

Reconstruction from Non-Uniform Samples Using a DCT- p Prior

Vinay
SR NO.: 23653

Abstract

This report describes an optimization-based method to restore grayscale images from sparse, noisy pixel measurements. The approach enforces sparsity in the Discrete Cosine Transform domain via a nonconvex ℓ_p -style penalty with $0 < p < 1$. A Majorization-Minimization (MM) routine converts the problem into a sequence of weighted quadratic subproblems. Each subproblem is solved by Preconditioned Conjugate Gradients (PCG). We present the derivation, implementation details, and numerical results on standard test images across multiple sampling ratios and hyperparameters.

1 Introduction

Reconstructing images from incomplete observations is central to many imaging tasks. When only a subset of pixels are available and measurements are noisy, the inverse problem is ill-posed. A common remedy is to impose a prior. Natural images are approximately sparse in transform domains such as the DCT. We exploit this by penalizing DCT coefficients with an ℓ_p -type penalty for $0 < p < 1$. The overall objective balances data fidelity at sampled pixels with a sparsity-promoting regularizer in the DCT domain.

2 Method and Detailed Derivation

Model. Let $x^* \in \mathbb{R}^N$ be the vectorized ground-truth image and let $W \in \mathbb{R}^{M \times N}$ be the sampling operator that extracts the M observed pixels. The noisy measurements are

$$m = Wx^* + \eta, \quad \eta \sim \mathcal{N}(0, \sigma^2 I). \quad (1)$$

Let $C \in \mathbb{R}^{N \times N}$ denote the orthonormal DCT matrix (“ortho” normalization), so that $y = Cx = \text{DCT}(x)$ and $C^\top C = I$ with $\text{IDCT} = C^\top$. We estimate x by minimizing

$$J(x) = \underbrace{\|Wx - m\|_2^2}_{\text{data term}} + \lambda \sum_{i=1}^N \phi(y_i^2), \quad y = Cx, \quad \phi(t) = (\varepsilon + t)^p, \quad (2)$$

with $0 < p < 1$ and $\varepsilon > 0$ small.

Concavity and a tangent majorizer. For $0 < p < 1$,

$$\phi'(t) = p(\varepsilon + t)^{p-1}, \quad \phi''(t) = p(p-1)(\varepsilon + t)^{p-2} \leq 0 \quad \text{for } t \geq 0,$$

so ϕ is concave on $[0, \infty)$. By concavity, for any $s \geq 0$,

$$\phi(t) \leq \phi(s) + \phi'(s)(t - s) \quad (\text{first-order tangent upper bound}). \quad (3)$$

At MM iteration k , let $y^{(k)} = Cx^{(k)}$ and choose $s_i = (y_i^{(k)})^2$ in (3) with $t = y_i^2$:

$$\phi(y_i^2) \leq \phi((y_i^{(k)})^2) + \underbrace{\phi'((y_i^{(k)})^2)}_{w_i^{(k)}} (y_i^2 - (y_i^{(k)})^2), \quad w_i^{(k)} = p(\varepsilon + (y_i^{(k)})^2)^{p-1} \geq 0. \quad (4)$$

Summing over i and discarding terms independent of x yields the quadratic surrogate

$$Q(x \mid x^{(k)}) = \|Wx - m\|_2^2 + \lambda \sum_{i=1}^N w_i^{(k)} y_i^2 = \|Wx - m\|_2^2 + \lambda x^\top C^\top (\text{diag}(w^{(k)})) C x. \quad (5)$$

By construction, $Q(x^{(k)} \mid x^{(k)}) = J(x^{(k)})$ and $J(x) \leq Q(x \mid x^{(k)}) + \text{const}$, so minimizing Q guarantees descent of J .

Normal equations of the surrogate. Differentiate (5) and set the gradient to zero:

$$\nabla_x Q(x \mid x^{(k)}) = 2W^\top(Wx - m) + 2\lambda C^\top \text{diag}(w^{(k)}) C x = 0, \quad (6)$$

$$\Rightarrow \quad \boxed{\left(W^\top W + \lambda C^\top \text{diag}(w^{(k)}) C \right) x = W^\top m.} \quad (7)$$

Using $\text{IDCT} = C^\top$ and $\text{DCT} = C$,

$$\left(W^\top W + \lambda \text{IDCT} \text{diag}(w^{(k)}) \text{DCT} \right) x = W^\top m. \quad (8)$$

The coefficient matrix is symmetric positive definite (SPD): $W^\top W \succeq 0$, $\text{diag}(w^{(k)}) \succeq 0$, and C is orthonormal, hence the sum is SPD whenever either W samples any pixel or some weight is strictly positive. Therefore PCG is applicable.

Matrix-free application of the system matrix. Let $A_k := W^\top W + \lambda C^\top \text{diag}(w^{(k)}) C$. For any $z \in \mathbb{R}^N$,

$$A_k z = W^\top W z + \lambda C^\top (w^{(k)} \odot (Cz)), \quad (9)$$

where \odot is elementwise multiplication. In image-space code this is

$$z \mapsto \underbrace{W^\top (Wz)}_{\text{mask} \cdot z} + \lambda \text{IDCT} \left(w^{(k)} \odot \text{DCT}(z) \right),$$

which matches the routine `apply_M_operator`.

Jacobi preconditioning. Let $\alpha_k := \max_i w_i^{(k)}$. Since $0 \leq \text{diag}(w^{(k)}) \preceq \alpha_k I$,

$$C^\top \text{diag}(w^{(k)}) C \preceq \alpha_k C^\top I C = \alpha_k I. \quad (10)$$

Thus a cheap diagonal upper bound for A_k is

$$P_k := \underbrace{\text{diag}(W^\top W)}_{\text{mask in } \{0,1\}} + \lambda \alpha_k I, \quad (11)$$

i.e., elementwise $(P_k)_j = (\text{mask})_j + \lambda \alpha_k$. This is exactly the Jacobi preconditioner used in code:

$$r = b - A_k x, \quad z = P_k^{-1} r,$$

with standard PCG recurrences.

MM-PCG algorithm. Given $x^{(0)}$ (e.g., zero-filled or the masked data), iterate for $k = 0, 1, 2, \dots$

1. Compute $y^{(k)} = Cx^{(k)}$ and weights $w_i^{(k)} = p(\varepsilon + (y_i^{(k)})^2)^{p-1}$.
2. Form the linear system (7) implicitly via the operator (9).
3. Solve $A_k x^{(k+1)} = W^\top m$ with PCG, preconditioned by P_k , to tolerance `cg_tol` or a cap of `max_cg_iter`.
4. Stop the outer loop when $\frac{\|x^{(k+1)} - x^{(k)}\|_2}{\|x^{(k)}\|_2} < \text{mm_tol}$ or when k reaches `max_mm_iter`.

Each MM step strictly decreases J unless already at a fixed point of the surrogate. At convergence, the KKT conditions of (2) are satisfied.

Connection to the implementation.

- Weight update: $w^{(k)} \leftarrow p(\varepsilon + (\text{DCT}(x^{(k)}))^2)^{p-1}$.
- System application: $z \mapsto \text{mask} \cdot z + \lambda \text{IDCT}(w^{(k)} \odot \text{DCT}(z))$.
- Preconditioner: $P_k = \text{mask} + \lambda \max_i w_i^{(k)}$ (elementwise), with safeguard $P_{k,j} \leftarrow \max(P_{k,j}, 10^{-6})$.

3 Implementation details

The algorithm was implemented in Python using NumPy and SciPy. Key points:

- Images were treated as 256×256 grayscale arrays normalized to $[0, 1]$.
- Random sampling masks were generated for sampling ratios $r \in \{0.1, 0.2, 0.3, 0.5\}$.
- Additive Gaussian noise was scaled to obtain 30 dB SNR on the sampled pixels.
- Parameters swept: $p \in \{0.3, 0.4, 0.5\}$, and $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$.
- Each MM iteration recomputes weights, assembles the operator via DCT/IDCT and calls PCG.

4 Experimental setup

We ran experiments on two standard test images: *Cameraman* and *Lena*. For each image and sampling ratio:

1. Create a random binary sampling mask with fraction r of pixels observed.
2. Form noisy observations at the sampled pixels with target SNR = 30 dB.
3. Run MM until convergence or until a maximum number of iterations.
4. Record PSNR, ℓ_2 relative error and wall-clock time.

5 Results

5.1 Cameraman

Table 1 summarizes best results (best λ per (r, p)). Visual examples and diagnostics for the best case at $r = 0.5$, $p = 0.5$ are in Figures 1 and 2.

Table 1: Cameraman: best reconstruction metrics for each (r, p) .

r	p	Best λ	Best PSNR (dB)	Rel. Error (ℓ_2)	Time (s)
0.1	0.3	0.1	17.40	0.2548	4.37
0.1	0.4	0.1	18.51	0.2243	3.54
0.1	0.5	0.1	18.57	0.2226	3.39
0.2	0.3	0.1	19.14	0.2084	3.50
0.2	0.4	0.1	20.34	0.1816	3.50
0.2	0.5	0.1	20.38	0.1807	3.44
0.3	0.3	0.1	20.50	0.1783	3.28
0.3	0.4	0.1	21.61	0.1569	3.59
0.3	0.5	0.01	21.83	0.1529	3.45
0.5	0.3	0.01	23.45	0.1269	3.63
0.5	0.4	0.01	24.56	0.1117	3.89
0.5	0.5	0.01	24.88	0.1077	3.36

Visual Results ($r=0.5$, $p=0.5$, $\lambda=0.01$)

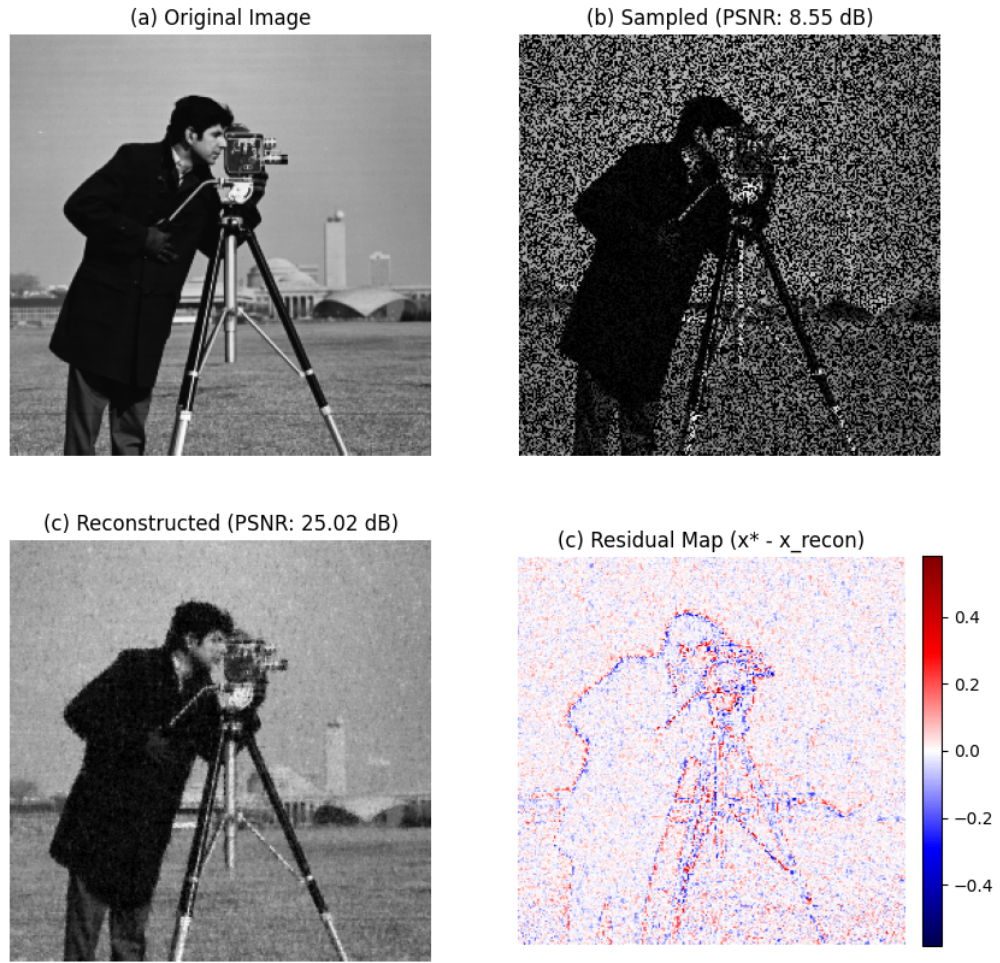


Figure 1: Visual comparison for Cameraman

Diagnostic Plots ($r=0.5$, $p=0.5$, $\lambda=0.01$)

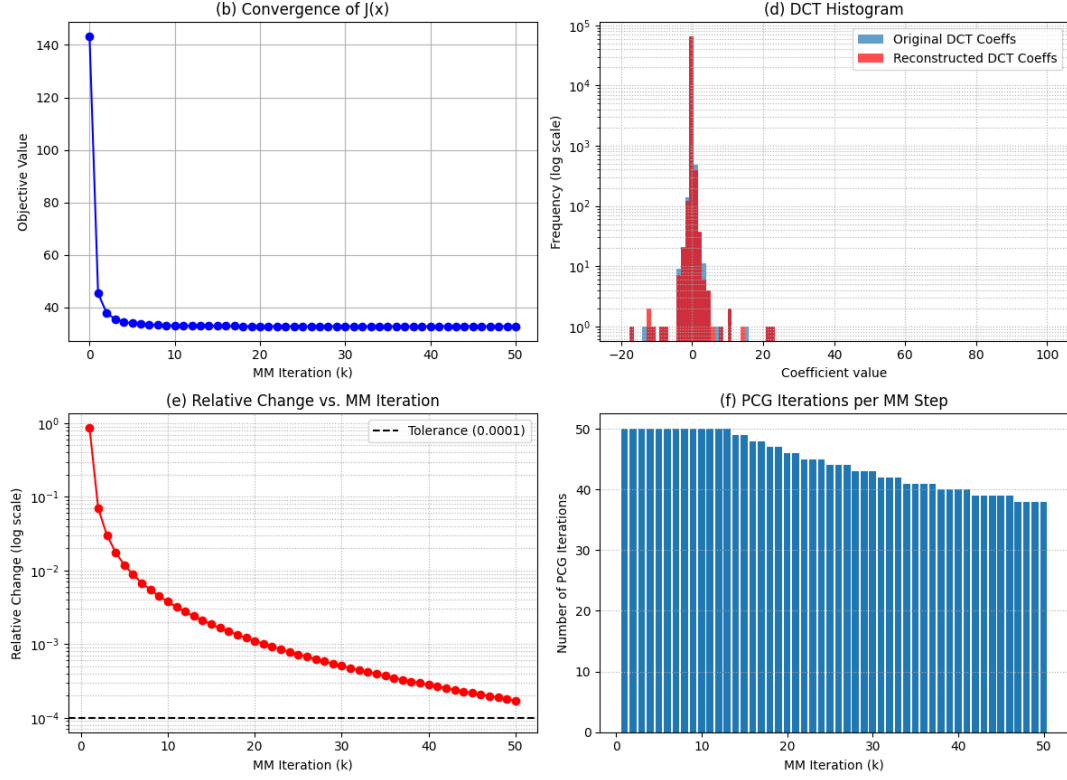


Figure 2: Diagnostic plots for the Cameraman reconstruction at $r=0.5$, $p=0.5$

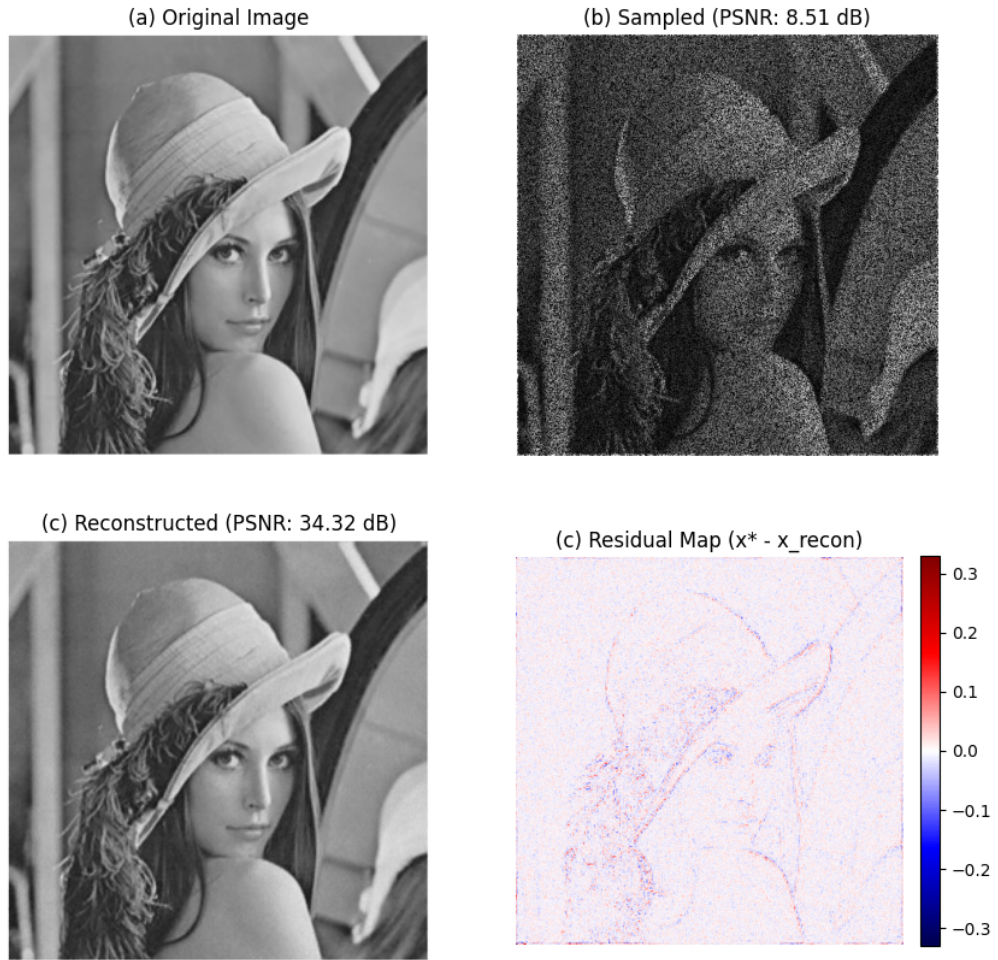
5.2 Lena

Table 2 lists the best-performing λ and metrics for different (r, p) . Representative images and diagnostics for $r = 0.5$, $p = 0.5$ appear in Figures 3 and 4.

Table 2: Lena: best reconstruction metrics for each (r, p) .

r	p	Best λ	Best PSNR (dB)	Rel. Error (ℓ_2)	Time (s)
0.1	0.4	0.1	19.48	0.2262	3.65
0.1	0.5	0.1	19.43	0.2275	3.69
0.2	0.3	0.1	20.54	0.2003	3.40
0.2	0.4	0.1	21.63	0.1766	3.67
0.2	0.5	0.01	22.10	0.1673	4.03
0.3	0.3	0.1	22.10	0.1673	4.03
0.3	0.4	0.01	23.59	0.1409	4.06
0.3	0.5	0.01	23.59	0.1409	4.06
0.5	0.3	0.01	25.93	0.1076	4.26
0.5	0.4	0.01	26.72	0.0983	5.07
0.5	0.5	0.01	27.08	0.0943	3.72

Visual Results ($r=0.5$, $p=0.5$, $\lambda=0.01$)



(a) Original

Figure 3: Lena example (best case at $r = 0.5$, $p = 0.5$).

Diagnostic Plots ($r=0.5$, $p=0.5$, $\lambda=0.01$)

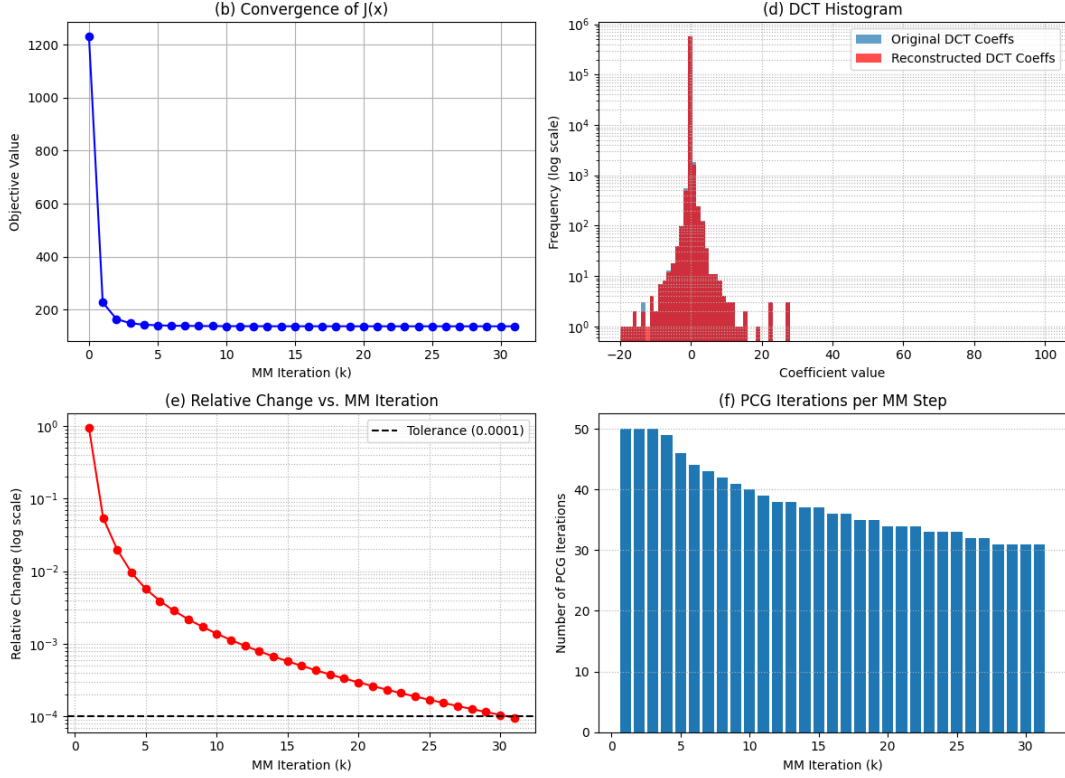


Figure 4: Diagnostic plots for the Lena best case.

6 Discussion

6.1 Sampling fraction

The amount of sampled data is the primary determinant of recovery quality. As the sampling ratio r increases, PSNR improves steadily. This is expected because more direct measurements reduce ambiguity.

6.2 Sparsity exponent p

Larger p values (closer to 1) lead to milder sparsity enforcement. Empirically this often improved PSNR and preserved subtle textures. Very small p values can over-suppress moderate DCT coefficients and harm detail.

6.3 Regularization strength λ

When measured data are scarce, stronger regularization (larger λ) is beneficial. With more samples, a smaller λ lets the algorithm fit observed pixels more closely while relying less on the prior.

6.4 Convergence and artifacts

The MM-PCG scheme converges reliably in a few dozen iterations for the settings used. Sparsity-based priors can introduce small artifacts while filling unobserved regions. These are visible in residual maps and high-frequency DCT bins.

7 Conclusion

We implemented and tested an MM-based solver that enforces DCT-domain sparsity using a nonconvex ℓ_p -style prior. Results on two benchmark images show that reconstruction improves with sampling rate, that moderate p (e.g., 0.5) is often preferable, and that the optimal λ depends on data availability. The approach is computationally efficient when each MM subproblem is solved with PCG and a simple preconditioner.

A Python Implementation Code

This appendix includes the complete Python implementation of the MM-PCG based image reconstruction experiment. All figures and results in the main report were generated using this script.

Source Code

```
1 import numpy as np
2 from PIL import Image
3 from numpy.linalg import norm
4 from scipy.fft import dct, idct
5 import matplotlib.pyplot as plt
6 import time
7 import sys
8
9 # =====
10 # SECTION 1: CORE ALGORITHM FUNCTIONS
11 # =====
12
13 def calculate_psnr(img_true, img_recon):
14     """Calculates the Peak Signal-to-Noise Ratio (PSNR)."""
15     N = img_true.size
16     mse = np.sum((img_true - img_recon)**2) / N
17     if mse == 0:
18         return float('inf')
19     max_intensity = np.max(img_true)
20     psnr_val = 20 * np.log10(max_intensity / np.sqrt(mse))
21     return psnr_val
22
23 def calculate_rel_error(img_true, img_recon):
24     """Calculates the l2 relative error."""
25     return norm(img_recon - img_true) / norm(img_true)
26
27 def calculate_objective(x, m, mask, lam, p_val, epsilon):
28     """Calculates the objective function J(x)."""
29     data_cost = np.sum((mask * x - m)**2)
30     dct_x = dct(dct(x, axis=0, norm='ortho'), axis=1, norm='ortho')
31     sparsity_cost = lam * np.sum((epsilon + dct_x**2)**p_val)
32     return data_cost + sparsity_cost
33
34 def create_random_mask(height, width, sampling_ratio):
35     """Creates a random binary mask."""
36     num_pixels = height * width
37     num_samples = int(num_pixels * sampling_ratio)
38     flat_indices = np.arange(num_pixels)
39     np.random.shuffle(flat_indices)
40     sample_indices = flat_indices[:num_samples]
```

```

41     mask_flat = np.zeros(num_pixels)
42     mask_flat[sample_indices] = 1.0
43     return mask_flat.reshape((height, width))
44
45 def add_snr_noise(image_sampled, mask, target_snr_db=30):
46     """Adds Gaussian noise to achieve a specific SNR."""
47     signal_pixels = image_sampled[mask > 0]
48     signal_norm = np.linalg.norm(signal_pixels)
49     noise_norm = signal_norm / (10**(target_snr_db / 20.0))
50     noise = np.random.randn(len(signal_pixels))
51     scaled_noise = noise * (noise_norm / np.linalg.norm(noise))
52     m = image_sampled.copy()
53     m[mask > 0] += scaled_noise
54     return m
55
56 def apply_M_operator(z, mask, weights, lam):
57     """Applies the linear operator M(k) to an image z."""
58     part1 = mask * z
59     dct_z = dct(dct(z, axis=0, norm='ortho'), axis=1, norm='ortho')
60     weighted_dct = weights * dct_z
61     part2 = idct(idct(weighted_dct, axis=0, norm='ortho'), axis=1, norm='ortho')
62     return part1 + (lam * part2)
63
64 def preconditioned_cg(b, operator_func, x0, preconditioner, tol=1e-6, max_iter=50):
65     """Solves M(x) = b using Preconditioned Conjugate Gradient."""
66     x = x0.copy()
67     r = b - operator_func(x)
68     z = r / preconditioner
69     p = z.copy()
70     rz_old = np.sum(r * z)
71
72     for i in range(max_iter):
73         Ap = operator_func(p)
74         alpha = rz_old / np.sum(p * Ap)
75         x = x + alpha * p
76         r = r - alpha * Ap
77
78         if np.sqrt(np.sum(r*r)) < tol:
79             break
80
81         z = r / preconditioner
82         rz_new = np.sum(r * z)
83         beta = rz_new / rz_old
84         p = z + beta * p
85         rz_old = rz_new
86
87     return x, i + 1
88
89 def run_reconstruction(m, mask, x_true, lam, p_val, epsilon, max_mm_iter=50,
90     ↪ mm_tol=1e-4, cg_tol=1e-6, max_cg_iter=50):
91     """Runs the full MM-PCG reconstruction and returns history."""
92     x_k = m.copy() #  $x^{(0)}$ 
93
94     history = {
95         'obj': [], 'psnr': [], 'rel_change': [], 'cg_iters': []
96     }
97
98     # Record initial state

```

```

98 obj_val = calculate_objective(x_k, m, mask, lam, p_val, epsilon)
99 psnr_val = calculate_psnr(x_true, x_k)
100 history['obj'].append(obj_val)
101 history['psnr'].append(psnr_val)
102
103 for k in range(max_mm_iter):
104     dct_x = dct(dct(x_k, axis=0, norm='ortho'), axis=1, norm='ortho')
105     weights = p_val * (epsilon + dct_x**2)**(p_val - 1)
106
107     b = m
108     operator_for_cg = lambda z: apply_M_operator(z, mask, weights, lam)
109
110     max_weight = np.max(weights)
111     preconditioner = mask + (lam * max_weight)
112     preconditioner[preconditioner == 0] = 1e-6
113
114     x_k_plus_1, cg_iters = preconditioned_cg(
115         b, operator_for_cg, x_k, preconditioner,
116         tol=cg_tol, max_iter=max_cg_iter
117     )
118
119     relative_change = norm(x_k_plus_1 - x_k) / norm(x_k)
120
121     # Calculate and record diagnostics
122     obj_val = calculate_objective(x_k_plus_1, m, mask, lam, p_val, epsilon)
123     psnr_val = calculate_psnr(x_true, x_k_plus_1)
124     history['obj'].append(obj_val)
125     history['psnr'].append(psnr_val)
126     history['rel_change'].append(relative_change)
127     history['cg_iters'].append(cg_iters)
128
129     x_k = x_k_plus_1
130
131     if relative_change < mm_tol and k > 0:
132         break
133
134     # Add final metrics for easy access
135     history['final_psnr'] = history['psnr'][-1]
136     history['final_rel_err'] = calculate_rel_error(x_true, x_k)
137     history['final_recon'] = x_k
138
139     return history
140
141 def plot_diagnostic_graphs(history, x_true, m, p, r, lam):
142     """
143     Generates the 2x2 diagnostic plots (b, c, d, e, f)
144     and the visual result plots.
145     """
146
147     print("\nGenerating diagnostic plots for best case...")
148
149     x_recon = history['final_recon']
150     psnr_input = calculate_psnr(x_true, m)
151     psnr_final = history['final_psnr']
152     mm_tol = 1e-4 # Define for plot
153
154     # Plot 1: Visual Results (Original, Sampled, Recon, Residual)
155     fig, axes = plt.subplots(2, 2, figsize=(10, 10))

```

```

156 fig.suptitle(f'Visual Results (r={r}, p={p}, $\lambda$={lam})', fontsize=16)
157
158 axes[0, 0].imshow(x_true, cmap='gray', vmin=0, vmax=1)
159 axes[0, 0].set_title(f'(a) Original Image')
160 axes[0, 0].axis('off')
161
162 axes[0, 1].imshow(m, cmap='gray', vmin=0, vmax=1)
163 axes[0, 1].set_title(f'(b) Sampled (PSNR: {psnr_input:.2f} dB)')
164 axes[0, 1].axis('off')
165
166 axes[1, 0].imshow(x_recon, cmap='gray', vmin=0, vmax=1)
167 axes[1, 0].set_title(f'(c) Reconstructed (PSNR: {psnr_final:.2f} dB)')
168 axes[1, 0].axis('off')
169
170 residual = x_true - x_recon
171 vmax = np.max(np.abs(residual))
172 im = axes[1, 1].imshow(residual, cmap='seismic', vmin=-vmax, vmax=vmax)
173 axes[1, 1].set_title(f'(c) Residual Map (x* - x_recon)')
174 axes[1, 1].axis('off')
175 fig.colorbar(im, ax=axes[1, 1], fraction=0.046, pad=0.04)
176 plt.savefig('visuals.png')
177
178 # Plot 2: Diagnostic Plots
179 fig, axes = plt.subplots(2, 2, figsize=(12, 10))
180 fig.suptitle(f'Diagnostic Plots (r={r}, p={p}, $\lambda$={lam})', fontsize=16)
181 mm_iters = np.arange(len(history['obj']))
182
183 # Plot (b) Convergence of J(x)
184 axes[0, 0].plot(mm_iters, history['obj'], 'bo-')
185 axes[0, 0].set_title(f'(b) Convergence of J(x)')
186 axes[0, 0].set_xlabel('MM Iteration (k)')
187 axes[0, 0].set_ylabel('Objective Value')
188 axes[0, 0].grid(True)
189
190 # Plot (d) DCT Histogram
191 dct_true_flat = dct(dct(x_true, axis=0, norm='ortho'), axis=1,
192   ↪ norm='ortho').ravel()
193 dct_recon_flat = dct(dct(x_recon, axis=0, norm='ortho'), axis=1,
194   ↪ norm='ortho').ravel()
195 axes[0, 1].hist(dct_true_flat, bins=100, range=(-20, 100), log=True,
196   alpha=0.7, label='Original DCT Coeffs')
197 axes[0, 1].hist(dct_recon_flat, bins=100, range=(-20, 100), log=True,
198   alpha=0.7, label='Reconstructed DCT Coeffs', color='red')
199 axes[0, 1].set_title('(d) DCT Histogram')
200 axes[0, 1].set_xlabel('Coefficient value')
201 axes[0, 1].set_ylabel('Frequency (log scale)')
202 axes[0, 1].legend()
203 axes[0, 1].grid(True, which='both', linestyle=':')
204
205 # Plot (e) Relative Change
206 axes[1, 0].plot(mm_iters[1:], history['rel_change'], 'ro-')
207 axes[1, 0].set_title('(e) Relative Change vs. MM Iteration')
208 axes[1, 0].set_xlabel('MM Iteration (k)')
209 axes[1, 0].set_ylabel('Relative Change (log scale)')
210 axes[1, 0].set_yscale('log')
211 axes[1, 0].axhline(y=mm_tol, color='k', linestyle='--', label=f'Tolerance
212   ↪ ({mm_tol})')
213 axes[1, 0].legend()

```

```

211 axes[1, 0].grid(True, which='both', linestyle=':')
212
213 # Plot (f) PCG Iterations
214 axes[1, 1].bar(mm_iters[1:], history['cg_iters'])
215 axes[1, 1].set_title('(f) PCG Iterations per MM Step')
216 axes[1, 1].set_xlabel('MM Iteration (k)')
217 axes[1, 1].set_ylabel('Number of PCG Iterations')
218 axes[1, 1].grid(True, axis='y', linestyle=':')
219
220 plt.tight_layout(rect=[0, 0.03, 1, 0.95])
221 plt.savefig('diagnostic_plots.png')
222
223 def save_table_as_figure(data_list_of_dicts, col_headers, title, filename):
224     """
225     Saves the final results table as a PNG image.
226     """
227     print(f"\nGenerating table image: {filename}...")
228
229     # 1. Convert data from list-of-dicts to list-of-lists (strings)
230     cell_text = []
231     for run_data in data_list_of_dicts:
232         row = [
233             f"{run_data['r']:.1f}",
234             f"{run_data['p']:.1f}",
235             f"{run_data['best_lambda']:.2g}", # Use .2g for 0.01, 0.1
236             f"{run_data['best_psnr']:.2f}",
237             f"{run_data['rel_err']:.4f}",
238             f"{run_data['runtime']:.2f}"
239         ]
240         cell_text.append(row)
241
242     # 2. Create the figure and table
243     # Adjust figsize; (width, height)
244     fig, ax = plt.subplots(figsize=(12, len(cell_text) * 0.4 + 1))
245     ax.axis('off') # Hide axes (x, y)
246     ax.axis('tight')
247
248     # Create the table
249     table = ax.table(cellText=cell_text,
250                     colLabels=col_headers,
251                     loc='center',
252                     cellLoc='center')
253
254     # 3. Style the table
255     table.auto_set_font_size(False)
256     table.set_fontsize(10)
257     table.scale(1.1, 1.4) # Adjust scale (width, height)
258
259     # Style header row
260     for (i, j), cell in table.get_celld().items():
261         if i == 0: # Header row
262             cell.set_text_props(weight='bold')
263
264     # 4. Add title
265     plt.title(title, weight='bold', fontsize=12, y=1.08)
266
267     # 5. Save the figure
268     plt.savefig(filename, bbox_inches='tight', dpi=200, pad_inches=0.1)

```

```

269 plt.close(fig) # Close the figure to free memory
270 print(f"Saved table to {filename}")
271
272 # =====
273 # SECTION 2: MAIN EXPERIMENT SCRIPT
274 # =====
275 if __name__ == '__main__':
276
277     # --- 1. Define All Experiment Parameters ---
278
279     IMAGE_PATH = 'images/lena.png'
280     EPSILON = 1e-6
281     MM_TOL = 1e-4
282     MAX_MM_ITER = 50 # Set a fixed number for consistent timing
283
284     # Parameters to sweep
285     r_values = [0.1, 0.2, 0.3, 0.5]
286     p_values = [0.3, 0.4, 0.5]
287     lambda_values = [1e-4, 1e-3, 1e-2, 1e-1, 1.0]
288
289     col_headers = ["Sampling (r)", "p-value", "Best Lambda",
290                   "Best PSNR (dB)", "Rel. Error (l2)", "Runtime (s)"]
291
292     # --- 2. Load the Original Image ---
293     try:
294         x_true = np.array(Image.open(IMAGE_PATH).convert('L')) / 255.0
295     except FileNotFoundError:
296         print(f"Error: Image '{IMAGE_PATH}' not found.")
297         sys.exit()
298
299     print(f"--- Running Full Experiment Sweep for '{IMAGE_PATH}' ---")
300
301     # Print the table header
302     print("="*70)
303     print(f"{'Sampling (r)':<12} | {'p-value':<7} | {'Best Lambda':<11} | {'Best PSNR  

304     ↳ (dB)':<15} | {'Rel. Error (l2)':<15} | {'Runtime (s)':<10}")
305     print("-"*70)
306
307     # --- 3. Start the Triple Loop ---
308
309     all_best_results = []
310     # This will store the data needed for Plot (a)
311     psnr_vs_lambda_data_r0_5_p0_5 = []
312     # This will store the full history for the best run for Plots (b-f)
313     best_history_r0_5_p0_5 = None
314     best_lambda_for_plots = None
315     best_psnr_for_plots = -1.0
316
317     # --- Loop 1: Sampling Ratio (r) ---
318     for r in r_values:
319
320         # Create mask and noisy data ONCE for this r
321         mask = create_random_mask(x_true.shape[0], x_true.shape[1], r)
322         x_sampled = x_true * mask
323         m = add_snr_noise(x_sampled, mask, target_snr_db=30)
324
325         # --- Loop 2: p-value ---
326         for p in p_values:

```

```

326 sweep_results = []
327
328
329 # --- Loop 3: Lambda ( $\lambda$ ) ---
330 for lam in lambda_values:
331
332     print(f" Running: r={r}, p={p}, \u03BB={lam:.1e}...")
333
334     start_time = time.time()
335
336     history = run_reconstruction(
337         m, mask, x_true,
338         lam=lam, p_val=p, epsilon=EPSILON,
339         max_mm_iter=MAX_MM_ITER, mm_tol=MM_TOL
340     )
341
342     end_time = time.time()
343     runtime = end_time - start_time
344
345     # Store the results of this single run
346     run_data = {
347         'lambda': lam,
348         'psnr': history['final_psnr'],
349         'rel_err': history['final_rel_err'],
350         'runtime': runtime,
351         'history': history # Store the full history
352     }
353     sweep_results.append(run_data)
354
355     # --- Special step: Save data for the plots ---
356     # If this is the specific case we want to plot (r=0.5, p=0.5)
357     if r == 0.5 and p == 0.5:
358         psnr_vs_lambda_data_r0_5_p0_5.append(run_data)
359
360         # Check if this is the best PSNR *for this case*
361         if run_data['psnr'] > best_psnr_for_plots:
362             best_psnr_for_plots = run_data['psnr']
363             best_history_r0_5_p0_5 = history
364             best_lambda_for_plots = lam
365
366     # --- Find the BEST result from the lambda sweep ---
367     best_run = max(sweep_results, key=lambda x: x['psnr'])
368     table_row_data = {
369         'r': r,
370         'p': p,
371         'best_lambda': best_run['lambda'],
372         'best_psnr': best_run['psnr'],
373         'rel_err': best_run['rel_err'],
374         'runtime': best_run['runtime']
375     }
376     # Store the best one for the table
377     all_best_results.append(table_row_data)
378
379     # Print the row for the table
380     print(f"{r:<12.1f} | {p:<7.1f} | {best_run['lambda']:<7.2g} |
381           \u2192 {best_run['psnr']:<15.2f} | {best_run['rel_err']:<15.4f} |
382           \u2192 {best_run['runtime']:<10.2f}")

```

```

382 print("="*80)
383 print("--- Experiment Sweep Complete ---")
384
385 # --- 4. Generate Plot (a) ---
386 print("\nGenerating 'psnr_vs_lambda_plot.png'...")
387
388 # Extract data for plotting
389 lambdas = [run['lambda'] for run in psnr_vs_lambda_data_r0_5_p0_5]
390 psnrns = [run['psnr'] for run in psnr_vs_lambda_data_r0_5_p0_5]
391
392 plt.figure(figsize=(8, 6))
393 plt.plot(lambdas, psnrns, 'bo-')
394 plt.title(f'(a) PSNR vs.  $\lambda$  (r=0.5, p=0.5)')
395 plt.xlabel(f' $\lambda$  (Lambda)')
396 plt.ylabel('PSNR (dB)')
397 plt.xscale('log')
398 plt.grid(True, which='both', linestyle=':')
399 plt.savefig('psnr_vs_lambda_plot.png')
400
401 # --- 5. Generate Plots (b, c, d, e, f) ---
402 if best_history_r0_5_p0_5:
403     # Re-create the mask and noise for this specific case to plot
404     r_plot, p_plot = 0.5, 0.5
405     mask_plot = create_random_mask(x_true.shape[0], x_true.shape[1], r_plot)
406     x_sampled_plot = x_true * mask_plot
407     m_plot = add_snr_noise(x_sampled_plot, mask_plot, target_snr_db=30)
408
409     plot_diagnostic_graphs(best_history_r0_5_p0_5,
410                           x_true, m_plot,
411                           p=p_plot, r=r_plot, lam=best_lambda_for_plots)
412     print("Generated 'final_reconstruction_visuals.png' and
413           ↪ 'final_diagnostic_plots.png'")
414
415 # --- 6. NEW: Save the final table as a PNG ---
416 table_title = f'Table: Best Reconstruction Metrics for {IMAGE_PATH}'
417 save_table_as_figure(all_best_results, col_headers, table_title,
418                     ↪ 'results_table.png')
419
420 print("\nAll tasks complete.")

```

Repository Link

The complete implementation, figures, and dataset are available on GitHub:

<https://github.com/vinaaaaaay/LN0-Project.git>