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

$$g(\mathbf{z}) \leq \nabla g(\mathbf{z}^{(k)})^T \mathbf{z} + \text{constant}$$

$$\frac{\partial g(\mathbf{z}^{(k)})}{\partial z_i^{(k)}} = p(\varepsilon + z_i^{(k)})^{p-1} = p(\varepsilon + (DCT\mathbf{x}^{(k)})_i^2)^{p-1} =: w_i^{(k)} (\text{By Definition})$$

$$\implies \nabla g(\mathbf{z}^{(k)}) = \left[w_1^{(k)} w_2^{(k)} \cdots w_N^{(k)}\right]^T \implies g(\mathbf{z}) \leq \left[w_1^{(k)} w_2^{(k)} \cdots w_N^{(k)}\right]^T \odot \mathbf{z} + \text{const}$$

$$\implies g(\mathbf{z}) \leq \left[w_1^{(k)} w_2^{(k)} \cdots w_N^{(k)}\right]^T \odot (DCT\mathbf{x})^2 + \text{const} = \sum_{i=i}^{N} w_i^{(k)} (DCT\mathbf{x})_i^2 + \text{const}$$

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 $W_k = diag(w_1^{(k)} w_2^{(k)} \cdots w_N^{(k)})$. 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:

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

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

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

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 choosen 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. Similarily, 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 evident from Table.1, also for r = 0.5 we get better results than r=0.2, 0.3 which is quite expected.

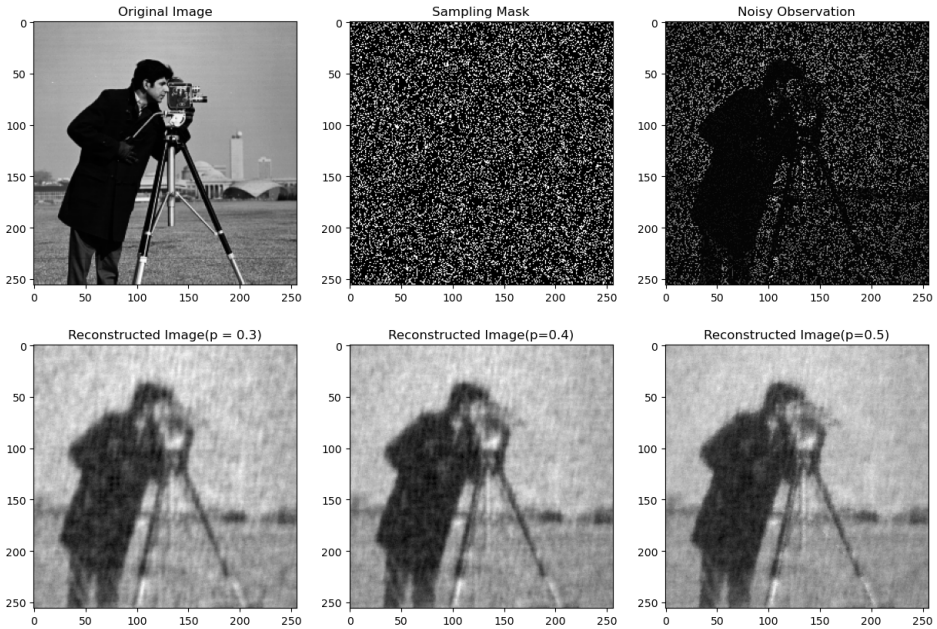

Figure 1: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for **r = 0.2**

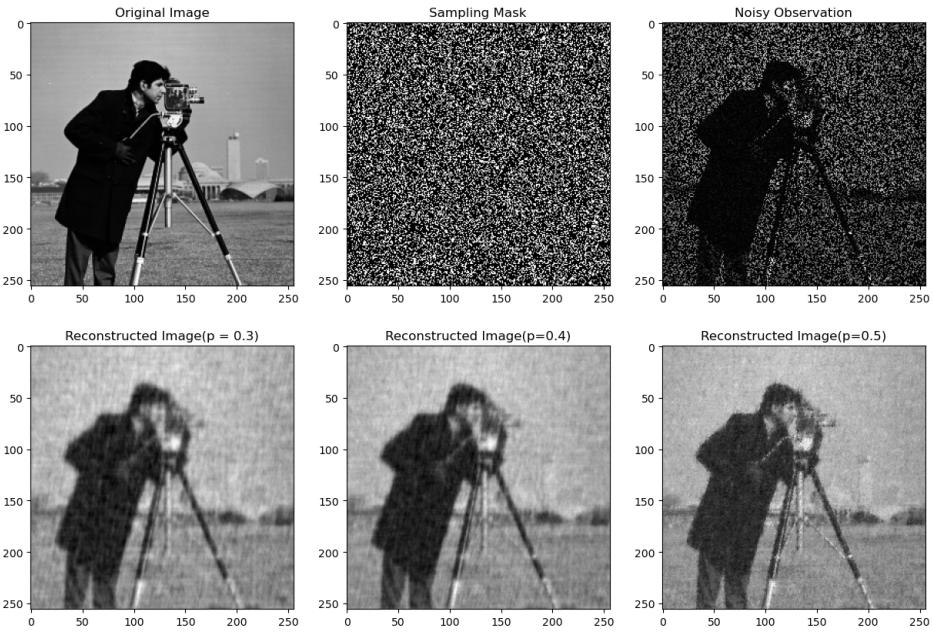

Figure 2: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for $\mathbf{r = 0.3}$

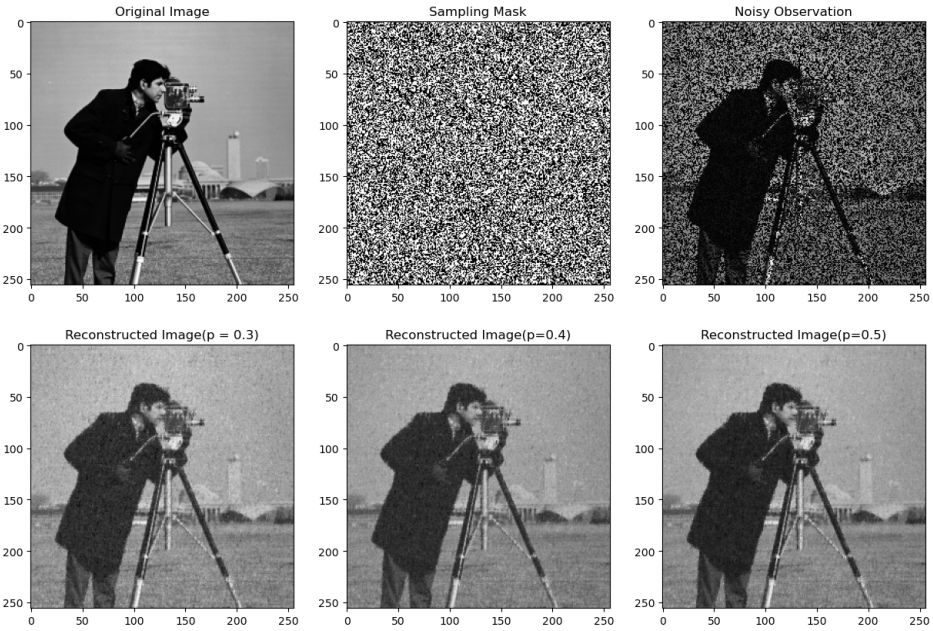

Figure 3: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for $\mathbf{r = 0.5}$

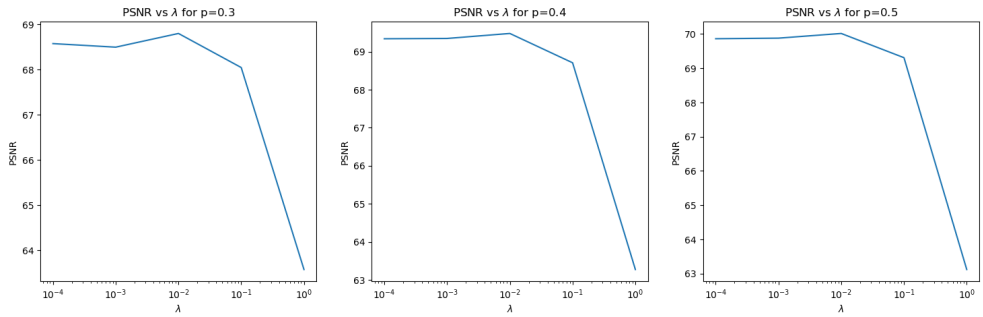

Figure 4: PSNR vs λ for 3 combination of (r,p) where r = 0.5 and p $\in \{0.3, 0.4, 0.5\}$

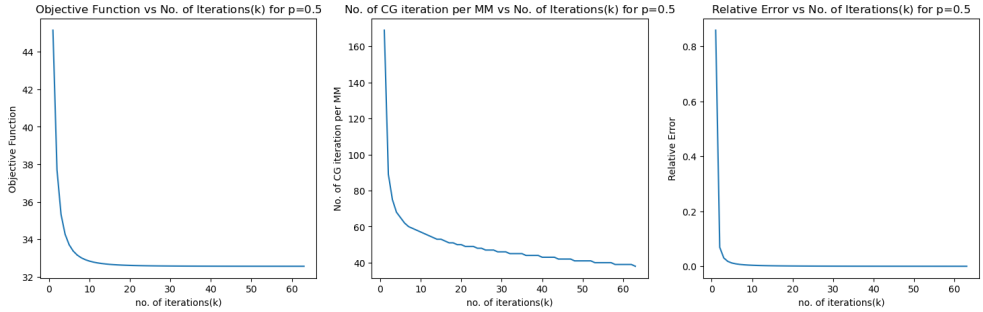

Figure 5: (From left) Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step for (r,p,λ(r,p))=(0.5, 0.5, 0.01)

3.2 Lena

Fig.6 gives the original image, sampled noisy image and the Reconstructed image for sampling percentage 0.2. Similarily, 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.

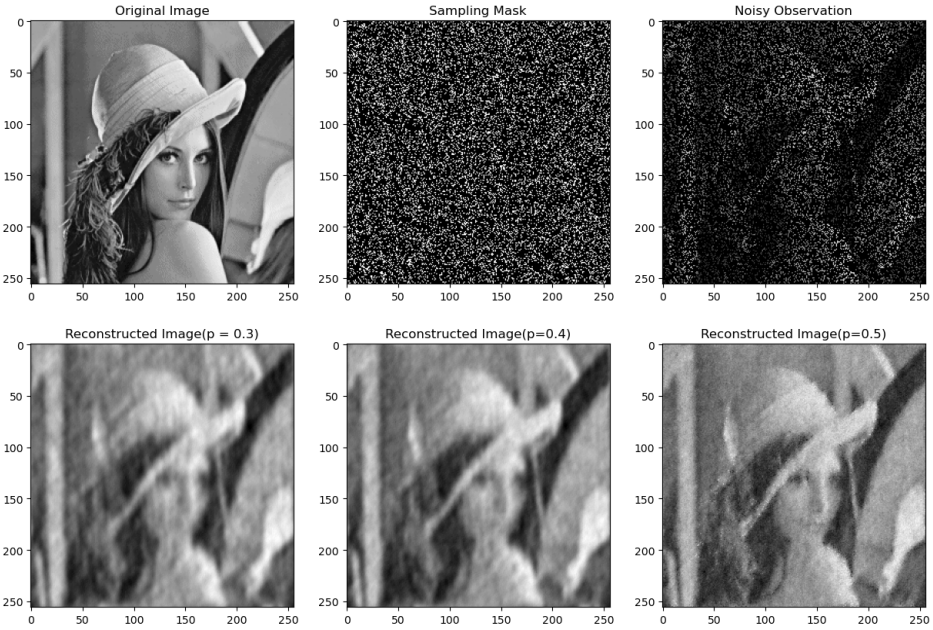

Figure 6: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for $\mathbf{r = 0.2}$

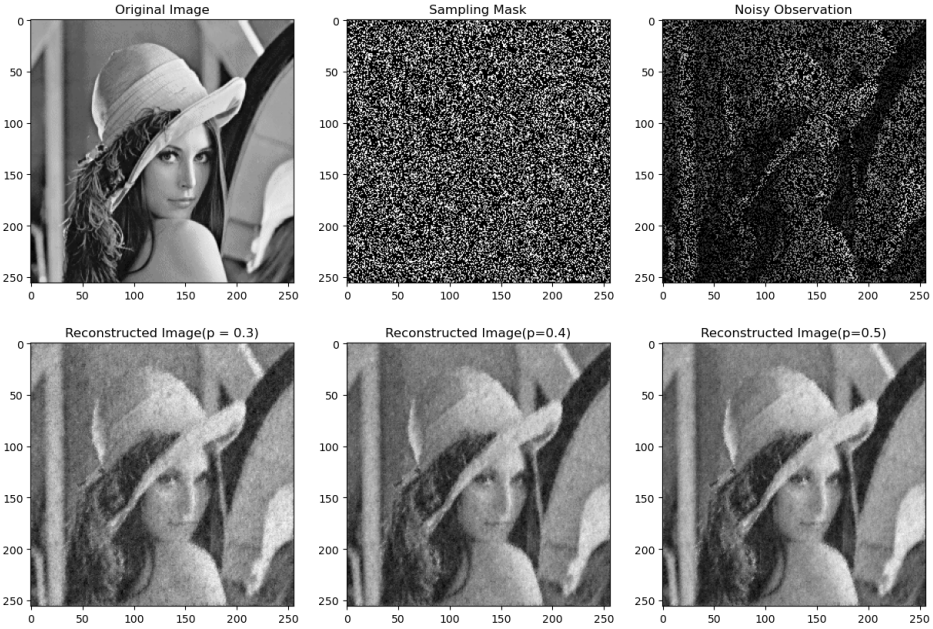

Figure 7: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for $\mathbf{r = 0.3}$

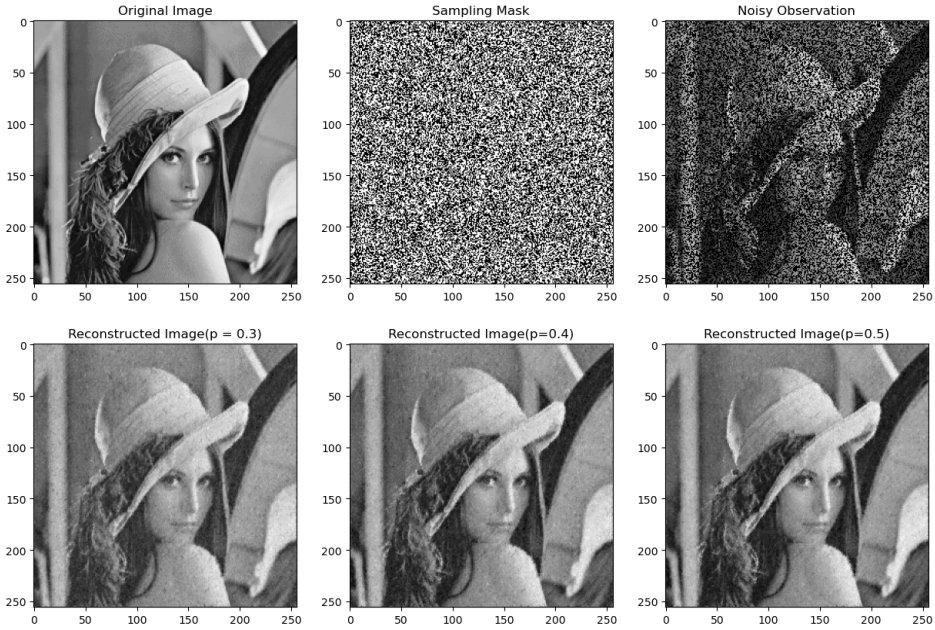

Figure 8: (First Row)Original Image, Sampling Mask, Noisy Masked Image (Second Row)Reconstructed Image(p = 0.3,0.4,0.5) for $\mathbf{r = 0.5}$

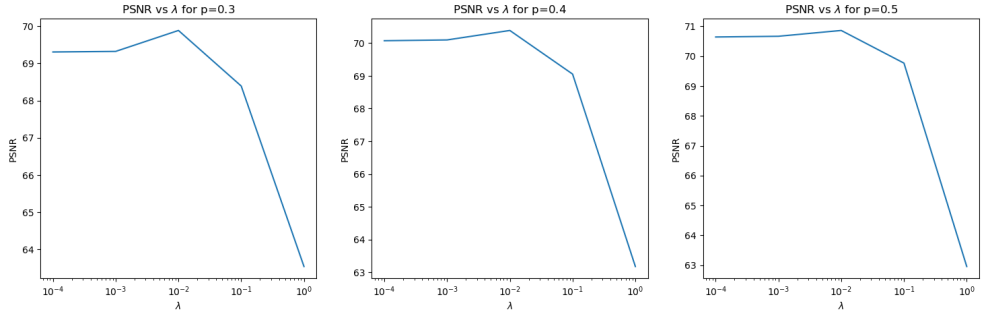

Figure 9: PSNR vs λ for 3 combination of (r,p) where r = 0.5 and p $\in \{0.3, 0.4, 0.5\}$

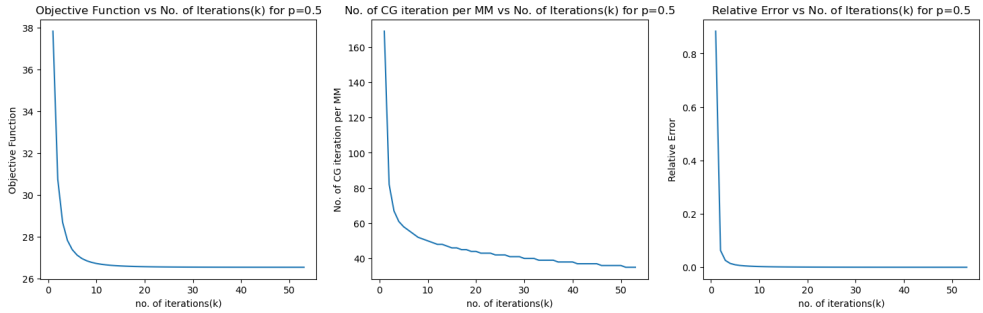

Figure 10: (From left) Objective Function vs iteration step, No. of CG iteration vs iteration step, Relative error vs iteration step for (r,p,λ(r,p))=(0.5, 0.5, 0.01)

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	65.06	0.184	19.06
		0.4	0.1	65.48	0.175	18.86
		0.5	0.1	**65.56**	**0.173**	**7.74**
	0.3	0.3	0.1	66.29	0.159	16.86
		0.4	0.1	66.81	0.150	17.95
		0.5	0.01	**67.07**	**0.140**	**7.34**
	0.5	0.3	0.01	68.79	0.119	42.11
		0.4	0.01	69.47	0.110	32.65
		0.5	0.01	**70.07**	**0.104**	**4.89**
lena	0.2	0.3	0.1	65.53	0.149	14.70
		0.4	0.1	66.06	0.140	15.67
		0.5	0.01	**66.32**	**0.136**	**7.78**
	0.3	0.3	0.01	66.81	0.129	31.68
		0.4	0.01	67.70	0.116	39.45
		0.5	0.01	**68.12**	**0.111**	**5.80**
	0.5	0.3	0.01	69.88	0.0907	26.08
		0.4	0.01	70.38	0.0856	18.01
		0.5	0.01	**70.85**	**0.081**	**3.44**

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 is 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

```
def sampling(img, r):
    """
    Argument: img and the random binary masks
    Output: Sampled Noisy Vector and the matrix W(index)
    """
    #Flatten Image to a vector
    img_vector = img.flatten()
    N = img.shape[0]*img.shape[1]
    #Calculate the value of M, where M = r.N
    M = int(np.round(r*N))

    idx = np.random.choice(N, size=M, replace=False) #Random Mask
    Wx_ = img_vector[idx] #Does the operation Wx

    #Adds and returns the added noisy random sampled vector
    Wx_l2_2 = (np.linalg.norm(Wx_))**2
    sigma = np.sqrt(Wx_l2_2/(1000*M))
    noise = np.random.normal(0, sigma, (M,))
    return (Wx_ + noise), idx
```

A.1.2 CG Inner loop

```
def conjugate_gradient(Q_operator, b, x_0):
    """
    Argument: Linear operator Q, RHS vector b, initial guess x_0
    Output: Solution x and CG iteration count
    """
    g_0 = Q_operator(x_0) - b#Initial gradient
    d_0 = -g_0#Initial descent direction
    iteration = 0

    while True:
        Q_operator_d0 = Q_operator(d_0)
        den = d_0.T @ Q_operator_d0 #Denominator for step size
        alpha_k = -(g_0.T @ d_0) / den #Step size
        x_0 = x_0 + alpha_k*d_0 #Update solution

```

```
        g_0 = Q_operator(x_0) - b #Update gradient
        #Direction update factor
        beta_k = (g_0.T @ Q_operator_d0) / den
        d_0 = -g_0 + beta_k*d_0 #Update search direction

        iteration += 1
        if np.linalg.norm(g_0) < 1e-6:
            break
    return x_0, iteration
```

A.1.3 MM CG Outer loop

```
def mm_cg(N, m, idx, lammbda, p, epsilon = 1e-6):
    """
    Argument: Problem size, sampled image, sampled indices,
   regularization params
    Output: Reconstructed signal and convergence metrics
    """
    #Initial back-projection estimate
    x_0 = Wtranspose(N, m, idx)
    x_k = x_0

    #Sampling mask
    mask = np.zeros((N,))
    mask[idx] = 1

    list_iteration_cg = [] #Store CG iterations
    list_objective_function = [] #Store objective values
    list_relative_error = [] #Store relative errors

    while True:
        y_k = dct(x_k, norm="ortho")#DCT of current estimate
        w_k = p*((epsilon + y_k**2)**(p-1))#MM weights

        #Defines operator as given in the problem statement
        def Q_operator(z):
            dct_z = dct(z, norm="ortho")
            return mask*z + lammbda*idct(w_k*dct_z, norm="ortho")

        x_cg, iteration_cg = conjugate_gradient(Q_operator, x_0,
   x_k)#CG update
        relative_error = relative_change(x_cg, x_k)

        list_objective_function.append(objective_function(idx,
   x_cg, m, lammbda, p))
        list_iteration_cg.append(iteration_cg)
        list_relative_error.append(relative_error)

```

```
        if relative_error < 1e-4:#Stopping condition
            break
        else:
            x_k = x_cg#Update iterate
            continue

    return x_cg, list_objective_function, list_iteration_cg,
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.

```
def reconstruct(img, r):
    N = img.shape[0]*img.shape[1] #Total vector size
    m,idx = sampling(img, r) #Vector m
    sampling_mask_image = sampling_mask(N, idx).reshape((img.shape
[0],img.shape[1])) #Sampling Mask to print
    noisy_image = Wtranspose(N,m,idx).reshape((img.shape[0],img.
shape[1])) #Noisy Image to print

    reconstructed_image_p = []
    list_of_p = []
    list_iter_p = []
    list_re_p = []
    psnr_list_p = []
    start_time_p = time.time()
    for p in [0.3,0.4,0.5]:
        reconstructed_image_l = []
        runtime_list = []
        psnr_list = []
        list_of_l = []
        list_iter_l = []
        list_re_l = []
        for l in np.logspace(-4,0,num=5):
            start_time = time.time()
            reconstructed_image, list_of, list_iter,list_re =
mm_cg(N,m,idx,l,p) #MM CG Call for one combination of r,p,
lambda
            reconstructed_image = reconstructed_image.reshape
((256,256))
            end_time = time.time()
            psnr_list.append(psnr(reconstructed_image, img))
            runtime_list.append((end_time-start_time))
```

```
            reconstructed_image_l.append(reconstructed_image)
            list_of_l.append(list_of)
            list_iter_l.append(list_iter)
            list_re_l.append(list_re)
            print("One lambda completed...")

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

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

    plt.figure(figsize=(15,10))
    plt.subplot(2,3,1)
    plt.title("Original Image")
    plt.imshow(img, cmap= "gray")
    plt.subplot(2,3,2)
    plt.title("Sampling Mask")
    plt.imshow(sampling_mask_image, cmap= "gray")
    plt.subplot(2,3,3)
    plt.title("Noisy Observation")
    plt.imshow(noisy_image, cmap= "gray")
    plt.subplot(2,3,4)
    plt.title("Reconstructed Image(p = 0.3)")
    plt.imshow(reconstructed_image_p[0], cmap= "gray")
    plt.subplot(2,3,5)
    plt.title("Reconstructed Image(p=0.4)")
    plt.imshow(reconstructed_image_p[1], cmap= "gray")
    plt.subplot(2,3,6)
    plt.title("Reconstructed Image(p=0.5)")
    plt.imshow(reconstructed_image_p[2], cmap= "gray")
    plt.show()

    plt.figure(figsize=(18,5))
    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.
```

```
    xlabel(r"$\lambda$");plt.ylabel("PSNR")
71   plt.subplot(1,3,2); plt.title(r"PSNR vs $\lambda$ for p=$0.4$"
    );plt.semilogx(np.logspace(-4,0,num=5), psnr_list_p[1]);plt.
    xlabel(r"$\lambda$");plt.ylabel("PSNR")
72   plt.subplot(1,3,3); plt.title(r"PSNR vs $\lambda$ for p=$0.5$"
    );plt.semilogx(np.logspace(-4,0,num=5), psnr_list_p[2]);plt.
    xlabel(r"$\lambda$");plt.ylabel("PSNR")
73   plt.show()
74
75   plt.figure(figsize=(18,5))
76   plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
    Iterations(k) for p=$0.3$");plt.plot(range(1,len(list_of_p[0])
    +1), list_of_p[0]);plt.xlabel("no. of iterations(k)");plt.
    ylabel("Objective Function")
77   plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
    No. of Iterations(k) for p=$0.3$");plt.plot(range(1,len(
    list_iter_p[0])+1), list_iter_p[0]);plt.xlabel("no. of
    iterations(k)");plt.ylabel("No. of CG iteration per MM")
78   plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of
    Iterations(k) for p=$0.3$");plt.plot(range(1,len(list_re_p[0])
    +1), list_re_p[0]);plt.xlabel("no. of iterations(k)");plt.
    ylabel("Relative Error")
79   plt.show()
80
81   plt.figure(figsize=(18,5))
82   plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
    Iterations(k) for p=$0.4$");plt.plot(range(1,len(list_of_p[1])
    +1), list_of_p[1]);plt.xlabel("no. of iterations(k)");plt.
    ylabel("Objective Function")
83   plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
    No. of Iterations(k) for p=$0.4$");plt.plot(range(1,len(
    list_iter_p[1])+1), list_iter_p[1]);plt.xlabel("no. of
    iterations(k)");plt.ylabel("No. of CG iteration per MM")
84   plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of
    Iterations(k) for p=$0.4$");plt.plot(range(1,len(list_re_p[1])
    +1), list_re_p[1]);plt.xlabel("no. of iterations(k)");plt.
    ylabel("Relative Error")
85   plt.show()
86
87   plt.figure(figsize=(18,5))
88   plt.subplot(1,3,1); plt.title(r"Objective Function vs No. of
    Iterations(k) for p=$0.5$");plt.plot(range(1,len(list_of_p[2])
    +1), list_of_p[2]);plt.xlabel("no. of iterations(k)");plt.
    ylabel("Objective Function")
89   plt.subplot(1,3,2); plt.title(r"No. of CG iteration per MM vs
    No. of Iterations(k) for p=$0.5$");plt.plot(range(1,len(
    list_iter_p[2])+1), list_iter_p[2]);plt.xlabel("no. of
    iterations(k)");plt.ylabel("No. of CG iteration per MM")
```

```
    plt.subplot(1,3,3); plt.title(r"Relative Error vs No. of Iterations(k) for p=$0.5$");plt.plot(range(1,len(list_re_p[2])+1), list_re_p[2]);plt.xlabel("no. of iterations(k)");plt.ylabel("Relative Error")
    plt.show()

    return reconstructed_image_p
```

A.2.2 Other Functions

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

```
def psnr(img, img_ref):
    num = np.linalg.norm(img_ref, ord = np.inf)
    den = np.linalg.norm(img_ref - img)/np.sqrt(img_ref.shape[0]*img_ref.shape[1])
    return 20*np.log10(num/den)

def relative_change(x,y):
    return np.linalg.norm(x-y)/np.linalg.norm(y)

def sampling_mask(N,idx):
    sampling_mask = np.zeros((N,))
    sampling_mask[idx] = 1
    return sampling_mask

def objective_function(idx, x, m, lammbda, p, epsilon=1e-6):
    diff = x[idx] - m
    term1 = np.sum(diff * diff)
    term2 = lammbda * np.sum(np.abs(epsilon + dct(x, norm = "ortho")**2)**p)
    return term1 + term2

def Wtranspose(N,x, idx):
    Wtransposex = np.zeros((N,))
    Wtransposex[idx] = x
    return Wtransposex
```

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.