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

Vinay
SR NO.: 23653
B.Tech : Mathematics & Computing

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 $x^\star \in \mathbb{R}^N$ 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

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

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

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

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

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

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

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

$$\phi(t) \;\le\; \phi(s) + \phi'(s)\,(t - s) \qquad \text{(first-order tangent upper bound).} \tag{3}$$

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

$$\phi(y_i^2) \;\le\; \phi\big((y_i^{(k)})^2\big) + \underbrace{\phi'\big((y_i^{(k)})^2\big)}_{w_i^{(k)}} \Big(y_i^2 - (y_i^{(k)})^2\Big), \qquad w_i^{(k)} = p\big(\varepsilon + (y_i^{(k)})^2\big)^{p-1} \ge 0. \tag{4}$$

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

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

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

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

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

$$\Longrightarrow \qquad \boxed{\Big(W^\top W \;+\; \lambda\, C^\top \mathrm{diag}(w^{(k)})C\Big)\, x \;=\; W^\top m.} \tag{7}$$

Using IDCT $= C^\top$ and DCT $= C$,

$$\Big(W^\top W \;+\; \lambda\,\mathrm{IDCT}\,\mathrm{diag}(w^{(k)})\,\mathrm{DCT}\Big)\, x \;=\; W^\top m. \tag{8}$$

The coefficient matrix is symmetric positive definite (SPD): $W^\top W \succeq 0$, $\mathrm{diag}(w^{(k)}) \succeq 0$, 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 $A_k := W^\top W + \lambda C^\top \mathrm{diag}(w^{(k)})C$. For any $z \in \mathbb{R}^N$,

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

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

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

which matches the routine `apply_M_operator`.

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

$$C^\top \mathrm{diag}(w^{(k)})C \;\preceq\; \alpha_k\, C^\top I C \;=\; \alpha_k I. \tag{10}$$

Thus a cheap diagonal upper bound for A_k is

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

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

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

with standard PCG recurrences.

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

1. Compute $y^{(k)} = Cx^{(k)}$ and weights $w_i^{(k)} = p\big(\varepsilon + (y_i^{(k)})^2\big)^{p-1}$.
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 $\frac{\|x^{(k+1)} - x^{(k)}\|_2}{\|x^{(k)}\|_2} <$ `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: $w^{(k)} \leftarrow p(\varepsilon + (\mathrm{DCT}(x^{(k)}))^2)^{p-1}$.
- System application: $z \mapsto \mathrm{mask} \cdot z + \lambda\, \mathrm{IDCT}\big(w^{(k)} \odot \mathrm{DCT}(z)\big)$.
- Preconditioner: $P_k = \mathrm{mask} + \lambda \max_i w_i^{(k)}$ (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

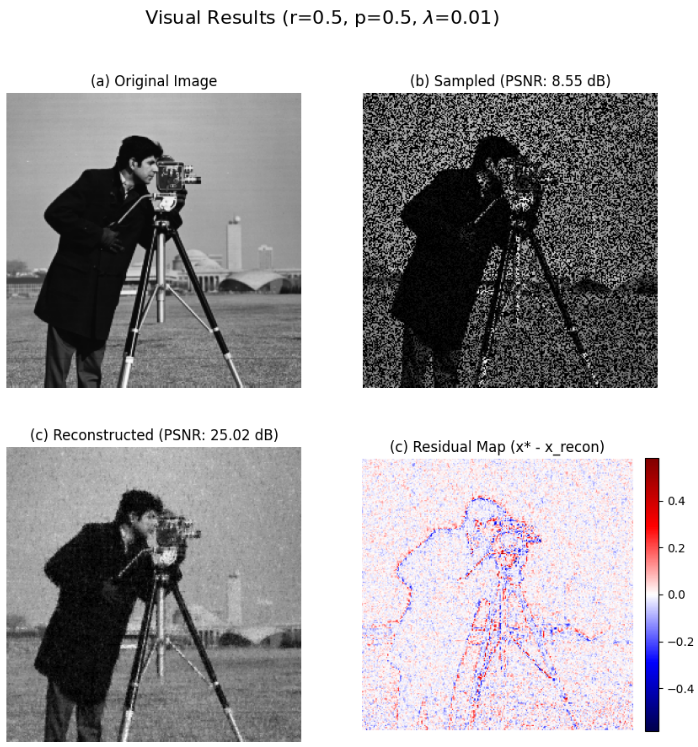


Figure 1: Visual comparison for Cameraman

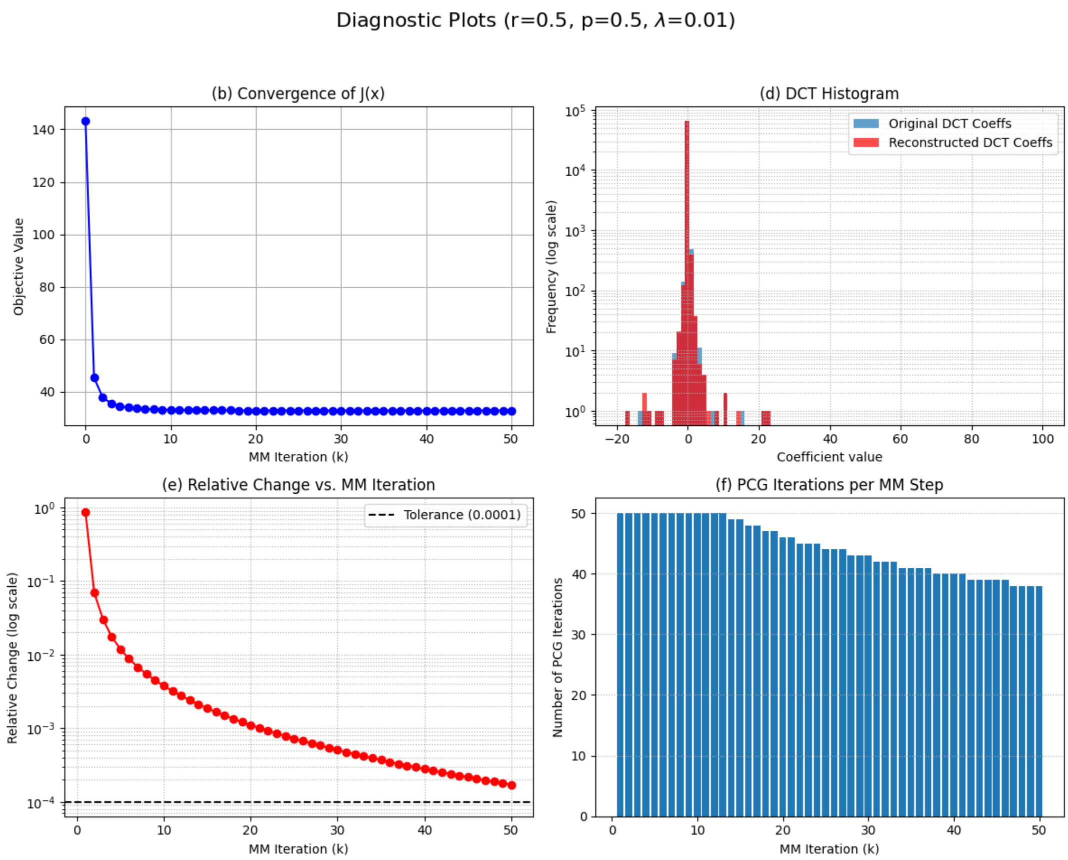


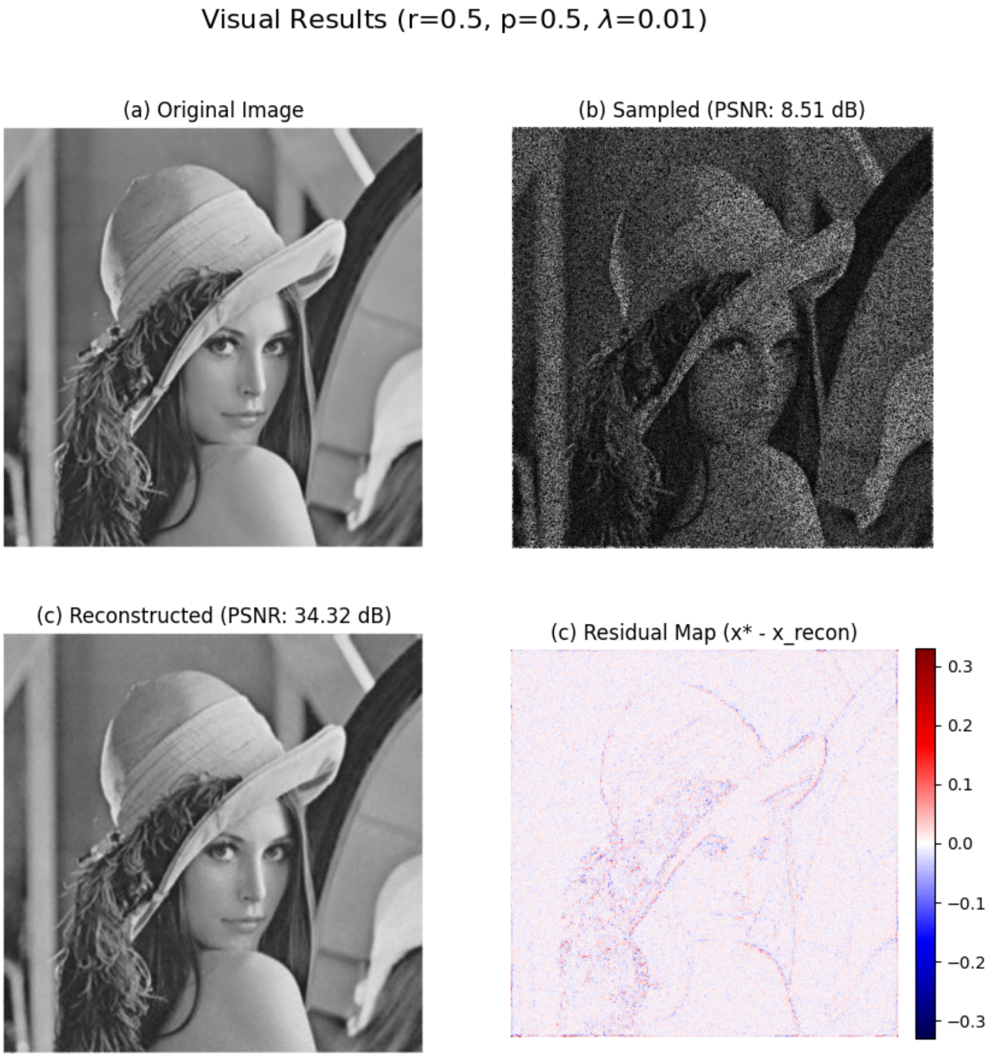
Figure 2: Diagnostic plots for the Cameraman reconstruction at r=0.5, p=0.5

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



(a) Original

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

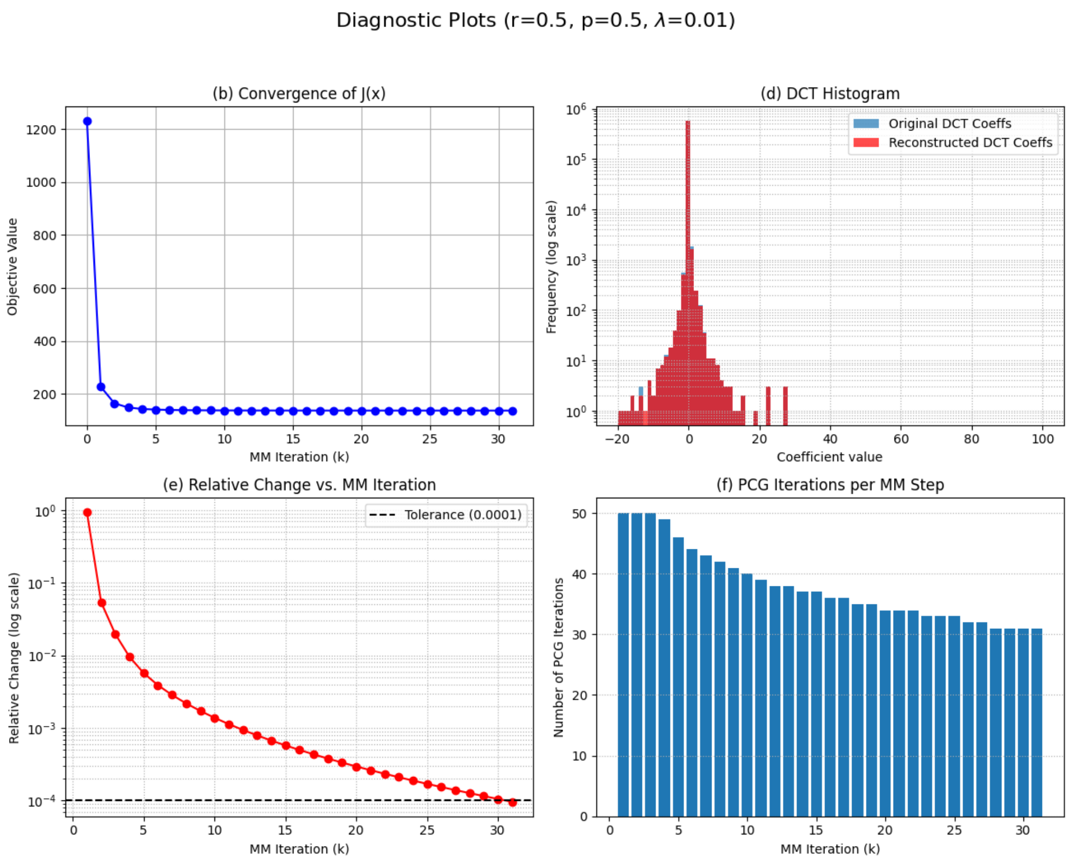


Figure 4: Diagnostic plots for the Lena best case.

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

```
import numpy as np
from PIL import Image
from numpy.linalg import norm
from scipy.fft import dct, idct
import matplotlib.pyplot as plt
import time
import sys

# ====================================
# SECTION 1: CORE ALGORITHM FUNCTIONS
# ====================================

def calculate_psnr(img_true, img_recon):
    """Calculates the Peak Signal-to-Noise Ratio (PSNR)."""
    N = img_true.size
    mse = np.sum((img_true - img_recon)**2) / N
    if mse == 0:
        return float('inf')
    max_intensity = np.max(img_true)
    psnr_val = 20 * np.log10(max_intensity / np.sqrt(mse))
    return psnr_val

def calculate_rel_error(img_true, img_recon):
    """Calculates the l2 relative error."""
    return norm(img_recon - img_true) / norm(img_true)

def calculate_objective(x, m, mask, lam, p_val, epsilon):
    """Calculates the objective function J(x)."""
    data_cost = np.sum((mask * x - m)**2)
    dct_x = dct(dct(x, axis=0, norm='ortho'), axis=1, norm='ortho')
    sparsity_cost = lam * np.sum((epsilon + dct_x**2)**p_val)
    return data_cost + sparsity_cost

def create_random_mask(height, width, sampling_ratio):
    """Creates a random binary mask."""
    num_pixels = height * width
    num_samples = int(num_pixels * sampling_ratio)
    flat_indices = np.arange(num_pixels)
    np.random.shuffle(flat_indices)
    sample_indices = flat_indices[:num_samples]
```

```python
    mask_flat = np.zeros(num_pixels)
    mask_flat[sample_indices] = 1.0
    return mask_flat.reshape((height, width))

def add_snr_noise(image_sampled, mask, target_snr_db=30):
    """Adds Gaussian noise to achieve a specific SNR."""
    signal_pixels = image_sampled[mask > 0]
    signal_norm = np.linalg.norm(signal_pixels)
    noise_norm = signal_norm / (10**(target_snr_db / 20.0))
    noise = np.random.randn(len(signal_pixels))
    scaled_noise = noise * (noise_norm / np.linalg.norm(noise))
    m = image_sampled.copy()
    m[mask > 0] += scaled_noise
    return m

def apply_M_operator(z, mask, weights, lam):
    """Applies the linear operator M(k) to an image z."""
    part1 = mask * z
    dct_z = dct(dct(z, axis=0, norm='ortho'), axis=1, norm='ortho')
    weighted_dct = weights * dct_z
    part2 = idct(idct(weighted_dct, axis=0, norm='ortho'), axis=1, norm='ortho')
    return part1 + (lam * part2)

def preconditioned_cg(b, operator_func, x0, preconditioner, tol=1e-6, max_iter=50):
    """Solves M(x) = b using Preconditioned Conjugate Gradient."""
    x = x0.copy()
    r = b - operator_func(x)
    z = r / preconditioner
    p = z.copy()
    rz_old = np.sum(r * z)

    for i in range(max_iter):
        Ap = operator_func(p)
        alpha = rz_old / np.sum(p * Ap)
        x = x + alpha * p
        r = r - alpha * Ap

        if np.sqrt(np.sum(r*r)) < tol:
            break

        z = r / preconditioner
        rz_new = np.sum(r * z)
        beta = rz_new / rz_old
        p = z + beta * p
        rz_old = rz_new

    return x, i + 1

def run_reconstruction(m, mask, x_true, lam, p_val, epsilon, max_mm_iter=50,
↪   mm_tol=1e-4, cg_tol=1e-6, max_cg_iter=50):
    """Runs the full MM-PCG reconstruction and returns history."""
    x_k = m.copy() # x^(0)

    history = {
        'obj': [], 'psnr': [], 'rel_change': [], 'cg_iters': []
    }

    # Record initial state
```

```python
    obj_val = calculate_objective(x_k, m, mask, lam, p_val, epsilon)
    psnr_val = calculate_psnr(x_true, x_k)
    history['obj'].append(obj_val)
    history['psnr'].append(psnr_val)

    for k in range(max_mm_iter):
        dct_x = dct(dct(x_k, axis=0, norm='ortho'), axis=1, norm='ortho')
        weights = p_val * (epsilon + dct_x**2)**(p_val - 1)

        b = m
        operator_for_cg = lambda z: apply_M_operator(z, mask, weights, lam)

        max_weight = np.max(weights)
        preconditioner = mask + (lam * max_weight)
        preconditioner[preconditioner == 0] = 1e-6

        x_k_plus_1, cg_iters = preconditioned_cg(
            b, operator_for_cg, x_k, preconditioner,
            tol=cg_tol, max_iter=max_cg_iter
        )

        relative_change = norm(x_k_plus_1 - x_k) / norm(x_k)

        # Calculate and record diagnostics
        obj_val = calculate_objective(x_k_plus_1, m, mask, lam, p_val, epsilon)
        psnr_val = calculate_psnr(x_true, x_k_plus_1)
        history['obj'].append(obj_val)
        history['psnr'].append(psnr_val)
        history['rel_change'].append(relative_change)
        history['cg_iters'].append(cg_iters)

        x_k = x_k_plus_1

        if relative_change < mm_tol and k > 0:
            break

    # Add final metrics for easy access
    history['final_psnr'] = history['psnr'][-1]
    history['final_rel_err'] = calculate_rel_error(x_true, x_k)
    history['final_recon'] = x_k

    return history

def plot_diagnostic_graphs(history, x_true, m, p, r, lam):
    """
    Generates the 2x2 diagnostic plots (b, c, d, e, f)
    and the visual result plots.
    """

    print("\nGenerating diagnostic plots for best case...")

    x_recon = history['final_recon']
    psnr_input = calculate_psnr(x_true, m)
    psnr_final = history['final_psnr']
    mm_tol = 1e-4 # Define for plot

    # Plot 1: Visual Results (Original, Sampled, Recon, Residual)
    fig, axes = plt.subplots(2, 2, figsize=(10, 10))
```

```python
    fig.suptitle(f'Visual Results (r={r}, p={p}, $\lambda$={lam})', fontsize=16)

    axes[0, 0].imshow(x_true, cmap='gray', vmin=0, vmax=1)
    axes[0, 0].set_title(f'(a) Original Image')
    axes[0, 0].axis('off')

    axes[0, 1].imshow(m, cmap='gray', vmin=0, vmax=1)
    axes[0, 1].set_title(f'(b) Sampled (PSNR: {psnr_input:.2f} dB)')
    axes[0, 1].axis('off')

    axes[1, 0].imshow(x_recon, cmap='gray', vmin=0, vmax=1)
    axes[1, 0].set_title(f'(c) Reconstructed (PSNR: {psnr_final:.2f} dB)')
    axes[1, 0].axis('off')

    residual = x_true - x_recon
    vmax = np.max(np.abs(residual))
    im = axes[1, 1].imshow(residual, cmap='seismic', vmin=-vmax, vmax=vmax)
    axes[1, 1].set_title(f'(c) Residual Map (x* - x_recon)')
    axes[1, 1].axis('off')
    fig.colorbar(im, ax=axes[1, 1], fraction=0.046, pad=0.04)
    plt.savefig('visuals.png')

    # Plot 2: Diagnostic Plots
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
    fig.suptitle(f'Diagnostic Plots (r={r}, p={p}, $\lambda$={lam})', fontsize=16)
    mm_iters = np.arange(len(history['obj']))

    # Plot (b) Convergence of J(x)
    axes[0, 0].plot(mm_iters, history['obj'], 'bo-')
    axes[0, 0].set_title(f'(b) Convergence of J(x)')
    axes[0, 0].set_xlabel('MM Iteration (k)')
    axes[0, 0].set_ylabel('Objective Value')
    axes[0, 0].grid(True)

    # Plot (d) DCT Histogram
    dct_true_flat = dct(dct(x_true, axis=0, norm='ortho'), axis=1,
    ↪   norm='ortho').ravel()
    dct_recon_flat = dct(dct(x_recon, axis=0, norm='ortho'), axis=1,
    ↪   norm='ortho').ravel()
    axes[0, 1].hist(dct_true_flat, bins=100, range=(-20, 100), log=True,
                    alpha=0.7, label='Original DCT Coeffs')
    axes[0, 1].hist(dct_recon_flat, bins=100, range=(-20, 100), log=True,
                    alpha=0.7, label='Reconstructed DCT Coeffs', color='red')
    axes[0, 1].set_title('(d) DCT Histogram')
    axes[0, 1].set_xlabel('Coefficient value')
    axes[0, 1].set_ylabel('Frequency (log scale)')
    axes[0, 1].legend()
    axes[0, 1].grid(True, which='both', linestyle=':')

    # Plot (e) Relative Change
    axes[1, 0].plot(mm_iters[1:], history['rel_change'], 'ro-')
    axes[1, 0].set_title('(e) Relative Change vs. MM Iteration')
    axes[1, 0].set_xlabel('MM Iteration (k)')
    axes[1, 0].set_ylabel('Relative Change (log scale)')
    axes[1, 0].set_yscale('log')
    axes[1, 0].axhline(y=mm_tol, color='k', linestyle='--', label=f'Tolerance
    ↪   ({mm_tol})')
    axes[1, 0].legend()
```

```python
    axes[1, 0].grid(True, which='both', linestyle=':')

    # Plot (f) PCG Iterations
    axes[1, 1].bar(mm_iters[1:], history['cg_iters'])
    axes[1, 1].set_title('(f) PCG Iterations per MM Step')
    axes[1, 1].set_xlabel('MM Iteration (k)')
    axes[1, 1].set_ylabel('Number of PCG Iterations')
    axes[1, 1].grid(True, axis='y', linestyle=':')

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.savefig('diagnostic_plots.png')

def save_table_as_figure(data_list_of_dicts, col_headers, title, filename):
    """
    Saves the final results table as a PNG image.
    """
    print(f"\nGenerating table image: {filename}...")

    # 1. Convert data from list-of-dicts to list-of-lists (strings)
    cell_text = []
    for run_data in data_list_of_dicts:
        row = [
            f"{run_data['r']:.1f}",
            f"{run_data['p']:.1f}",
            f"{run_data['best_lambda']:.2g}", # Use .2g for 0.01, 0.1
            f"{run_data['best_psnr']:.2f}",
            f"{run_data['rel_err']:.4f}",
            f"{run_data['runtime']:.2f}"
        ]
        cell_text.append(row)

    # 2. Create the figure and table
    # Adjust figsize; (width, height)
    fig, ax = plt.subplots(figsize=(12, len(cell_text) * 0.4 + 1))
    ax.axis('off') # Hide axes (x, y)
    ax.axis('tight')

    # Create the table
    table = ax.table(cellText=cell_text,
                     colLabels=col_headers,
                     loc='center',
                     cellLoc='center')

    # 3. Style the table
    table.auto_set_font_size(False)
    table.set_fontsize(10)
    table.scale(1.1, 1.4) # Adjust scale (width, height)

    # Style header row
    for (i, j), cell in table.get_celld().items():
        if i == 0: # Header row
            cell.set_text_props(weight='bold')

    # 4. Add title
    plt.title(title, weight='bold', fontsize=12, y=1.08)

    # 5. Save the figure
    plt.savefig(filename, bbox_inches='tight', dpi=200, pad_inches=0.1)
```

```
    plt.close(fig) # Close the figure to free memory
    print(f"Saved table to {filename}")

# ========================================
# SECTION 2: MAIN EXPERIMENT SCRIPT
# ========================================
if __name__ == '__main__':

    # --- 1. Define All Experiment Parameters ---

    IMAGE_PATH = 'images/lena.png'
    EPSILON = 1e-6
    MM_TOL = 1e-4
    MAX_MM_ITER = 50 # Set a fixed number for consistent timing

    # Parameters to sweep
    r_values = [0.1, 0.2, 0.3, 0.5]
    p_values = [0.3, 0.4, 0.5]
    lambda_values = [1e-4, 1e-3, 1e-2, 1e-1, 1.0]

    col_headers = ["Sampling (r)", "p-value", "Best Lambda",
                   "Best PSNR (dB)", "Rel. Error (l2)", "Runtime (s)"]

    # --- 2. Load the Original Image ---
    try:
        x_true = np.array(Image.open(IMAGE_PATH).convert('L')) / 255.0
    except FileNotFoundError:
        print(f"Error: Image '{IMAGE_PATH}' not found.")
        sys.exit()

    print(f"--- Running Full Experiment Sweep for '{IMAGE_PATH}' ---")

    # Print the table header
    print("="*70)
    print(f"{'Sampling (r)':<12} | {'p-value':<7} | {'Best Lambda':<11} | {'Best PSNR (dB)':<15} | {'Rel. Error (l2)':<15} | {'Runtime (s)':<10}")
    print("-"*70)

    # --- 3. Start the Triple Loop ---

    all_best_results = []
    # This will store the data needed for Plot (a)
    psnr_vs_lambda_data_r0_5_p0_5 = []
    # This will store the full history for the best run for Plots (b-f)
    best_history_r0_5_p0_5 = None
    best_lambda_for_plots = None
    best_psnr_for_plots = -1.0

    # --- Loop 1: Sampling Ratio (r) ---
    for r in r_values:

        # Create mask and noisy data ONCE for this r
        mask = create_random_mask(x_true.shape[0], x_true.shape[1], r)
        x_sampled = x_true * mask
        m = add_snr_noise(x_sampled, mask, target_snr_db=30)

        # --- Loop 2: p-value ---
        for p in p_values:
```

```
            sweep_results = []

            # --- Loop 3: Lambda (λ) ---
            for lam in lambda_values:

                print(f"  Running: r={r}, p={p}, \u03BB={lam:.1e}...")

                start_time = time.time()

                history = run_reconstruction(
                    m, mask, x_true,
                    lam=lam, p_val=p, epsilon=EPSILON,
                    max_mm_iter=MAX_MM_ITER, mm_tol=MM_TOL
                )

                end_time = time.time()
                runtime = end_time - start_time

                # Store the results of this single run
                run_data = {
                    'lambda': lam,
                    'psnr': history['final_psnr'],
                    'rel_err': history['final_rel_err'],
                    'runtime': runtime,
                    'history': history # Store the full history
                }
                sweep_results.append(run_data)

                # --- Special step: Save data for the plots ---
                # If this is the specific case we want to plot (r=0.5, p=0.5)
                if r == 0.5 and p == 0.5:
                    psnr_vs_lambda_data_r0_5_p0_5.append(run_data)

                    # Check if this is the best PSNR *for this case*
                    if run_data['psnr'] > best_psnr_for_plots:
                        best_psnr_for_plots = run_data['psnr']
                        best_history_r0_5_p0_5 = history
                        best_lambda_for_plots = lam

            # --- Find the BEST result from the lambda sweep ---
            best_run = max(sweep_results, key=lambda x: x['psnr'])
            table_row_data = {
                'r': r,
                'p': p,
                'best_lambda': best_run['lambda'],
                'best_psnr': best_run['psnr'],
                'rel_err': best_run['rel_err'],
                'runtime': best_run['runtime']
            }
            # Store the best one for the table
            all_best_results.append(table_row_data)

            # Print the row for the table
            print(f"{r:<12.1f} | {p:<7.1f} | {best_run['lambda']:<7.2g} | {best_run['psnr']:<15.2f} | {best_run['rel_err']:<15.4f} | {best_run['runtime']:<10.2f}")

```

```python
    print("="*80)
    print("--- Experiment Sweep Complete ---")

    # --- 4. Generate Plot (a) ---
    print("\nGenerating 'psnr_vs_lambda_plot.png'...")

    # Extract data for plotting
    lambdas = [run['lambda'] for run in psnr_vs_lambda_data_r0_5_p0_5]
    psnrs = [run['psnr'] for run in psnr_vs_lambda_data_r0_5_p0_5]

    plt.figure(figsize=(8, 6))
    plt.plot(lambdas, psnrs, 'bo-')
    plt.title(f'(a) PSNR vs. $\lambda$ (r=0.5, p=0.5)')
    plt.xlabel('$\lambda$ (Lambda)')
    plt.ylabel('PSNR (dB)')
    plt.xscale('log')
    plt.grid(True, which='both', linestyle=':')
    plt.savefig('psnr_vs_lambda_plot.png')

    # --- 5. Generate Plots (b, c, d, e, f) ---
    if best_history_r0_5_p0_5:
        # Re-create the mask and noise for this specific case to plot
        r_plot, p_plot = 0.5, 0.5
        mask_plot = create_random_mask(x_true.shape[0], x_true.shape[1], r_plot)
        x_sampled_plot = x_true * mask_plot
        m_plot = add_snr_noise(x_sampled_plot, mask_plot, target_snr_db=30)

        plot_diagnostic_graphs(best_history_r0_5_p0_5,
                               x_true, m_plot,
                               p=p_plot, r=r_plot, lam=best_lambda_for_plots)
        print("Generated 'final_reconstruction_visuals.png' and
        ↪  'final_diagnostic_plots.png'")

    # --- 6. NEW: Save the final table as a PNG ---
    table_title = f'Table: Best Reconstruction Metrics for {IMAGE_PATH}'
    save_table_as_figure(all_best_results, col_headers, table_title,
    ↪  'results_table.png')

    print("\nAll tasks complete.")
```

Repository Link

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

https://github.com/vinaaaaay/LNO-Project.git