Image Processing Algorithms: Theory and Implementation[*]

Lee Wei Xuan (EID: wl22963)

December 5, 2024

Contents

[*]Written for a course project of M 348: Scientific Computation in Numerical Analysis at the University of Texas at Austin.

1 Image Transformation

1.1 Increasing Image Contrast

Our first task here is to increase the contrast of the image. The idea is to apply a hyperbolic tangent function to the entries of the image matrix. We shall use the following function to do this:

$$f(x, \alpha) = \frac{255}{2} \cdot \tanh\left(\alpha(x - 128)\right) + 128$$

where x is the pixel value and α is a intensity of the contrast.

The reason why we are using this function is because it is a smooth function that is bounded between 0 and 256 and it increases the pixel value x for $x > 128$ and decreases the pixel value for $x < 128$. As the intensity α increases, the change becomes more drastic when compared to the original value. This can be seen in the following figure:

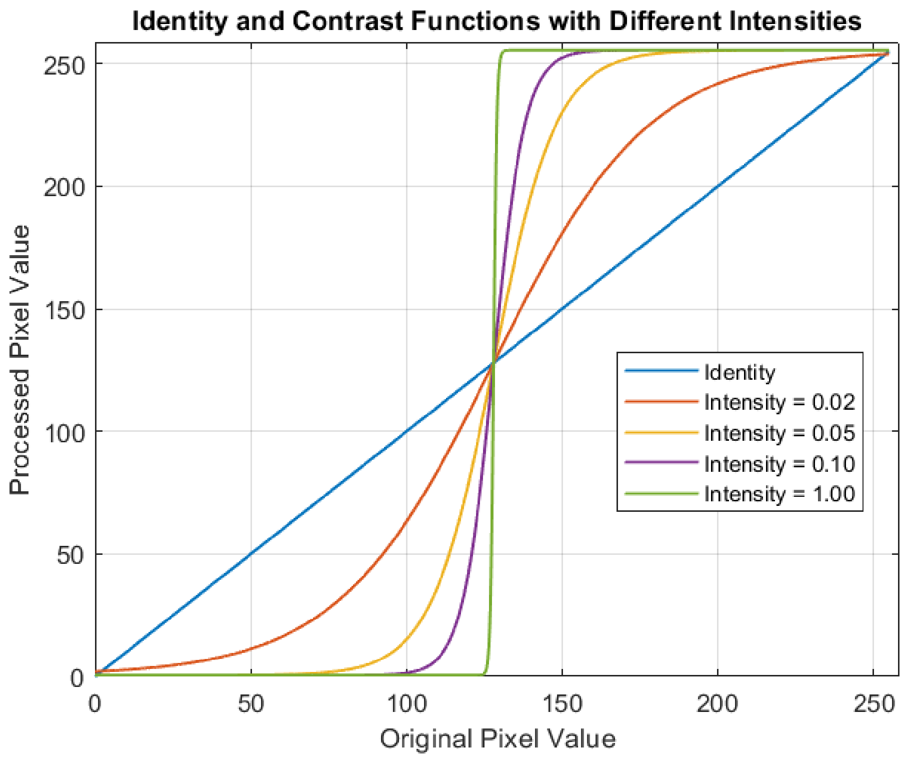

Figure 1: Identity and Contrast Functions with Different Intensities.

Thus, we define this function as `contrast.m` with the following code:

```
% Adjusts contrast with hyperbolic tangent function
% Input: original pixel value x,
%        contrast intensity alpha
% Output: pixel value after contrast adjustment
function f = contrast(x, alpha)
        f = (255/2)*tanh(alpha * (x-128)) + 128;
end
```

Listing 1: Code for `contrast` Function

After identifying the function that we want to use, we now focus on approximating it on the interval $[0, 255]$ with a degree 10 polynomial.

First, we shall introduce the function `nest` which uses Horner's method for efficient polynomial evaluation. The code can be found in the book by Sauer[2].

```
%From Sauer's Book: Program 0.1
%Nested multiplication evaluates polynomial
%from nested form using Horner's method
%Input:  degree d of polynomial,
%        array of d+1 coefficients c
%        which starts with constant first,
%        x-coordinate x at which to evaluate, and
%        array of d base points b, if needed
%Output: value y of polynomial at x
function y=nest(d,c,x,b)
if nargin<4
    b=zeros(d,1);
end
y=c(d+1);
for i=d:-1:1
  y = y.*(x-b(i))+c(i);
end
```

Listing 2: Code for `nest` Function

Next, we introduce the function `newtdd` for obtaining the coefficients of the interpolating polynomial through Newton's Divided Difference. The code can also be found in the book by Sauer[2].

```
% From Sauer's Book: Program 3.1
% Newton's Divided Difference Interpolation Method
% Computes coefficients of interpolating polynomial
% Input:  x and y are vectors containing the
%         x and y coordinates of the n data points
% Output: coefficients c of interpolating
%         polynomial in nested form
% Use nest.m to evaluate interpolating polynomial
function c=newtdd(x,y,n)
    v = zeros(n,n);    % v is the Newton triangle
    c = zeros(n,1);
    for j=1:n          % Fill in the 1st
        v(j,1)=y(j);   % column of Newton triangle
    end
    for i=2:n          % For column i, fill in
        for j=1:n+1-i  % column from top to bottom
            v(j,i)=(v(j+1,i-1)-v(j,i-1))...
                    /(x(j+i-1)-x(j));
        end
    end
    for i=1:n
        c(i)=v(1,i);   % Read along top of triangle
    end                % for output coefficients
end
```

Listing 3: Code for `newtdd` Function

Lastly, note that since we are trying to interpolate the `contrast` function on $[0, 255]$, our Chebyshev nodes are given by:

$$x_i = \frac{255}{2} + \frac{255}{2}\cos\left(\frac{2i-1}{2n}\pi\right) = \frac{255}{2}\left[1 + \cos\left(\frac{2i-1}{2n}\pi\right)\right]$$

where $i = 1, 2, \ldots, 11$ and $n = 11$. We now present the code for increasing the image contrast with intensities $= 1, 0.02$, where the contrast function is interpolated with a degree 10 polynomial:

```matlab
% Convert entries to double to avoid overflow
A = double(imread('sunflower.jpeg'));
% Adjust image contrast with different intensities
for intensity = [1, 0.02]
    % 11 points required for degree 10
    % interpolating polynomial
    i = 1:11;
    % Base points for Chebyshev interpolation
    base_points = 255/2*(1+cos((2*i-1)*pi/(2*11)));
    values = contrast(base_points(i), intensity);
    % Coefficients of interpolating polynomial
    c = newtdd(base_points, values, 11);
    % Interpolating polynomial
    interpolate = @(x) nest(10, c, x, base_points);
    % Initialize matrix for new image
    A_new = zeros(size(A));
    for i = 1:853
       for j = 1:640
           for k = 1:3
            A_new(i,j,k) = interpolate(A(i,j,k));
           end
       end
    end
    % Produce new image with adjusted contrast
    figure
    % Convert entries to uint8 to display image
    A_new = uint8(A_new);
    image(A_new)
end
```

Listing 4: Code for Increasing Image Contrast

The results are as follows:

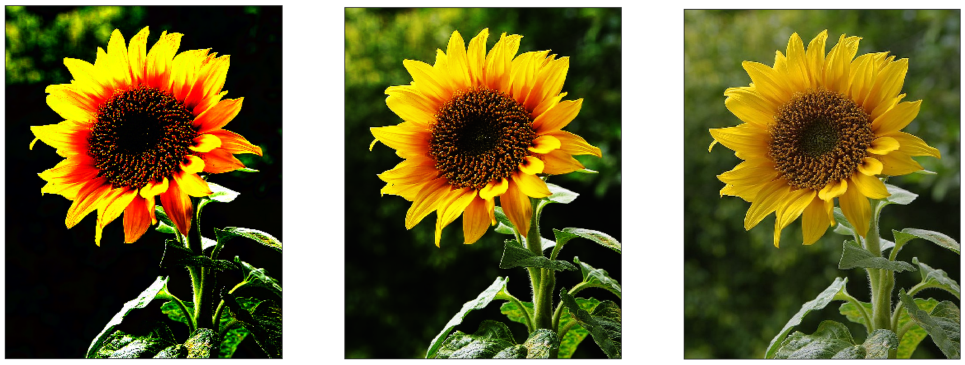

Figure 2: (Left to Right) Contrast intensity $= 1, 0.02$ and Original Image

Below we also compare the interpolating polynomial of `contrast` functions with different intensities. As we can see from the figure below, the interpolating polynomial of the contrast function with intensity $= 0.02$ is very close to the original contrast function. However, the interpolating polynomial of the contrast function with intensity $= 1$ is not as close to the original contrast function. Therefore as we increase the intensity of the contrast function, the interpolating polynomial becomes less accurate and the improvement in contrast is less noticeable.

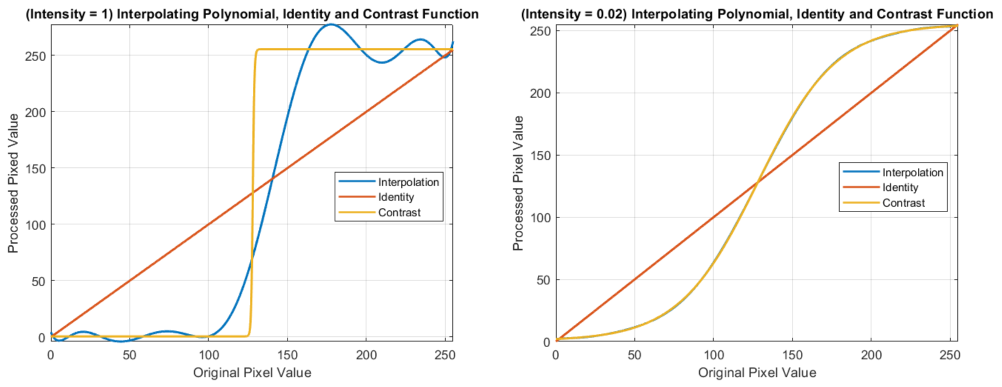

Figure 3: Interpolating Polynomials, Identity and Contrast Functions.

1.2 Rotating the Image

Now we are tasked to rotate the image by $-40°$ (clockwise). We start with a 0 matrix (called `canvas` in our code) with dimension $1200 \times 1200 \times 3$, which we call C. For each color channel, we want to figure out what the (i, j)-th of C should be. Thus, we want a map $T : \{1, \dots, 1200\} \times \{1, \dots, 1200\} \to \{0, \cdots, 255\}$ that tells up the (i, j)-th entry of C (for fixed color channel) should be $T(i, j)$. To simplify our illustration, we now think of C, A as 2D matrices and momentarily ignore the color channels. When implementing, all we have to do is apply the methods described below to each channel.

First, the rotation is about the center. Thus the central entry of C is equal to the central entry of A, which means $T(600, 600) = A(437, 320)$. Next, if we rotate C around its center by $+40°$, it should look like A, with some white padding around it. Thus, let $i_{\text{roc}}, j_{\text{roc}}$ be the position of i, j after a $+40°$ rotation, then $T(i, j) = A(i_{\text{roc}}, j_{\text{roc}})$. However, it might be the case that $i_{\text{roc}}, j_{\text{roc}}$ are not within the bounds of entries of A. In this case, we know that it is a white pixel. Lastly, $i_{\text{roc}}, j_{\text{roc}}$ might not be integers, thus we just pick the nearest integer. We first show the result, then the algorithm.

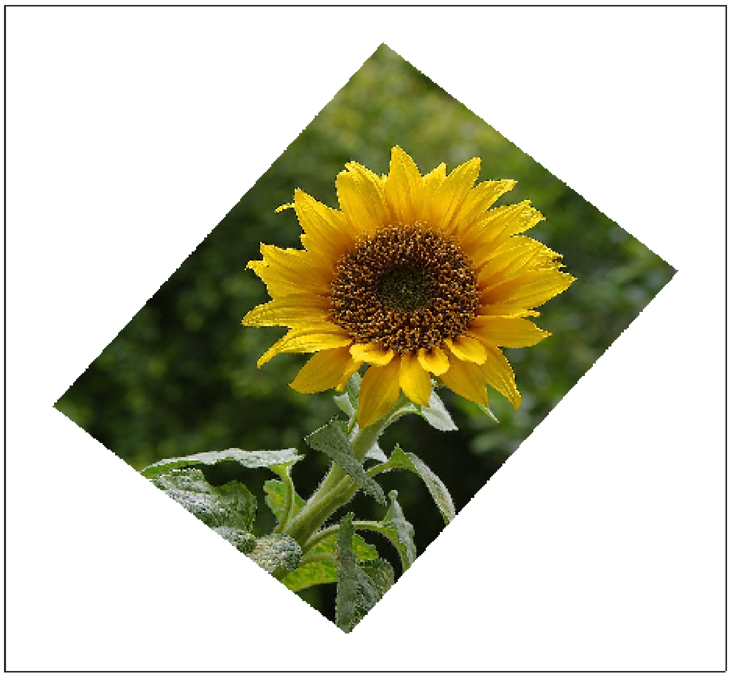

Figure 4: Sunflower Rotated 40° Clockwise.

```
A = double(imread('sunflower.jpeg'));
canvas = zeros(1200, 1200, 3);
for c = 1:3
    for i = 1:1200
        for j = 1:1200
            % Find the 40 degree
            % (counter-clockwise) rotation
            % for the (i,j)-th entry
            % int32 to round to nearest integer
            i_rot = int32(cosd(40)*(i - 600) - ...
                        sind(40)*(j - 600))+ 427;
            j_rot = int32(sind(40)*(i - 600) + ...
                        cosd(40)*(j - 600))+ 320;
            % Check if out of bounds
            if i_rot < 1 || ...
               i_rot > size(A, 1) || ...
               j_rot < 1 || ...
               j_rot > size(A, 2)
               % Set to white
                canvas(i, j, c) = 255;
            else
                canvas(i, j, c) = ...
                A(i_rot, j_rot, c);
            end
        end
    end
end
image(uint8(canvas))
```

Listing 5: Code for Rotating the Image

2 Image Compression

2.1 Power Method

We first introduce the function `normalize` which normalizes a vector.

```matlab
% Normalize a vector
function normalized_vector = normalize(vector)
    normalized_vector = vector/norm(vector);
end
```

Listing 6: Code for `normalize` Function

The power method is used to find the *dominant* eigenvector, eigenvalue pair of a matrix. We first show how the power method (`pwr_md`) works and then explain why it works. The algorithm is as follows:

```matlab
% Finding the eigenvector and DOMINANT eigenvalue
% of a matrix using the power method
% Input: Matrix A,
%        tol for stopping criteion,
%        max_ite for maximum number of iterations
% Output: eigenvector egvc and eigenvalue egvl
function [egvc, egvl] = pwr_md(A, tol, max_ite)
    n = size(A, 1);
    vc = rand(n, 1); % Initialization
    vc = normalize(vc);
    for i = 1:max_ite
        vc_temp = A * vc;
        vc_temp = normalize(vc_temp);
        if norm(vc_temp - vc) < tol
            vc = vc_temp;
            break;
        end
        vc = vc_temp;
    end
    egvc = vc;
    egvl = vc' * A * vc;
end
```

Listing 7: Code for `pwr_md` Function

The idea here is that if we start with a random vector x_0 then $A^n x_0$ converges to a dominant eigenvector x (we choose to let it be a unit vector) as n tends to infinity. Moreover, the corresponding eigenvalue λ is obtained by calculating $x^T Ax$ because $x^T Ax = x^T(\lambda x) = \lambda(x^T x) = \lambda$.

The reason why the power method works for our problem is because of the following theorem (the power method is called *power iteration* here) that can be found in the book by Sauer[2] (p.560, Theorem 12.2):

Theorem. *Let A be an $m \times m$ matrix with real eigenvalues $\lambda_1, \ldots, \lambda_m$ satisfying $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_m|$. Assume that the eigenvectors of A span $\mathbb{R}^m$. For almost every initial vector, Power Iteration converges linearly to an eigenvector associated with λ_1 with convergence rate constant $S = \left|\frac{\lambda_2}{\lambda_1}\right|$.*

And since we are dealing with $A^T A$, which is a 640×640 real symmetric matrix (note that A is a 853×640 matrix for each color channel), we know that it is (orthogonally) diagonalizable and hence the eigenvectors span $\mathbb{R}^{640}$. Therefore, the power method works for our problem. And we see that the rate of convergence is the ratio of the second largest eigenvalue to the largest eigenvalue. We shall find the largest eigenvalue now with the power method and find the second largest next with the inverse power method.

Note that since here we have three color channels, we apply the power method three times to each of them. The code is as follows:

```
A = double(imread('sunflower.jpeg'));
egvc1 = zeros(640,3); % Largest eigenvectors
egvl1 = zeros(1,3); % Largest eigenvalues
tol = 1e-6; max_ite = 1000;
for c = 1:3 % For each color channel
    cA = A(:, :, c);
    cAt_cA = cA' * cA;
    [egvc1(:, c), egvl1(:, c)] = ...
    pwr_md(cAt_cA, tol, max_ite);
end
```

Listing 8: Code for Computing Largest Eigenvector, Eigenvalue Pair

The eigenvalues for red: 6.3075×10^9, blue: 5.2808×10^9, green: 5.5189×10^8. The corresponding eigenvectors are the columns of the matrix `egvc1`.

2.2 Inverse Power Method

Now, we deal with finding the second largest eigenvalue and its corresponding eigenvector. We shall use the inverse power method to do this. Here is the idea, let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of A. Pick a number μ, to find the eigenvalue of A, that is closest to μ, we apply the power method on the matrix $(A - \mu I)^{-1}$. This works because now the eigenvalues are of the form $\frac{1}{\lambda_i - \mu}$ and the eigenvalue that is closest to μ with the largest in magnitude.

Instead of computing the inverse matrix directly to compute $(A^T A - \mu I)^{-1} x$, we solve for y in the equation $(A^T A - \mu I)y = x$. To do this, one might consider using the Gauss-Seidel or SOR method. However, they are not guaranteed to converge because we do not know a priori that the matrix $(A^T A - \mu I)$ is diagonally dominant. Moreover, although $(A^T A - \mu I)$ is symmetric, it is not positive definite. This is because the eigenvalues of the matrix subtracting μI are precisely the original eigenvalues subtracting μ. Therefore, if μ is greater than the smallest eigenvalue, then we would have negative eigenvalues, and it is not positive definite

Thus, we shall use the $PA = LU$ factorization method together with substitutions to solve the system of equations and achieve the effect of multiplying a vector by the inverse of a matrix. We define the functions `backsub` and `fwdsub` to do the back substitution and forward substitution respectively and then use them in the function `LU_solver`. The codes are as follows:

```matlab
% Backward substitution method
% Input: Upper triangular matrix U, vector b
% Output: Solution vector x
function x = backsub(U, b)
    n = size(U, 1);
    x = zeros(n, 1);
    for i = n:-1:1
        x(i) = b(i);
        for j = i+1:n
            x(i) = x(i) - U(i, j) * x(j);
        end
        x(i) = x(i) / U(i, i);
    end
end
```

Listing 9: Code for Backward Substitution

```matlab
% Forward substitution method
% Input: Lower triangular matrix L, vector b
% Output: Solution vector x
function x = fwdsub(L, b)
    n = size(L, 1);
    x = zeros(n, 1);
    for i = 1 : n
        sum = b(i, 1);
        for j = 1 : i-1
            sum = sum - L(i, j) * x(j, 1);
        end
        x(i, 1) = sum / L(i, i);
    end
end
```

Listing 10: Code for Forward Substitution

```matlab
% Function to solve a system of
% linear equations using LU decomposition
% Input: matrix A, vector b
% Output: solution vector x
function x = LU_solver(A, b)
    [L, U, P] = lu(A); % Predefined in MATLAB
    y = fwdsub(L, P * b);
    x = backsub(U, y);
end
```

Listing 11: Code for Solving System of Equations with LU Factorization

Now, we implement the inverse power method, which we call it `i_pwr`. The code is as follows:

```matlab
% Finding the eigenvalue closest to
% a given value mu and its eigenvector
% Input: Matrix A,
%        tol for stopping criteion,
%        max_ite for maximum number of iterations
%        mu for the given value
% Output: eigenvector egvc and eigenvalue egvl
```

```
function [egvc, egvl] = i_pwr(A, tol, max_ite, mu)
    n = size(A, 1);
    I = eye(n);
    vc = rand(n, 1); % Initialization
    vc = normalize(vc);
    for i = 1:max_ite
        % Use LU_solver to solve
        % (A - mu * I) * vc_temp = vc, to get
        % vc_temp = (A - mu * I)^(-1) * vc
        vc_temp = LU_solver(A - mu * I, vc);
        vc_temp = normalize(vc_temp);
        if norm(vc_temp - vc) < tol
            vc = vc_temp;
            break;
        end
        vc = vc_temp;
    end
    egvc = vc;
    egvl = 1/(vc' * LU_solver(A - mu*I, vc)) + mu;
end
```

Listing 12: Code for Inverse Power Method

To find the second largest eigenvalue, by trial and error, we find setting μ to be half of the largest eigenvalue gives us the result that we want.

```
A = double(imread('sunflower.jpeg'));
egvc2 = zeros(640,3); % 2nd largest eigenvectors
egvl2 = zeros(1,3); % 2nd largest eigenvalues
tol = 1e-6; max_ite = 1000;
for c = 1:3 % For each color channel
    cA = A(:, :, c);
    cAt_cA = cA' * cA;
    [egvc2(:, c), egvl2(:, c)] = ...
    i_pwr(cAt_cA, tol, max_ite, egvl1(:, c) / 2);
end
```

Listing 13: Code for Computing Second Largest Eigenvector, Eigenvalue Pair

The eigenvalues for red: 3.9003×10^8, blue: 2.8642×10^8, green: 9.7986×10^7. The corresponding eigenvectors are the columns of the matrix `egvc2`.

2.3 SVD Compression

We first show the following result:

Theorem. $A = \sum_{i=1}^n \sqrt{\lambda_i} u_i v_i^T$ *where* λ_i *is the* i*-th largest eigenvalue of* $A^T A, v_i$ *is its corresponding unit eigenvector, and* $u_i = \frac{Av_i}{\|Av_i\|_2}$.

Proof. Let $A = U\Sigma V^T$ be the SVD of A and Σ be the diagonal matrix with the singular values arranged in descending order. Then the i-th column of U, V are u_i, v_i. Now, we have

$$\begin{aligned} A &= \begin{pmatrix} u_1 \cdots u_n \end{pmatrix} \begin{pmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_n} \end{pmatrix} \begin{pmatrix} v_1^T \\ \vdots \\ v_n^T \end{pmatrix} \\ &= \begin{pmatrix} u_1 \cdots u_n \end{pmatrix} \begin{pmatrix} \sqrt{\lambda_1} v_1^T \\ \vdots \\ \sqrt{\lambda_n} v_n^T \end{pmatrix} \\ &= \sum_{i=1}^n \sqrt{\lambda_i} u_i v_i^T. \end{aligned}$$

□

Now, our task is to find all the eigenvalues. To do this we use the power method together with the deflation method which can be found in Lecture Notes of Wong[3]. The main result that we will be using is

Theorem. *Let* A *be a* $n \times n$ *matrix with non-zero eigenvalues* $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$ *and corresponding unit eigenvectors* $v_1, \cdots, v_n$. *Then the deflated matrix* $B = A - \lambda_1 v_1 v_1^T$ *has eigenvalues* $0, \lambda_2, \cdots, \lambda_n$ *with eigenvectors* $v_1, w_2, \cdots, w_n$, *where*

$$w_i = v_i - \left(\frac{\lambda_1}{\lambda_i} v_1 v_1^T \right) v_i. \qquad (i = 2, \cdots, n)$$

Proof. $Bv_1 = Av_1 - \lambda_1 v_1 (v_1 v_1^T) = \lambda_1 v_1 - \lambda_1 v_1 = 0$, for $i \geq 2$, we have

$$\begin{aligned} Bw_i &= Av_i - \frac{\lambda_1}{\lambda_i}(Av_1) v_1^T v_i - \lambda_1 v_1 v_1^T v_i + \frac{\lambda_1^2}{\lambda_i} v_1 (v_1^T v_1) v_1^T v_i \\ &= \lambda_i v_i - \lambda_1 v_1 v_1^T v_i - \frac{\lambda_1}{\lambda_i}(\lambda_1 v_1) v_1^T v_i + \frac{\lambda_1^2}{\lambda_i} v_1 (1) v_1^T v_i \\ &= \lambda_i w_i. \end{aligned}$$

□

Suppose we already know the greatest eigenvalue λ_1 and its corresponding eigenvector v_1. Then let $B = A - \lambda_1 v_1 v_1^T$. Now, the greatest eigenvalue is then $lambda_2$ and we can find it by applying the power method.Note this only gives us w_2, to recover v_2 (call it `corrected_egvc` in our code) we solve

$$\underbrace{\left(I - \frac{\lambda_1}{\lambda_2} v_1 v_1^T\right)}_{\text{call it } \texttt{aux} \text{ in our code}} v_2 = w_2.$$

To find the rest of the eigenvalues, we repeat the process and now subtract $\lambda_2 w_2 w_2^T$ from B. Note that it is w_2 here. And we will get λ_3. Suppose we get the eigenvector w_3', to recover v_3, we solve

$$\underbrace{\left(I - \frac{\lambda_2}{\lambda_3} w_2 w_2^T\right)\left(I - \frac{\lambda_1}{\lambda_3} v_1 v_1^T\right)}_{\text{call it } \texttt{aux} \text{ in our code}} v_3 = w_3'.$$

We can do this repeatedly to get all the eigenvalues and eigenvectors. However, we are aware that computing the matrix `aux` becomes more and more expensive as we try to find more eigenvalues and eigenvectors. In fact, it grows cubically. Therefore we will only be computing the first K eigenvalues and eigenvectors. Now, we first introduce the function `K_eg` which computes the K greatest eigenvectors and eigenvalues of a matrix.

```
% Finding K greatest eigenvectors, eigenvalues
% of a matrix using the power method
% Input:  matrix A,
%         tol for stopping criteion,
%         max_ite for maximum number of iterations
%         K for number of eigenvectors to find
% Output: eigenvectors egvcs and eigenvalues egvls
%         egvcs(: i) is the eigenvector
%         corresponding to the eigenvalue egvls(i)
function [egvcs, egvls] = K_eg(A, tol, max_ite, K)
    n = size(A, 1);
    egvcs = zeros(n, K);
    egvls = zeros(K,1);
    pre_egvl = 0;
    pre_egvc = zeros(n, 1);
    I = eye(n);
    % Dominant eigenvectors of deflated matrix
    pre_egvcs = zeros(n, n);
    for i = 1:K
        % Transformation makes dominant
        % eigenvalue of preious matrix to be zero
        A = A - ...
            pre_egvl * (pre_egvc * pre_egvc');
        % Use Power Method to find the dominant
        % eigenvalue and eigenvector
        [cur_egvc, cur_egvl] = ...
        pwr_md(A, tol, max_ite);
        % Aux matrix for equation
        % to find corrected eigenvector
        aux = I;
        for j = 1:i-1
            aux = (I - egvls(j)/cur_egvl * ...
        pre_egvcs(:, j) * pre_egvcs(:, j)')*aux;
        end
        % Recover the original eigenvector
        corrected_egvc = LU_solver(aux, cur_egvc);
        egvcs(:, i) = normalize(corrected_egvc);
        egvls(i) = cur_egvl;
        % Update eigenvectors of deflated matrix
        pre_egvcs(:, i) = cur_egvc;
        pre_egvc = cur_egvc;
        pre_egvl = cur_egvl;
    end
end
```

Listing 14: Code for Computing the K Greatest Eigenvectors, Eigenvalues

We now carry out the process of approximating the matrix A using the $\sum_{i=1}^{n} \sqrt{\lambda_i} u_i v_i^T$. However due to constraints in computational power and time, we will only be adding the first K terms and we have decided to use $K = 200$. We apply this to each color channel. The code is as follows:

```matlab
A = double(imread('sunflower.jpeg'));
K = 200; % Number of eigenvectors to find
egvcs = zeros(640,K,3); % For the eigenvectors
egvls = zeros(K,3); % For the eigenvalues
U = zeros(853,K,3); % For the u_i's
loss = zeros(K,1); % Loss of image quality
A_cmprs = zeros(853,640,3); % Compressed image
tol = 1e-6; max_ite = 1000;
for c = 1:3 % For each color channel
    cA = A(:, : , c);
    cAt_cA = cA' * cA;
    [egvcs(:, :, c), egvls(:,c)] = ...
    eg_vc_vl(cAt_cA, tol, max_ite, K);
    for i = 1:K % Compute the u_i's
        U(:, i, c) = ...
        normalize(cA * egvcs(:, i, c));
    end
end
for k = 1:K
    cur_loss = 0;
    for c = 1:3
        A_cmprs(:, :, c) = ...
        A_cmprs(:, :, c) + sqrt(egvls(k,c)) * ...
        U(:, k, c) * egvcs(:, k, c)';
        cur_loss = cur_loss + ...
        norm(A(:, :, c) - A_cmprs(:, :, c));
    end
    if ismember(k, [5, 10, 20, 40, 80, 160])
        figure
        image(uint8(A_cmprs))
    end
    loss(k) = cur_loss;
end
figure
semilogy(1:K, loss)
```

Listing 15: Code for Image Compression

Also, we have tracked the loss in the image quality as K increases and it is defined to be

$$\sum_{\text{color}=r,g,b} \|A^{(\text{color})} - A^{(\text{color})}_{\text{compressed},K}\|_2$$

where

$$A_{\text{compressed},K} = \sum_{i=1}^{k} \sqrt{\lambda_i} u_i v_i^T.$$

We first show the loss of image quality as K increases:

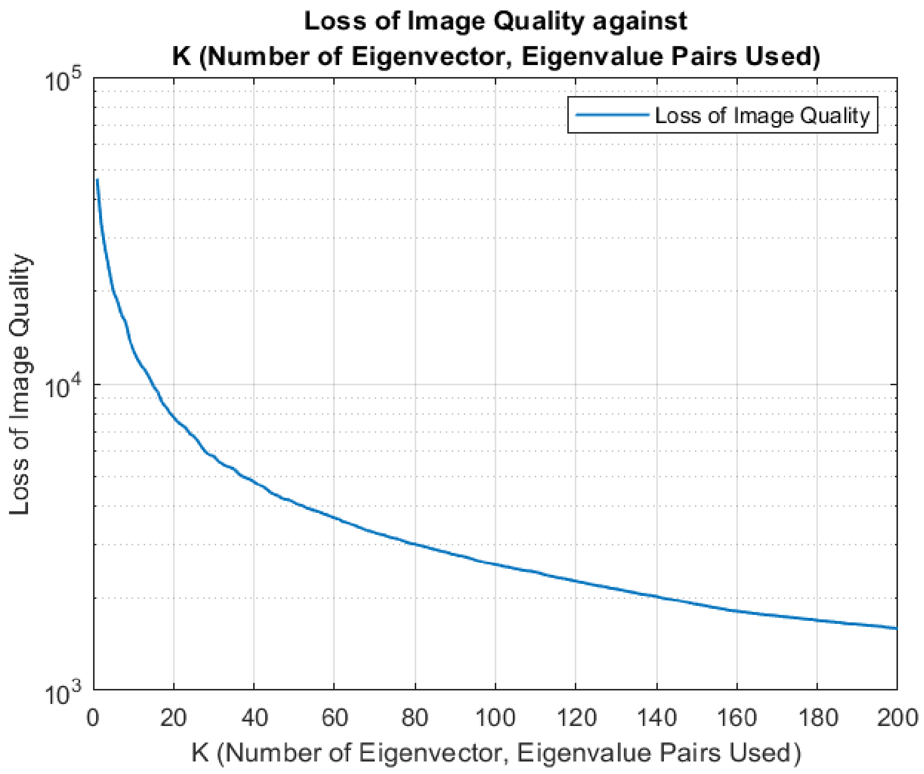

Figure 5: Loss of Image Quality against K.

And we can see a steady decrease in loss as K increases. Now, we see the actual results of the compression:

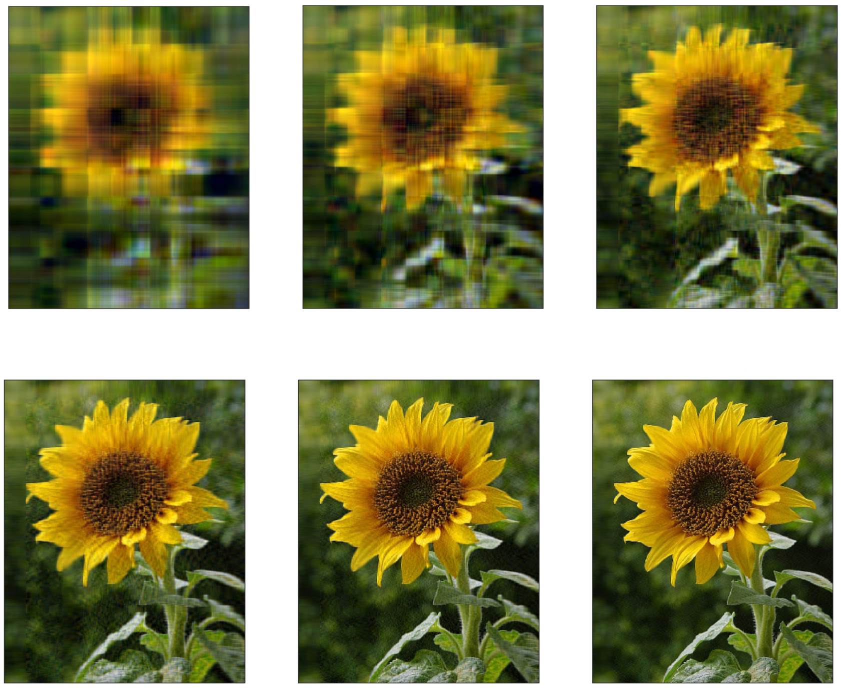

Figure 6: Compressed Image for $K = 5, 10, 20, 40, 80, 160$.

We think that at $K = 20$, the image already resembles a sunflower and this is a remarkable compression of information. The original image requires $640 \times 853 \times 3 = 1,637,760$ entries while the compressed image at $K = 20$ only needs 20 singular values and $u_i, v_i, i = 1, \dots, 20$ for each color channel. And each u_i, v_i requires 640 entries. Thus, the compressed image requires $20 + 20 \times 640 \times 3 \times 2 = 76,820$ entries. The compression ratio is 21.3. Furthermore, at $K = 80$, we would say that the image is of good quality, and further calculation shows that the compression ratio is 5.3.

2.4 Brief Note on SVD Compression

One reason why truncating the SVD is a good idea for image compression is because $A_k = \sum_{i=1}^{k} \sqrt{\lambda_i} u_i v_i^T$ is matrix of rank at most k that best approximates A in the 2-norm. See Theorem 4.9 of the lecture notes by Sanjeev Arora[1] for the proof. Therefore as K increases, we have more "freedom" when it comes to approximating the A, and thus we can expect the error (in 2-norm) to steadily decrease and this makes it a good approximation.

References

[1] Sanjeev Arora and Ravi Kannan. *4 Singular Value Decomposition (SVD)*. 2012. URL: `https://www.cs.princeton.edu/courses/archive/spring12/cos598C/svdchapter.pdf`.

[2] Tim Sauer. *Numerical analysis Timothy Sauer, George Mason University*. 3rd ed. Pearson, 2019.

[3] Jeffrey Wong. *Math 361S Lecture notes, Finding eigenvalues: The power method.* 2019. URL: `https://services.math.duke.edu/~jtwong/math361-2019/lectures/Lec10eigenvalues.pdf`.