages ):
181     print( f'{trunc( 100 * sample_percentage ): 6.2f}%:\n    SSIM
        : {ssim_results[ i ]}\n    MSE:  {mse_results[ i ]}\n    PSNR:
        {psnr_results[ i ]}\n' )
```

cs.py