Homework 4 (DRAFT)

David L. Olson

April 10, 2025

Contents

1 Problem 1

1.1 Solution

1.1.1 Part A

The `landweber()` function was implemented in MATLAB®, and a screenshot of the code is provided in figure 1.

```
function X = landweber(G, m0, d, omega, k)
% Landweber Iteration for Sttepest Decent Method
%
% Performs "k" iterations of the Landweber interation on Gm = d, with
% initial solution m0, and parameter omega.
%
% On output, X is a matrix of size n by k. The k columns of X are the
% iterates m1, m2, ..., mk.
%
% For convergence, we need 0 < omega < 2/s(1)^2. Outside of this range, you
% can expect the method to diverge.

%% Input and Output Arguments

arguments (Input)
    G (:,:) {isfloat, mustBeReal, mustBeFinite}
    m0 (:,1) {isfloat, mustBeReal, mustBeFinite}
    d (:,1) {isfloat, mustBeReal, mustBeFinite}
    omega (1,1) {isfloat, mustBeReal, mustBeFinite, mustBePositive}
    k (1,1) {mustBeInteger, mustBePositive}
end

arguments (Output)
    X (:,:) {isfloat, mustBeReal, mustBeFinite}
end


%% Error Checking

[m, n] = size(G);
assert(...
    n == length(m0), ...
    "Initial model m0 must have the same number of columns as G!");
assert(...
    m == length(d), ...
    "Data vector d must have the same number of rows as G!");

assert(...
    omega < (2/(svds(G,1)^2)), ...
    "This omega value will not let the iterations converge to a solution!")


%% Implementation

X = zeros(n, k);
X(:,1) = m0;
for ii = 2 : k
    X(:,ii) = X(:,ii-1) - omega * G.' * (G*X(:,ii-1) - d);
end


end
```

Figure 1: MATLAB® Landweber Function Screenshot

1.1.2 Part B

MATLAB ® code was modified from `ex_6_2.m` to load image data, apply noise, and plot the resulting images. The fixed step size

$$\omega = 0.95\frac{2}{s_1^2} = 6190.5$$

was computed using the `svds()` function in MATLAB ® . A total of 500 iterations were running using the `landweber()` function shown in figure 1. Iterations 10, 50, 100, 150, 250, and 500 are evaluated.

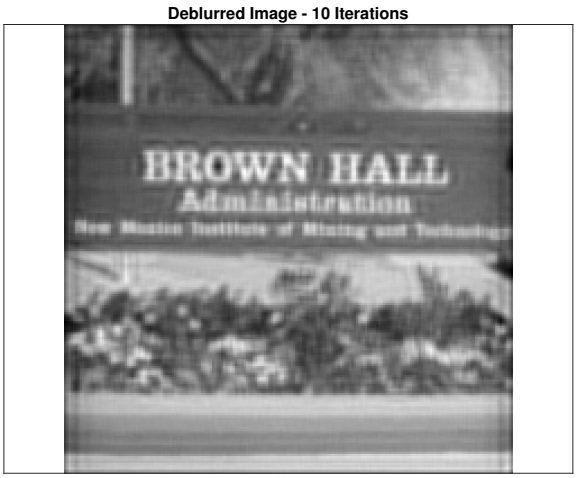

Figure 2: Deblurred Image - 10 Iterations

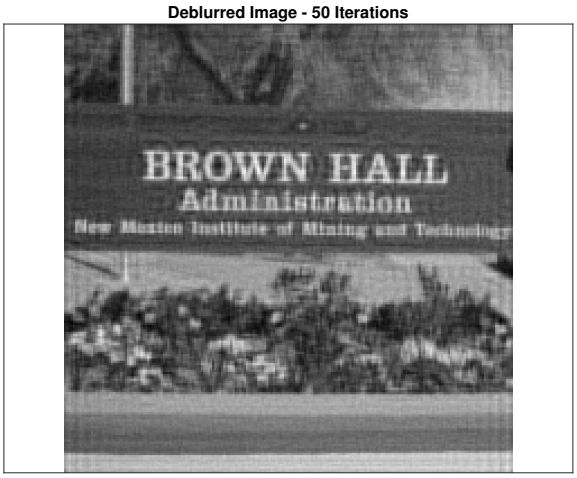

Figure 3: Deblurred Image - 50 Iterations

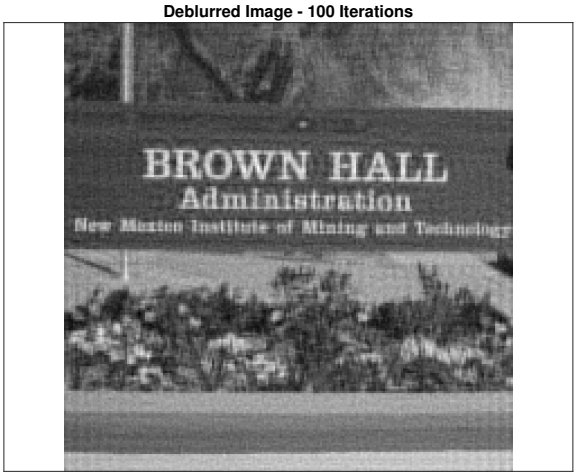

Figure 4: Deblurred Image - 100 Iterations

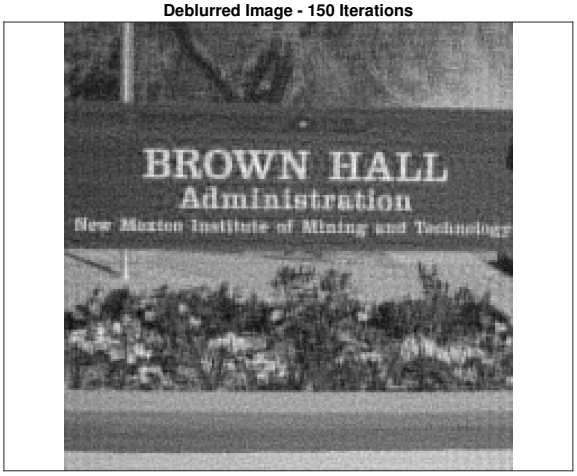

Figure 5: Deblurred Image - 150 Iterations

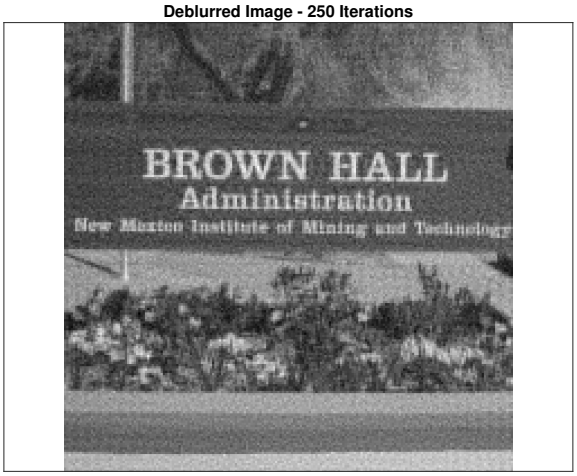

Figure 6: Deblurred Image - 250 Iterations

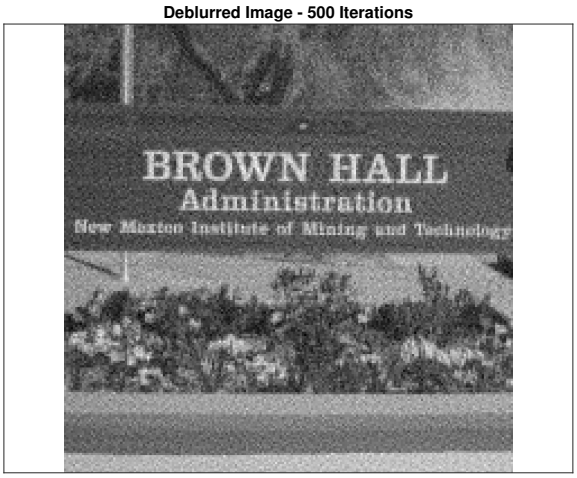

Figure 7: Deblurred Image - 500 Iterations

It becomes clear that there is a "diminishing rate of returns" on the number of iterations used to deblur the image. In my opinion, it seems that the sharpest image appears at 150 iterations. The images after this number of iterations are also as sharp, but from there I can no longer tell a difference.

It is less clear in this PDF, but on my screen in MATLAB it shows just a bit clearer. Unfortunately, you'll just have to take my word for it.

2 Problem 2

Exercise 6 in Section 7.8

2.1 Solution

The image `inpaint.png` of size 512×512 was loaded into MATLAB® and reshaped into a column vector of size 512^2. To repair the image, a logical index of pixels which equal 255, often called a "mask" in MATLAB® was created to identify the bad pixels.

To express the problem in the format $G\boldsymbol{m} = \boldsymbol{d}$, the model operator G is simply the identity matrix with rows corresponding to bad pixels removed. Likewise, the data vector $\boldsymbol{d}$ is simply the column vector of pixels of the image with any bad pixels removed.

$$I_{good}\boldsymbol{m}_{all} = \boldsymbol{d}_{good}$$

The initial identity matrix of size 512^2 would have required $\approx 512GB$ to store in MATLAB®, so a sparse matrix was used instead.

To compute the total variation of the image, an L_1 roughing matrix was formed. The total variation formula is given below.

$$\mathrm{TV}_1(m) = \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} |M_{i+1,j} - M_{i,j}| + |M_{i,j+1} - M_{i,j}|$$

To convert from image of size $m \times n$ to a column vector, the resulting L_1 matrix becomes size $L_1 \in \mathbb{R}^{(mn-m-n+1)\times(mn)}$. The L_1 matrix below is an example for an image of 3×3 pixels to assist in visualizing the pattern.

$$M \in \mathbb{R}^{3\times 3},\ \ \boldsymbol{m} \in \mathbb{R}^9$$

$$L_1 = \begin{bmatrix} -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2 & 1 & 0 & 1 & 0 \end{bmatrix}$$

This pattern was followed for the size of the provided image in MATLAB®, again using a sparse matrix to avoid using a large amount of memory storage.

```
% Create L Matrix
L = -2 * speye(p^2, p^2 + p);
I_1 = [sparse(p^2,1), speye(p^2, p^2 + p - 1)];
I_2 = [sparse(p^2,p), speye(p^2)];
L = L + I_1 + I_2;
L = L(1:p^2, 1:p^2);
rowsToRemove = mod(1:p^2, p) == 0;
L(rowsToRemove,:) = [];
```

Figure 8: MATLAB ® L-Matrix Formation

The provided library function `admml1reg()` was used to carry out the regularization. As a test, three values of α were used, $\alpha_1 = 0.1$, $\alpha_2 = 1$, $\alpha_3 = 10$.

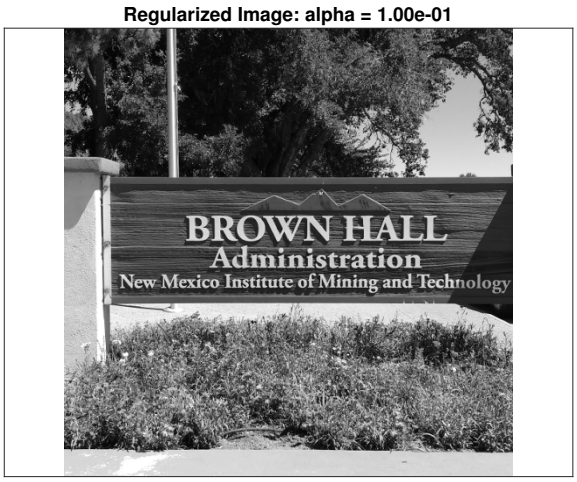

Figure 9: Repaired Image - $\alpha = 0.1$

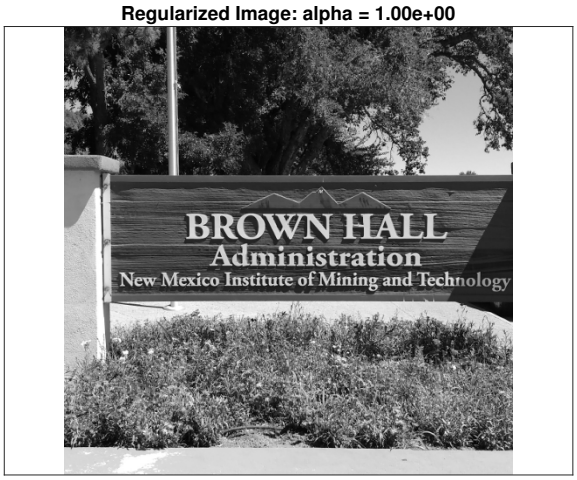

Figure 10: Repaired Image - $\alpha = 1$

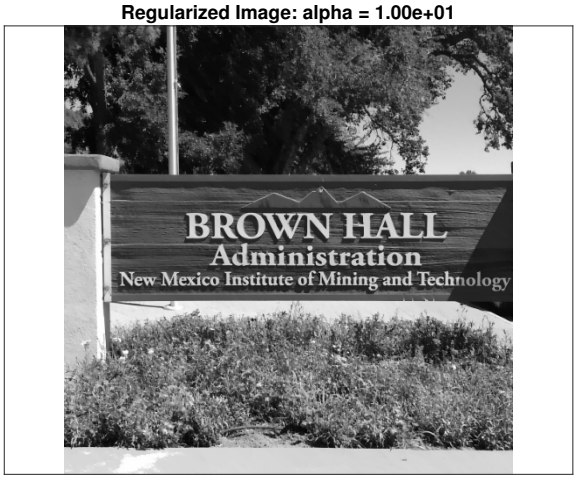

Figure 11: Repaired Image - $\alpha = 10$

For each value of alpha that was tested, it seems that the regularization was successful. Each run took approximately 10 minutes of computation time, so I decided to not test outside of these three cases.