

E1 251 Course Project

Reconstruction from Non-Uniform

Samples Using a DCT- l_p Prior

Dwaipayan Haldar (SR. No.: 27128)

1 Derivation of MM-CG Algorithm

$$J(x) = \|Wx - m\|_2^2 + \lambda \sum_{i=1}^N (\varepsilon + (\text{DCT } x)_i^2)^p$$

Focusing our attention the second term, we can see that $\lambda \sum_{i=1}^N (\varepsilon + (\text{DCT } x)_i^2)^p$ is concave for the given range $0.2 < p < 0.5$. So the cost function can be re-written as

$$J(\mathbf{x}) = f_0(\mathbf{x}) + f_{ccv}(\mathbf{x})$$

where $f_0(x)$ is the convex term $\|W\mathbf{x} - m\|_2^2$ and $f_{ccv}(\mathbf{x}^{(k)})$ is $\lambda \sum_{i=1}^N (\varepsilon + (\text{DCT } \mathbf{x})_i^2)^p$. For the concave term, the linear approximation at $\mathbf{x}^{(k)}$ maximizes the function around $\mathbf{x}^{(k)}$. We can thus write

$$f_{ccv}(\mathbf{x}) \leq f_{ccv}(\mathbf{x}^{(k)}) + \nabla f_{ccv}(\mathbf{x}^{(k)})^T (\mathbf{x} - \mathbf{x}^{(k)})$$

So, we can write

$$J(\mathbf{x}) \leq f_0(\mathbf{x}) + \nabla f_{ccv}(\mathbf{x}^{(k)})^T \mathbf{x} + \text{constant}$$

Let $z_i = (\text{DCT } \mathbf{x})_i^2$ and $g(\mathbf{z}) = \sum_{i=1}^N (\varepsilon + z_i)^p$, where $\mathbf{z} = [z_1 z_2 \dots z_N]^T$. Then $f_{ccv}(\mathbf{x}) = \lambda \mathbf{z}$. Using the above equations we can write,

$$\begin{aligned}
 g(\mathbf{z}) &\leq \nabla g(\mathbf{z}^{(k)})^T \mathbf{z} + \text{constant} \\
 \frac{\partial g(\mathbf{z}^{(k)})}{\partial z_i^{(k)}} &= p(\varepsilon + z_i^{(k)})^{p-1} = p(\varepsilon + (\text{DCT } \mathbf{x}^{(k)})_i^2)^{p-1} =: w_i^{(k)} \text{ (By Definition)} \\
 \implies \nabla g(\mathbf{z}^{(k)}) &= [w_1^{(k)} w_2^{(k)} \dots w_N^{(k)}]^T \implies g(\mathbf{z}) \leq [w_1^{(k)} w_2^{(k)} \dots w_N^{(k)}]^T \odot \mathbf{z} + \text{const} \\
 \implies g(\mathbf{z}) &\leq [w_1^{(k)} w_2^{(k)} \dots w_N^{(k)}]^T \odot (\text{DCT } \mathbf{x})^2 + \text{const} = \sum_{i=1}^N w_i^{(k)} (\text{DCT } \mathbf{x})_i^2 + \text{const}
 \end{aligned}$$

So, the quadratic surrogate of $J(\mathbf{x})$ is:

$$J(\mathbf{x}) \leq \|W\mathbf{x} - m\|_2^2 + \lambda \sum_{i=1}^N w_i^{(k)} (DCT\mathbf{x})_i^2 + \text{Constant}$$

$DCT\mathbf{x}$ can be replaced by $C\mathbf{x}$ as it is a linear transform of \mathbf{x} . And $IDCT\mathbf{x}$ can be replaced by $C^T\mathbf{x}$. Let $W_k = \text{diag}(w_1^{(k)} w_2^{(k)} \dots w_N^{(k)})$. Doing these replacements, the equation can be rewritten as:

$$J(\mathbf{x}) \leq \|W\mathbf{x} - m\|_2^2 + \lambda(C\mathbf{x})^T W_k (C\mathbf{x}) + \text{Constant} = Q(\mathbf{x}) \text{(let)}$$

Making the gradient equals to 0 to get the minimum value.

$$\nabla Q(\mathbf{x}) = (2W^T W\mathbf{x} - 2W^T \mathbf{m}) + (2\lambda C^T W_k C\mathbf{x}) = \mathbf{0}$$

The equation can be rewritten as

$$(W^T W + \lambda C^T W_k C)(\mathbf{x}) = W^T \mathbf{m}$$

which is basically the equation as:

$$(W^T W + \lambda \text{IDCT}(\text{diag}(w^{(k)}) \text{ DCT}))\mathbf{x} = W^T \mathbf{m}$$

■

2 Description of Experimental setup

2.1 Model Images

Images used for the project are `cameraman.tif` and `lena`. Both have a resolution of 256×256 and normalized to $[0, 1]$ using `img_as_float` function of `skimage` package.

2.2 Sampling and Noise

I have used binary random masks using the `numpy.random` with $r \in \{0.2, 0.3, 0.5\}$, where r is the fraction of pixels used.

Noise $\eta_i \sim \mathcal{N}(0, \sigma^2)$, where σ corresponds to an SNR of 30dB. A closed form value of σ can be derived as follows:

$$\text{SNR} = 20 \log_{10} \frac{\|Wx^*\|_2}{\|\eta\|_2} = 30 \implies \frac{\|Wx^*\|_2}{\|\eta\|_2} = 10^{1.5} \implies \|\eta\|_2^2 = \frac{\|Wx^*\|_2^2}{10^3}$$

$\because \mathbb{E}(\eta_i^2) = \sigma^2$ (Normal Distribution with 0 Mean)

$$\implies \|\eta\|_2^2 = N\sigma^2 \implies \sigma^2 = \frac{\|Wx^*\|_2^2}{M \cdot 10^3} \implies \sigma = \sqrt{\frac{\|Wx^*\|_2^2}{M \cdot 10^3}}$$

This is the formula used to calculate the random noise in A.1.1.

2.3 Choice of parameters, Evaluation and Experiments

Experiments are run for each choice of (r, p) . Results are shown in 3 for best λ in case of each (r, p) . Best λ is chosen on the basis of the highest PSNR value. Table.1 gives the concise report for both the images.

3 Results

3.1 Cameraman

Fig.1 gives the original image, sampled noisy image and the Reconstructed image for sampling percentage 0.2. Similarly, Fig.2 and Fig.3 gives the same results for sampling percentage 0.3 and 0.5 respectively. Fig.4 gives the PSNR vs λ plot for $r = 0.5$ and $p \in \{0.3, 0.4, 0.5\}$. Fig.5 gives the Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step plots for $r = 0.5$, $p = 0.5$ and $\lambda = 0.01$ which is the best λ for the given configuration. Qualitatively, $p = 0.5$ outperforms $p = 0.4, 0.3$ which would be evident from Table.1, also for $r = 0.5$ we get better results than $r=0.2, 0.3$ which is quite expected.

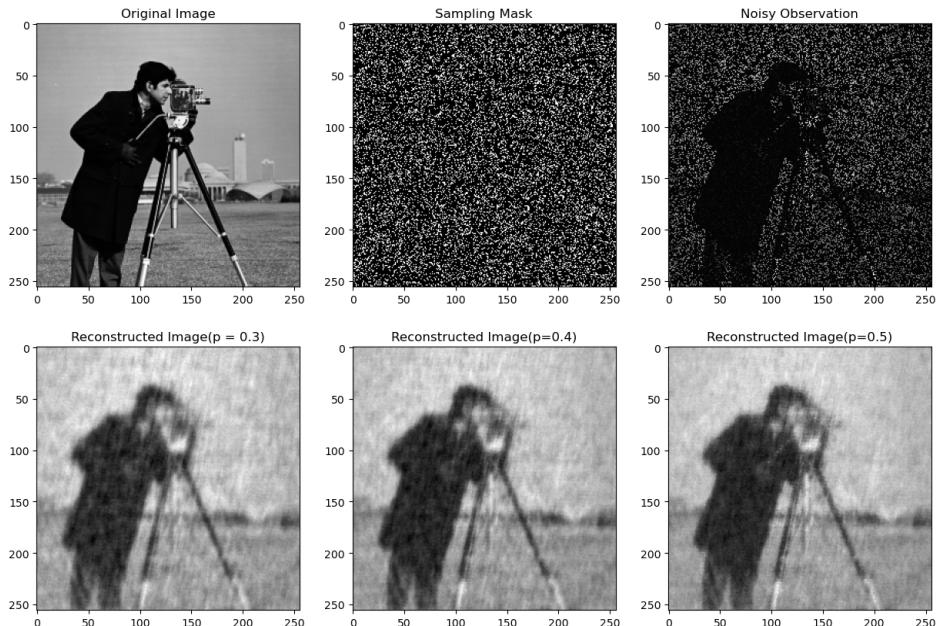


Figure 1: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $r = 0.2$

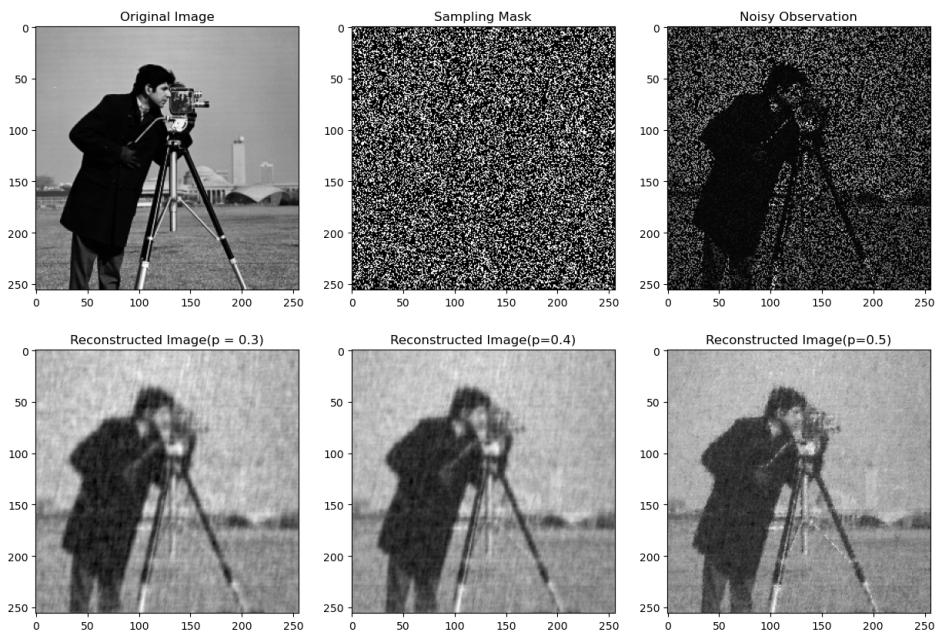


Figure 2: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $\mathbf{r} = \mathbf{0.3}$

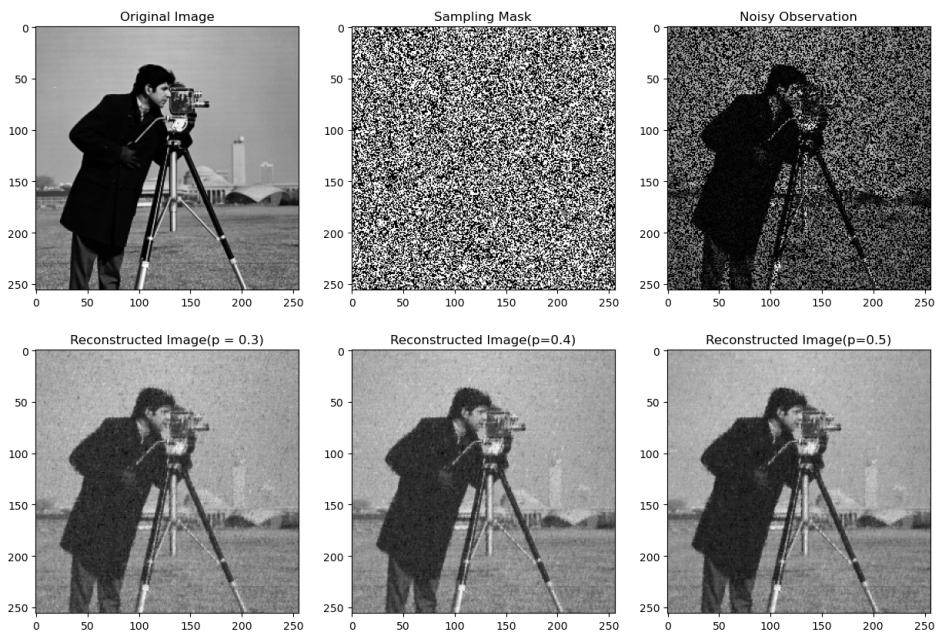


Figure 3: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $\mathbf{r} = \mathbf{0.5}$

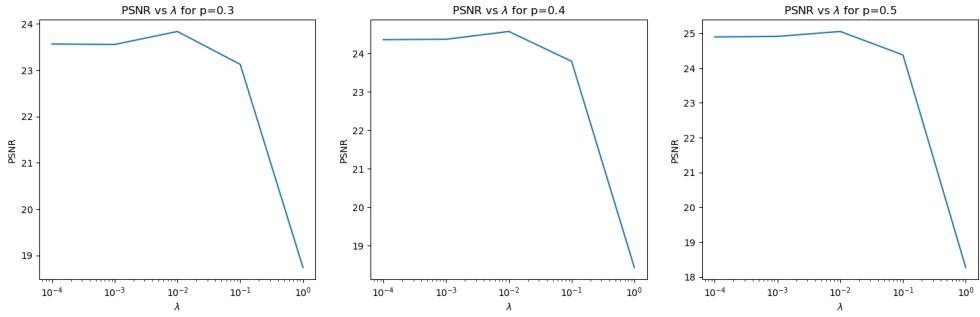


Figure 4: PSNR vs λ for 3 combination of (r,p) where $r = 0.5$ and $p \in \{0.3, 0.4, 0.5\}$

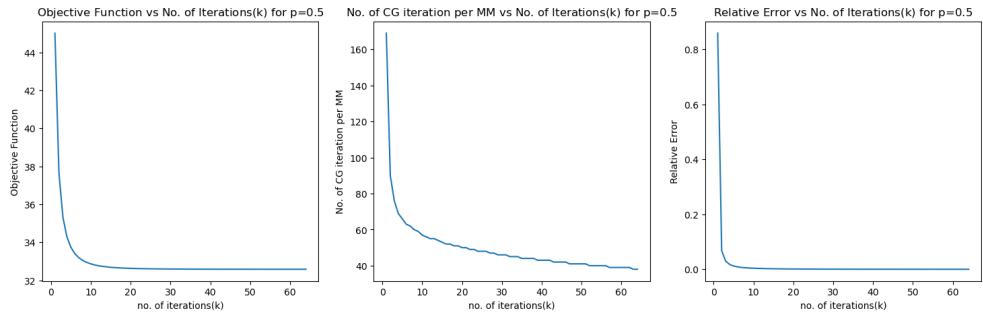


Figure 5: (From left) Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step for $(r,p,\lambda(r,p))=(0.5, 0.5, 0.01)$

3.2 Lena

Fig.6 gives the original image, sampled noisy image and the Reconstructed image for sampling percentage 0.2. Similarly, Fig.7 and Fig.8 gives the same results for sampling percentage 0.3 and 0.5 respectively. Fig.9 gives the PSNR vs λ plot for $r = 0.5$ and $p \in \{0.3, 0.4, 0.5\}$. Fig.10 gives the Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step plots for $r = 0.5$, $p = 0.5$ and $\lambda = 0.01$ which is the best λ for the given configuration. Similar results are seen for lena image also. Qualitatively, $p = 0.5$ outperforms $p = 0.4, 0.3$ which is also supported from Table.1, also for $r = 0.5$ we get better results than $r=0.2, 0.3$ which is quite expected.

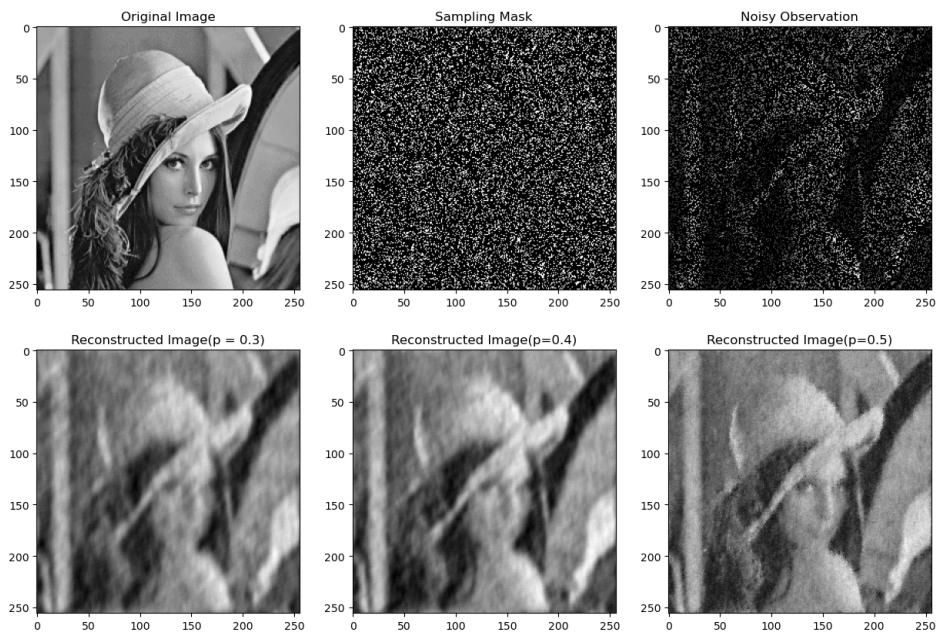


Figure 6: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $\mathbf{r} = \mathbf{0.2}$

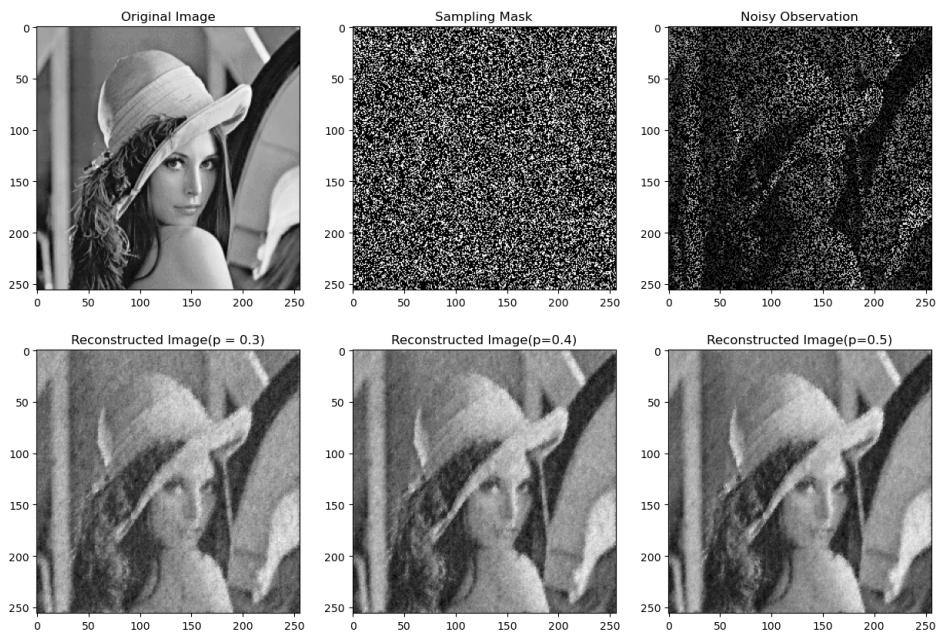


Figure 7: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $\mathbf{r} = \mathbf{0.3}$

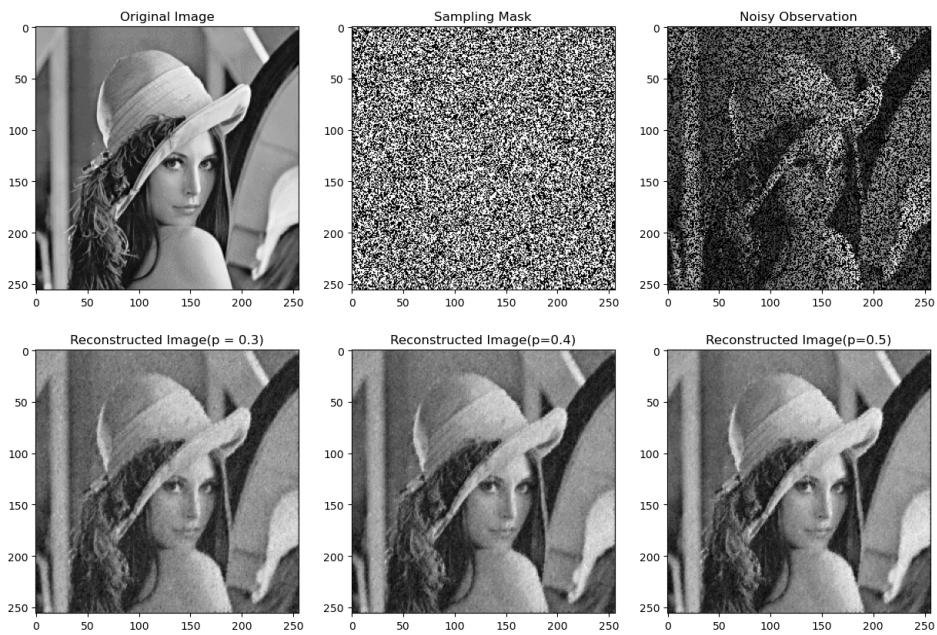


Figure 8: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image($p = 0.3, 0.4, 0.5$) for $\mathbf{r} = 0.5$

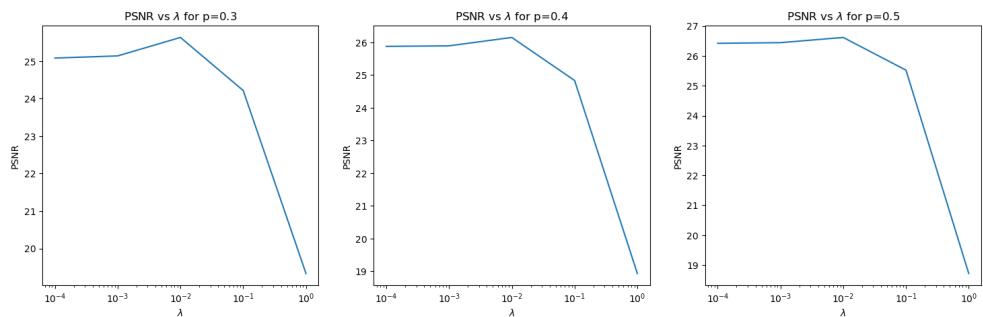


Figure 9: PSNR vs λ for 3 combination of (r, p) where $r = 0.5$ and $p \in \{0.3, 0.4, 0.5\}$

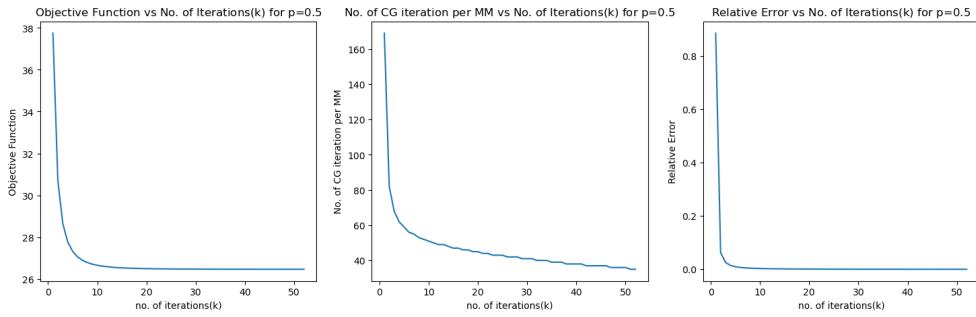


Figure 10: (From left) Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step for $(r,p,\lambda(r,p))=(0.5, 0.5, 0.01)$

3.3 Combined Results

Table 1: Comparison of (r,p) values for both the images

Image	r	p	$\lambda(r,p)$	PSNR	Relative Error	Runtime(s)
cameraman	0.2	0.3	0.1	20.25	0.183	17.67
		0.4	0.1	20.61	0.175	16.11
		0.5	0.1	20.72	0.173	7.44
	0.3	0.3	0.1	21.39	0.160	18.33
		0.4	0.1	21.90	0.151	16.79
		0.5	0.01	22.19	0.146	6.80
	0.5	0.3	0.01	23.83	0.121	42.17
		0.4	0.01	24.46	0.111	24.46
		0.5	0.01	25.05	0.105	4.59
lena	0.2	0.3	0.1	21.21	0.151	13.51
		0.4	0.1	21.79	0.142	15.58
		0.5	0.01	22.10	0.136	8.02
	0.3	0.3	0.01	22.71	0.127	32.31
		0.4	0.01	23.39	0.118	29.49
		0.5	0.01	23.88	0.111	5.16
	0.5	0.3	0.01	25.63	0.0912	39.47
		0.4	0.01	26.15	0.0859	18.30
		0.5	0.01	26.61	0.081	3.82

4 Discussion

4.1 Effect of λ

λ is the weight given to the sparsity term. Here λ is varied over the log scale $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$. Almost in all the cases, $\lambda = 0.1$ or 0.01 is the winner. That is expected since $\lambda = 10^{-4}$ means too less weightage to the sparsity term. So, the images are likely to deviate from natural images. Because, natural images are sparse in the DCT domain. $\lambda = 1$ is giving too much weightage to the sparsity term. So, the image is likely to be oversmoothed. So, a moderate value of λ is expected to give a balanced approach.

4.2 Effect of p

p is another parameter that controls the sparsity. As per Table.1, it is quite evident that $p = 0.5$ comes to be the winner in all the cases. This is because $p = 0.3$ is an over aggressive norm for the sparsity term. Whereas $p = 0.5$ gives the most effective balance for the sparsity prior.

4.3 Effect of r

Understanding the effect of r is pretty trivial. As per Table.1, with increase in r there is a consistent increase in PSNR, consistent decrease in relative error. r is the fraction of samples selected. It is evident that if more samples are selected the final results will come out to better than less no. of samples selected.

4.4 Runtime

The runtime is least for $p = 0.5$, for each r . Further, it is least for $r = 0.5$ for both `cameraman` and `lena`. The runtime would be least where convergence is fastest for more samples and for a better model, it should be the least. It reasserts the fact that $p = 0.5$ gives the best model and balance between the sparsity and reconstruction and does not oversmooths the image.

The next section provides the commented code for all the functions used. All the output images are attached in the zip file. It contains all the plots for all possible combinations. `src.py` contains the source code for all the functions. `wrapper.ipynb` contains the outputs of those functions.

A Code

A.1 Main Functions

This section contains all the main functions used for the project namely the random sampling function, the MM inner loop, the MM-CG complete loop.

A.1.1 Sampling Function

```
1 def sampling(img, r):
2     """
3         Argument: img and the random binary masks
4         Output: Sampled Noisy Vector and the matrix W(index)
5     """
6     #Flatten Image to a vector
7     img_vector = img.flatten()
8     N = img.shape[0]*img.shape[1]
9     #Calculate the value of M, where M = r.N
10    M = int(np.round(r*N))
11
12    idx = np.random.choice(N, size=M, replace=False) #Random Mask
13    Wx_ = img_vector[idx] #Does the operation Wx
14
15    #Adds and returns the added noisy random sampled vector
16    Wx_12_2 = (np.linalg.norm(Wx_))**2
17    sigma = np.sqrt(Wx_12_2/(1000*M))
18    noise = np.random.normal(0, sigma, (M,))
19    return (Wx_ + noise), idx
```

A.1.2 CG Inner loop

```
1 def conjugate_gradient(Q_operator, b, x_0):
2     """
3         Argument: Linear operator Q, RHS vector b, initial guess x_0
4         Output: Solution x and CG iteration count
5     """
6     g_0 = Q_operator(x_0) - b #Initial gradient
7     d_0 = -g_0 #Initial descent direction
8     iteration = 0
9
10    while True:
11        Q_operator_d0 = Q_operator(d_0)
12        den = d_0.T @ Q_operator_d0 #Denominator for step size
13        alpha_k = -(g_0.T @ d_0) / den #Step size
14        x_0 = x_0 + alpha_k*d_0 #Update solution
15
```

```

16     g_0 = Q_operator(x_0) - b #Update gradient
17     #Direction update factor
18     beta_k = (g_0.T @ Q_operator_d0) / den
19     d_0 = -g_0 + beta_k*d_0 #Update search direction
20
21     iteration += 1
22     if np.linalg.norm(g_0) < 1e-6:
23         break
24
25     return x_0, iteration

```

A.1.3 MM CG Outer loop

```

1 def mm_cg(N, m, idx, lammbda, p, epsilon = 1e-6):
2 """
3     Argument: Problem size, sampled image, sampled indices,
4     regularization params
5     Output: Reconstructed signal and convergence metrics
6 """
7
8     #Initial back-projection estimate
9     x_0 = Wtranspose(N, m, idx)
10    x_k = x_0
11
12    #Sampling mask
13    mask = np.zeros((N,))
14    mask[idx] = 1
15
16    list_iteration_cg = [] #Store CG iterations
17    list_objective_function = [] #Store objective values
18    list_relative_error = [] #Store relative errors
19
20    while True:
21        y_k = dct(x_k, norm="ortho")#DCT of current estimate
22        w_k = p*((epsilon + y_k**2)**(p-1))#MM weights
23
24        #Defines operator as given in the problem statement
25        def Q_operator(z):
26            dct_z = dct(z, norm="ortho")
27            return mask*z + lammbda*idct(w_k*dct_z, norm="ortho")
28
29        x_cg, iteration_cg = conjugate_gradient(Q_operator, x_0,
30 x_k)#CG update
31        relative_error = relative_change(x_cg, x_k)
32
33        list_objective_function.append(objective_function(idx,
34 x_cg, m, lammbda, p))
35        list_iteration_cg.append(iteration_cg)
36        list_relative_error.append(relative_error)

```

```

34     if relative_error < 1e-4:#Stopping condition
35         break
36     else:
37         x_k = x_cg#Update iterate
38         continue
39
40     return x_cg, list_objective_function, list_iteration_cg,
41     list_relative_error

```

A.2 Auxiliary Functions

A.2.1 Plotting Function

This is the outermost function, when we call with image and r value, gives all the plots and all the values like PSNR, Relative Error etc. It is the function that is called for output in jupyter notebook.

```

1 def reconstruct(img, r):
2     N = img.shape[0]*img.shape[1] #Total vector size
3     m, idx = sampling(img, r) #Vector m
4     sampling_mask_image = sampling_mask(N, idx).reshape((img.shape
5     [0],img.shape[1])) #Sampling Mask to print
6     noisy_image = Wtranspose(N,m,idx).reshape((img.shape[0],img.
7     shape[1])) #Noisy Image to print
8
9     reconstructed_image_p = []
10    list_of_p = []
11    list_iter_p = []
12    list_re_p = []
13    psnr_list_p = []
14    start_time_p = time.time()
15    for p in [0.3,0.4,0.5]:
16        reconstructed_image_l = []
17        runtime_list = []
18        psnr_list = []
19        list_of_l = []
20        list_iter_l = []
21        list_re_l = []
22        for l in np.logspace(-4,0,num=5):
23            start_time = time.time()
24            reconstructed_image, list_of, list_iter, list_re =
25            mm_cg(N,m,idx,l,p) #MM CG Call for one combination of r,p,
26            lambda
27            reconstructed_image = reconstructed_image.reshape
28            ((256,256))
29            end_time = time.time()
30            psnr_list.append(psnr(reconstructed_image, img))
31            runtime_list.append((end_time-start_time))

```

```

27     reconstructed_image_l.append(reconstructed_image)
28     list_of_l.append(list_of)
29     list_iter_l.append(list_iter)
30     list_re_l.append(list_re)
31     print("One lambda completed...")

32
33     index = np.argmax(psnr_list)
34     lammbda = np.logspace(-4,0,num=5)[index]
35     print(f"Best lambda for p = {p} is {lammbda}")
36     print(f"PSNR for that best lambda = {lammbda} and p = {p} is {psnr_list[index]}")
37     print(f"Relative Change for that best lambda = {lammbda} and p = {p} is {relative_change(reconstructed_image_l[index], img)}")
38     print(f"Runtime for that best lambda = {lammbda} and p = {p} is {runtime_list[index]}")
39     reconstructed_image_p.append(reconstructed_image_l[index])
40     list_of_p.append(list_of_l[index])
41     list_iter_p.append(list_iter_l[index])
42     list_re_p.append(list_re_l[index])
43     psnr_list_p.append(psnr_list)

44
45     end_time_p = time.time()
46     print("Total runtime for one r:", (end_time_p-start_time_p))

47
48     plt.figure(figsize=(15,10))
49     plt.subplot(2,3,1)
50     plt.title("Original Image")
51     plt.imshow(img, cmap= "gray")
52     plt.subplot(2,3,2)
53     plt.title("Sampling Mask")
54     plt.imshow(sampling_mask_image, cmap= "gray")
55     plt.subplot(2,3,3)
56     plt.title("Noisy Observation")
57     plt.imshow(noisy_image, cmap= "gray")
58     plt.subplot(2,3,4)
59     plt.title("Reconstructed Image(p = 0.3)")
60     plt.imshow(reconstructed_image_p[0], cmap= "gray")
61     plt.subplot(2,3,5)
62     plt.title("Reconstructed Image(p=0.4)")
63     plt.imshow(reconstructed_image_p[1], cmap= "gray")
64     plt.subplot(2,3,6)
65     plt.title("Reconstructed Image(p=0.5)")
66     plt.imshow(reconstructed_image_p[2], cmap= "gray")
67     plt.show()

68
69     plt.figure(figsize=(18,5))
70     plt.subplot(1,3,1); plt.title(r"PSNR vs $\lambda$ for p=$0.3$")
      ;plt.semilogx(np.logspace(-4,0,num=5), psnr_list_p[0]);plt.

```

```

71     xlabel(r"$\lambda$"); plt.ylabel("PSNR")
72     plt.subplot(1,3,2); plt.title(r"PSNR vs $\lambda$ for p=$0.4$")
73     plt.semilogx(np.logspace(-4,0,num=5), psnr_list_p[1]); plt.
74     xlabel(r"$\lambda$"); plt.ylabel("PSNR")
75     plt.subplot(1,3,3); plt.title(r"PSNR vs $\lambda$ for p=$0.5$")
76     plt.semilogx(np.logspace(-4,0,num=5), psnr_list_p[2]); plt.
77     xlabel(r"$\lambda$"); plt.ylabel("PSNR")
78     plt.show()
79
80
81     plt.figure(figsize=(18,5))
82     plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
83     Iterations(k) for p=$0.3$"); plt.plot(range(1,len(list_of_p[0])
84     +1), list_of_p[0]); plt.xlabel("no. of iterations(k)"); plt.
85     ylabel("Objective Function")
86
87     plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
88     No. of Iterations(k) for p=$0.3$"); plt.plot(range(1,len(
89     list_iter_p[0])+1), list_iter_p[0]); plt.xlabel("no. of
90     iterations(k)"); plt.ylabel("No. of CG iteration per MM")
91
92     plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of
93     Iterations(k) for p=$0.3$"); plt.plot(range(1,len(list_re_p[0])
94     +1), list_re_p[0]); plt.xlabel("no. of iterations(k)"); plt.
95     ylabel("Relative Error")
96     plt.show()
97
98
99     plt.figure(figsize=(18,5))
100    plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
101    Iterations(k) for p=$0.4$"); plt.plot(range(1,len(list_of_p[1])
102    +1), list_of_p[1]); plt.xlabel("no. of iterations(k)"); plt.
103    ylabel("Objective Function")
104
105    plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
106    No. of Iterations(k) for p=$0.4$"); plt.plot(range(1,len(
107    list_iter_p[1])+1), list_iter_p[1]); plt.xlabel("no. of
108    iterations(k)"); plt.ylabel("No. of CG iteration per MM")
109
110    plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of
111    Iterations(k) for p=$0.4$"); plt.plot(range(1,len(list_re_p[1])
112    +1), list_re_p[1]); plt.xlabel("no. of iterations(k)"); plt.
113    ylabel("Relative Error")
114    plt.show()
115
116
117    plt.figure(figsize=(18,5))
118    plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
119    Iterations(k) for p=$0.5$"); plt.plot(range(1,len(list_of_p[2])
120    +1), list_of_p[2]); plt.xlabel("no. of iterations(k)"); plt.
121    ylabel("Objective Function")
122
123    plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
124    No. of Iterations(k) for p=$0.5$"); plt.plot(range(1,len(
125    list_iter_p[2])+1), list_iter_p[2]); plt.xlabel("no. of
126    iterations(k)"); plt.ylabel("No. of CG iteration per MM")

```

```

90     plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of
91 Iterations(k) for p=$0.5$");plt.plot(range(1,len(list_re_p[2])
92 +1), list_re_p[2]);plt.xlabel("no. of iterations(k)");plt.
93 ylabel("Relative Error")
94     plt.show()
95
96     return reconstructed_image_p

```

A.2.2 Other Functions

This section contains those functions like psnr, sampling mask etc.

```

1 def psnr(img, img_ref):
2     img = img.flatten()
3     img_ref = img_ref.flatten()
4     num = np.linalg.norm(img_ref, ord = np.inf)
5     den = np.linalg.norm(img_ref - img)/np.sqrt(img_ref.shape[0])
6     return 20*np.log10(num/den)
7
8 def relative_change(x,y):
9     return np.linalg.norm(x-y)/np.linalg.norm(y)
10
11 def sampling_mask(N, idx):
12     sampling_mask = np.zeros((N,))
13     sampling_mask[idx] = 1
14     return sampling_mask
15
16 def objective_function(idx, x, m, lammbda, p, epsilon=1e-6):
17     diff = x[idx] - m
18     term1 = np.sum(diff * diff)
19     term2 = lammbda * np.sum(np.abs(epsilon + dct(x, norm = "ortho"
20     )**2)**p)
21     return term1 + term2
22
23 def Wtranspose(N,x, idx):
24     Wtranspose = np.zeros((N,))
25     Wtranspose[idx] = x
26     return Wtranspose

```

After this, all the output is obtained by simply calling the `reconstruct` function with the image and the sampling factor in the `wrapper.ipynb` file and the output images are saved in the respective folder.