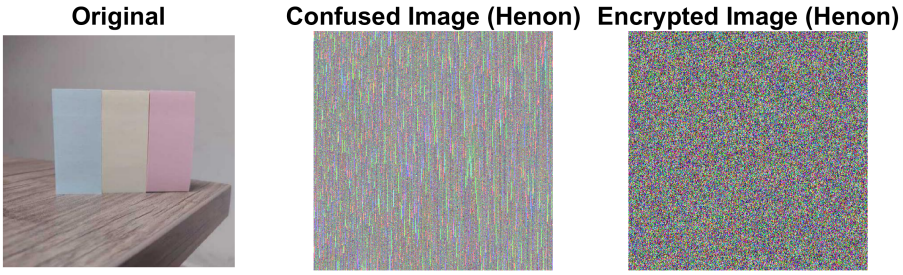
State of the art methods

Henon Method

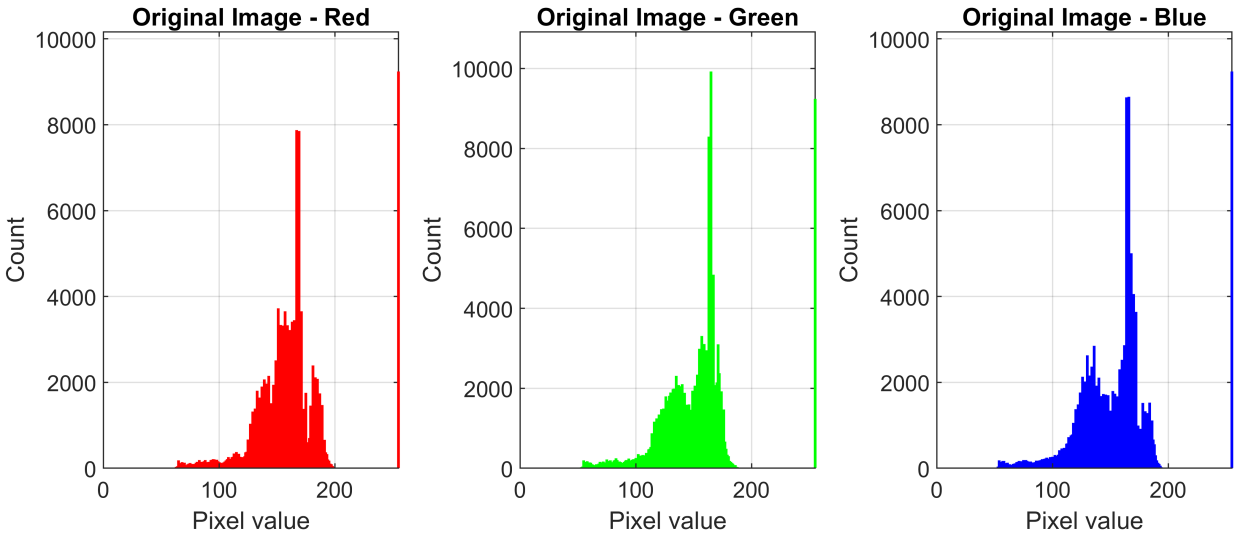
```
%% Chaotic Maps Image Encryption
clc; clear; close all;

%% HENON CHAOTIC MAP GENERATION
img =imread(' C:\Users\Control Lab\Downloads\codes\codes\state of art methods\paper.PNG');
a = 1.4;
b = 0.3;
x0 = 0.9;
y0 = 0.5;

[R, C, CH] = size(img);
N = R*C*CH;


%% Encryption Process:
% Generate Henon chaotic sequence
henon_seq = generate_henon_map(a, b, x0, y0, N);


%% CONFUSION USING HENON
% Confusion
confused_henon = confusion_stage(img, henon_seq );

%% DIFFUSION USING HENON
% Diffusion
encrypted_henon = diffusion_stage(confused_henon , henon_seq);
figure;
subplot(131), imshow(img), title('Original');
subplot(132), imshow(confused_henon), title('Confused Image (Henon)');
subplot(133), imshow(encrypted_henon), title('Encrypted Image (Henon)');
```



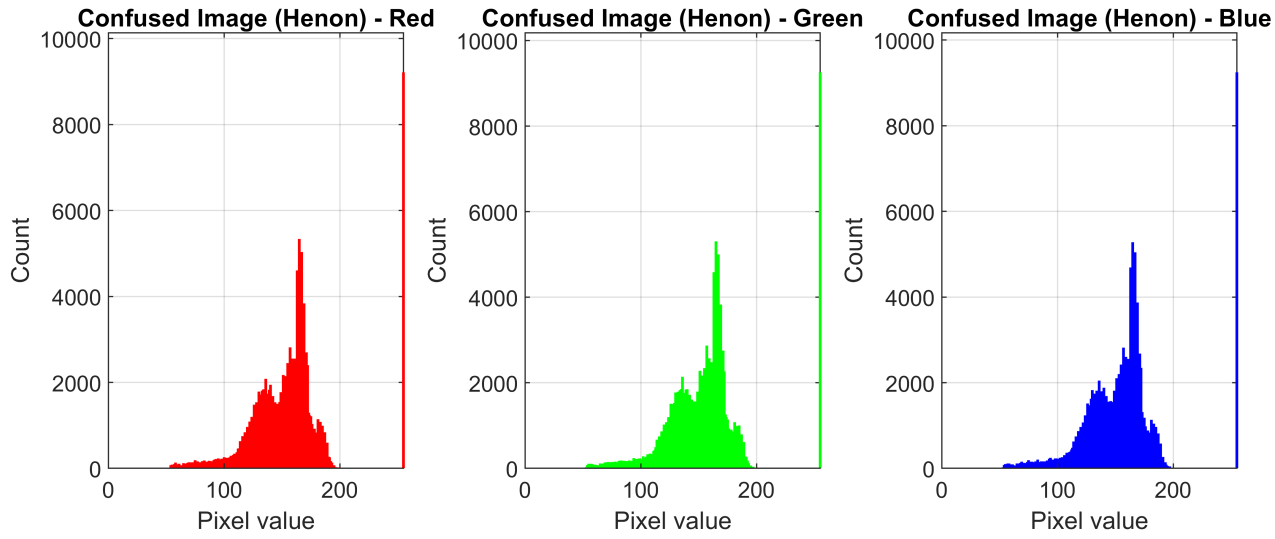

```
%% %% Histogram Analysis for Henon Map Encryption
fprintf('=== Henon Map Encryption Histogram Analysis ===\n');
```

```
=== Henon Map Encryption Histogram Analysis ===
```

```
plot_hist(img, 'Original Image');
```




```
plot_hist(confused_henon, 'Confused Image (Henon)');
```




```
plot_hist(encrypted_henon, 'Encrypted Image (Henon)');
```


```
chi_Image   = chi_square_hist(img, 'Original Image');
```

```
Chi-Square Values (Original Image):
Red Channel   : 629925.9294
Green Channel : 660452.6480
Blue Channel  : 616476.2014
```

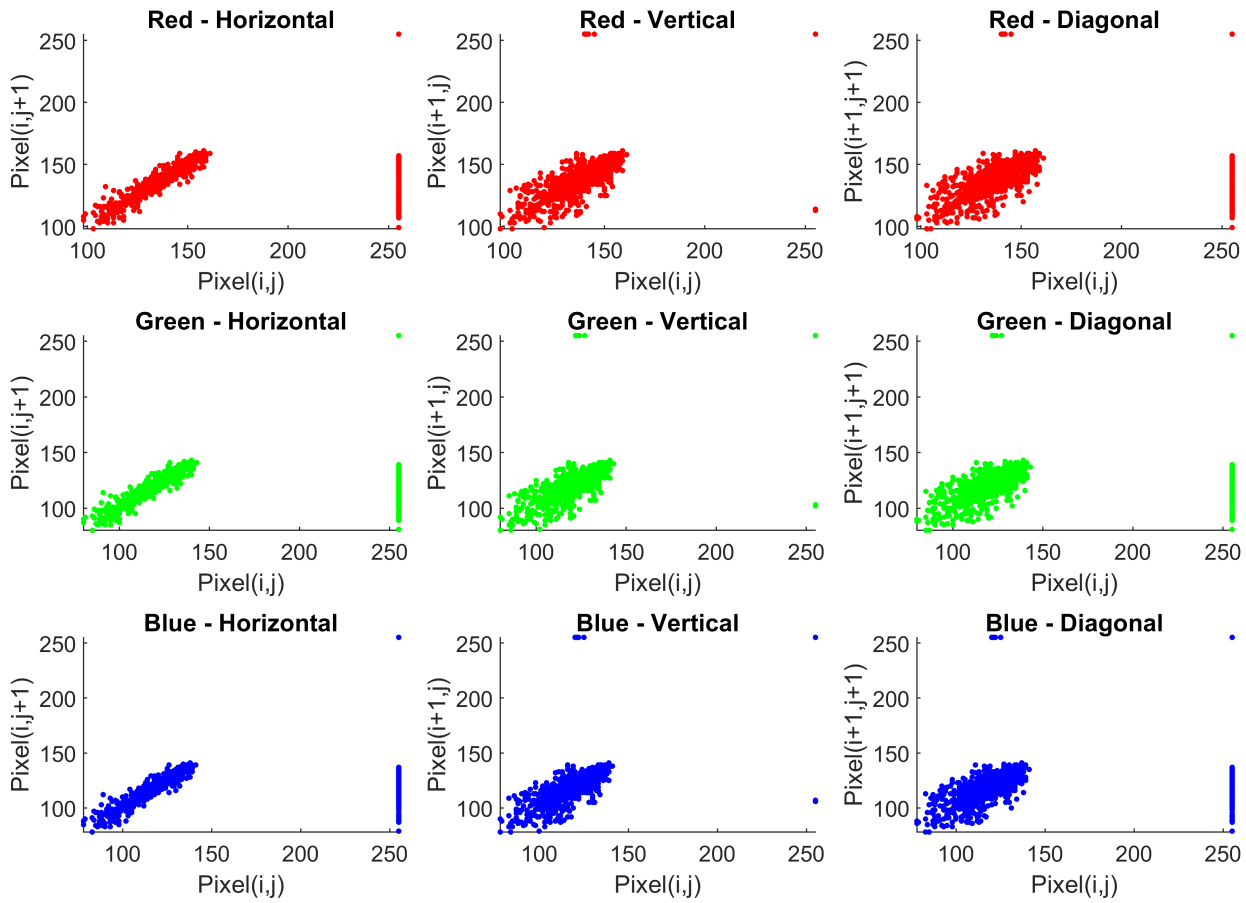
```
chi_henon    = chi_square_hist(encrypted_henon,  'Encrypted Image (Henon)');
```

```
Chi-Square Values (Encrypted Image (Henon)):
Red Channel   : 241.8850
Green Channel : 260.9341
Blue Channel  : 242.2534
```

```
%% Decryption process:
a = 1.4;
b = 0.3;
x0 = 0.9;
y0 = 0.5;
```

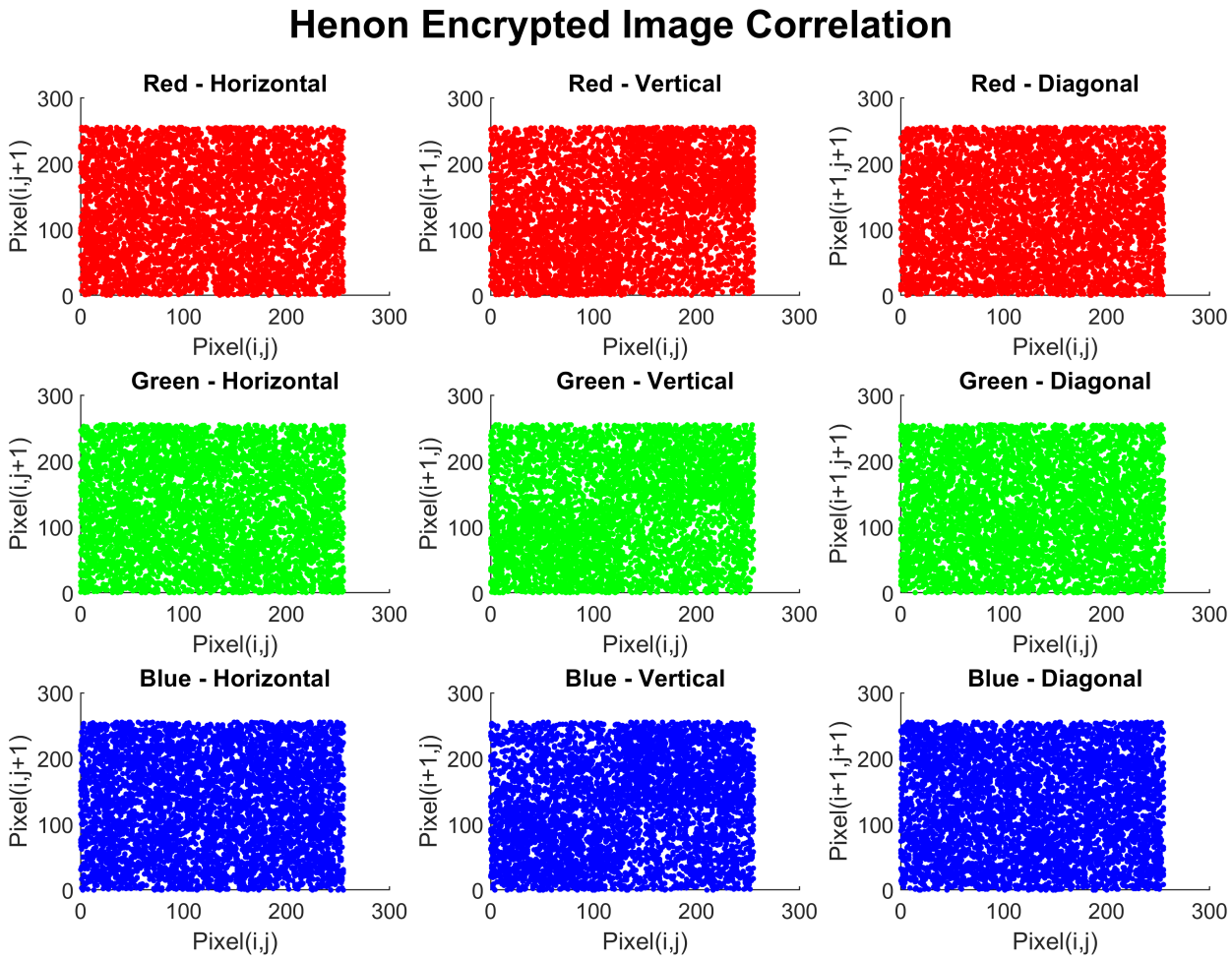
```
% Generate Henon chaotic sequence
henon_seq = generate_henon_map(a, b, x0, y0, N);
%%
% Step 1: Reverse diffusion
after_diffusion = reverse_diffusion(encrypted_henon, henon_seq );

% Step 2: Reverse confusion
decrypted_henon = reverse_confusion(after_diffusion, henon_seq );

% Display
figure;
subplot(131), imshow(encrypted_henon), title('Encrypted Image');
subplot(132), imshow(after_diffusion), title('After Reverse Diffusion');
subplot(133), imshow(decrypted_henon), title('Decrypted Image');
```


```
%% Correlation Analysis:
correlation_analysis(img, 'Original Image Correlation');
```




```
================= Correlation Summary: Original Image Correlation =================
              Horizontal     Vertical       Diagonal
-----------------------------------------------------------------
Red     :     0.9663        0.9568         0.9191
Green   :     0.9670        0.9587         0.9221
Blue    :     0.9688        0.9608         0.9263
=================================================================
```

```
correlation_analysis(encrypted_henon, 'Henon Encrypted Image Correlation');
```




```
================= Correlation Summary: Henon Encrypted Image Correlation =================
               Horizontal     Vertical       Diagonal
-----------------------------------------------------------------
Red     :      0.0010         0.1565         0.0045
Green   :      -0.0052         0.1589         -0.0045
Blue    :      -0.0052         0.1542         -0.0061
=================================================================
```

```
%% %% Sensitivity of Key for Decryption process:

%% For Henon:
% Encryption Key
a = 1.4;
b = 0.3;
x0 = 0.9;
y0 = 0.5;

[R, C, CH] = size(img);
N = R*C*CH;

% Generate Henon chaotic sequence
henon_seq = generate_henon_map(a, b, x0, y0, N);
%% CONFUSION USING HENON
```

```
confused_henon = confusion_stage(img, henon_seq );

%% DIFFUSION USING HENON
encrypted_henon = diffusion_stage(confused_henon , henon_seq);
figure;
subplot(131), imshow(img), title('Original');
subplot(132), imshow(encrypted_henon), title('Encrypted Image (Henon)');

%%  Decryption Key
a = 1.4;
b = 0.3;
x0 = 0.9;
y0 = 0.50000001;


% Generate Henon chaotic sequence
henon_seq = generate_henon_map(a, b, x0, y0, N);


% Step 1: Reverse diffusion
after_diffusion = reverse_diffusion(encrypted_henon, henon_seq );

% Step 2: Reverse confusion
decrypted_henon = reverse_confusion(after_diffusion, henon_seq );

% Display
subplot(133), imshow(decrypted_henon), title('Decrypted Image');
sgtitle('Encryption key:(0.9, 0.5) & Decryption key: (0.9, 0.50000001)')
```

Encryption key:(0.9, 0.5) & Decryption key: (0.9, 0.50000001)

```
%% %% Timing Analysis for Henon Map Encryption
fprintf('=== Henon Map Encryption Timing Analysis ===\n');
```

```
=== Henon Map Encryption Timing Analysis ===
```

```
% Start timer
tic;

% Generate Henon chaotic sequence
henon_seq = generate_henon_map(a, b, x0, y0, N);

% Confusion
confused_henon = confusion_stage(img, henon_seq );

% Diffusion
encrypted_henon = diffusion_stage(confused_henon , henon_seq);

% Stop timer
time_henon = toc;
fprintf('Henon Map Encryption Time: %.6f seconds\n', time_henon);
```

```
Henon Map Encryption Time: 0.050937 seconds
```

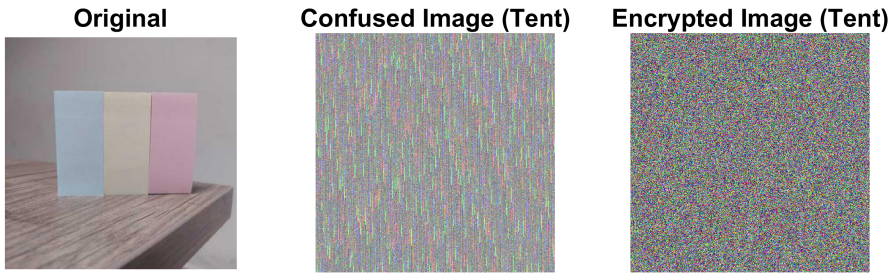
```
%% Optional: Repeat multiple times for average time
num_runs = 100;
times_henon = zeros(1,num_runs);
for k = 1:num_runs
    tic;
    henon_seq = generate_henon_map(a, b, x0, y0, N);
    confused_henon = confusion_stage(img, henon_seq );
    encrypted_henon = diffusion_stage(confused_henon , henon_seq);
    times_henon(k) = toc;
end
avg_time_henon = mean(times_henon);
fprintf('Average Henon Map Encryption Time over %d runs: %.6f seconds\n', num_runs, avg_time_he
```

```
Average Henon Map Encryption Time over 100 runs: 0.041816 seconds
```

Tent Method

```
%% Chaotic Maps Image Encryption
clc; clear; close all;
img =imread(' C:\Users\Control Lab\Downloads\codes\codes\state of art methods\paper.PNG');
[R, C, CH] = size(img);
N = R*C*CH;

%% TENT CHAOTIC MAP GENERATION
alpha = 1.99;
```

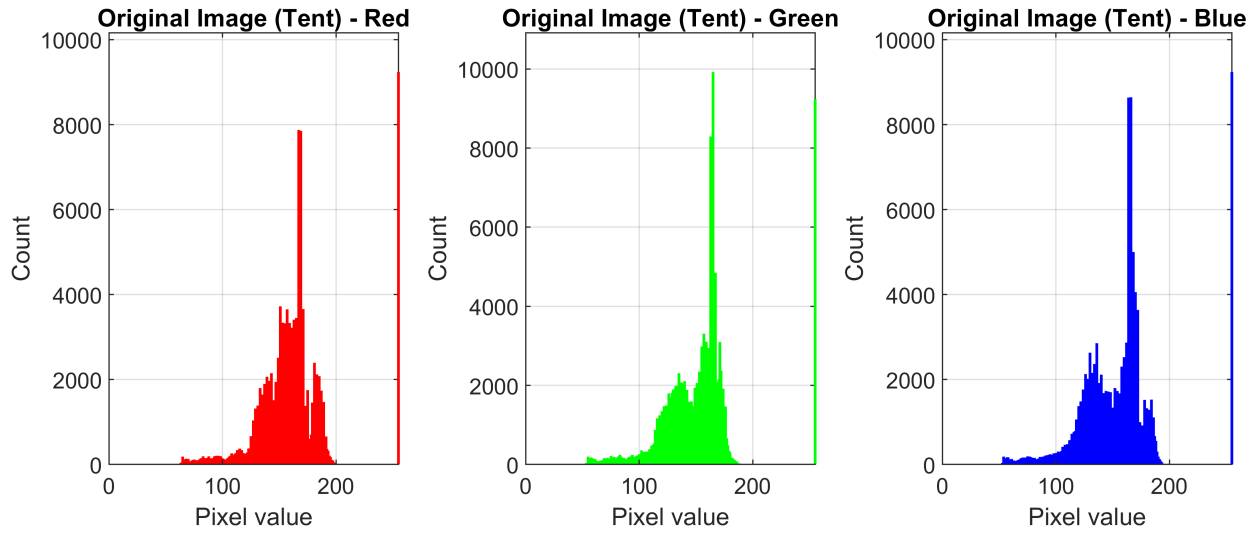
```
x0 = 0.52;

%% Encryption Process:
% Generate chaotic sequence
tent_seq = generate_tent_map(alpha, x0, N);

%% CONFUSION USING TENT
% Confusion
confused_tent = confusion_stage(img, tent_seq);

%% DIFFUSION USING TENT
% Diffusion
encrypted_tent = diffusion_stage(confused_tent, tent_seq);

figure;
subplot(131), imshow(img), title('Original');
subplot(132), imshow(confused_tent), title('Confused Image (Tent)');
subplot(133), imshow(encrypted_tent), title('Encrypted Image (Tent)');
```



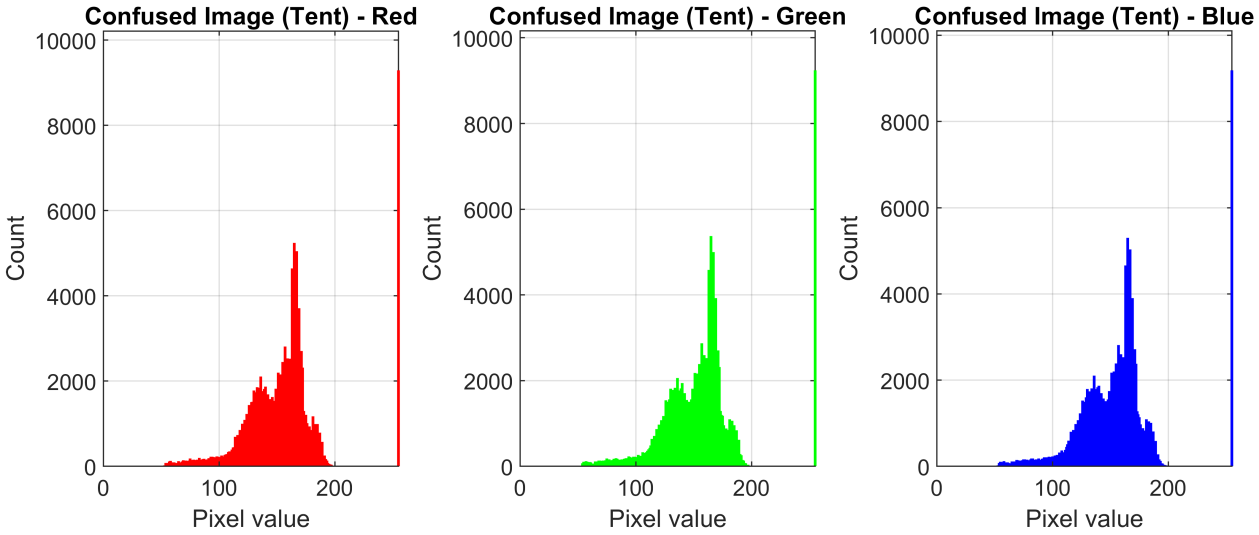

```
%% %% Histogram Analysis for Henon Map Encryption
fprintf('=== Tent Map Encryption Histogram Analysis ===\n');
```

```
=== Tent Map Encryption Histogram Analysis ===
```

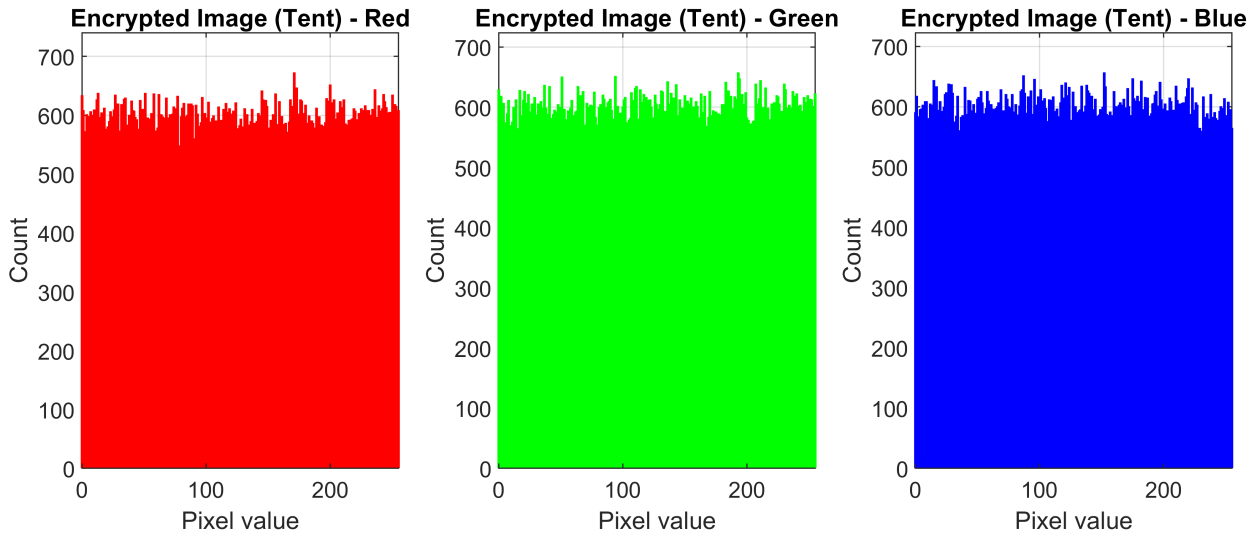
```
% Histogram Analysis
plot_hist(img, 'Original Image (Tent)');
```




```
plot_hist(confused_tent, 'Confused Image (Tent)');
```




```
plot_hist(encrypted_tent, 'Encrypted Image (Tent)');
```




```
chi_Image   = chi_square_hist(img, 'Original Image');
```

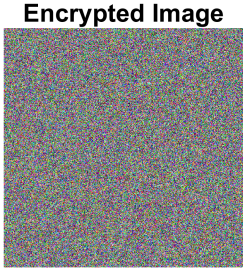
```
Chi-Square Values (Original Image):
Red Channel   : 629925.9294
Green Channel : 660452.6480
Blue Channel  : 616476.2014
```

```
chi_tent    = chi_square_hist(encrypted_tent,'Encrypted Image (Tent)');
```

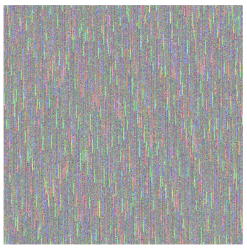
```
Chi-Square Values (Encrypted Image (Tent)):
Red Channel   : 219.5437
Green Channel : 215.6220
Blue Channel  : 230.7395
```

```
%% Decryption process:

% TENT CHAOTIC MAP GENERATION
% ==============================

alpha = 1.99;
x0 = 0.52;

% Generate chaotic sequence
tent_seq = generate_tent_map(alpha, x0, N);
%% Reverse Diffusion (undo XOR)
% Step 1: Reverse diffusion
after_diffusion = reverse_diffusion(encrypted_tent, tent_seq );

% Step 2: Reverse confusion
decrypted_tent= reverse_confusion(after_diffusion, tent_seq  );

% Display
figure;
subplot(131), imshow(encrypted_tent), title('Encrypted Image');
subplot(132), imshow(after_diffusion), title('After Reverse Diffusion');
subplot(133), imshow(decrypted_tent), title('Decrypted Image');
```

Encrypted Image

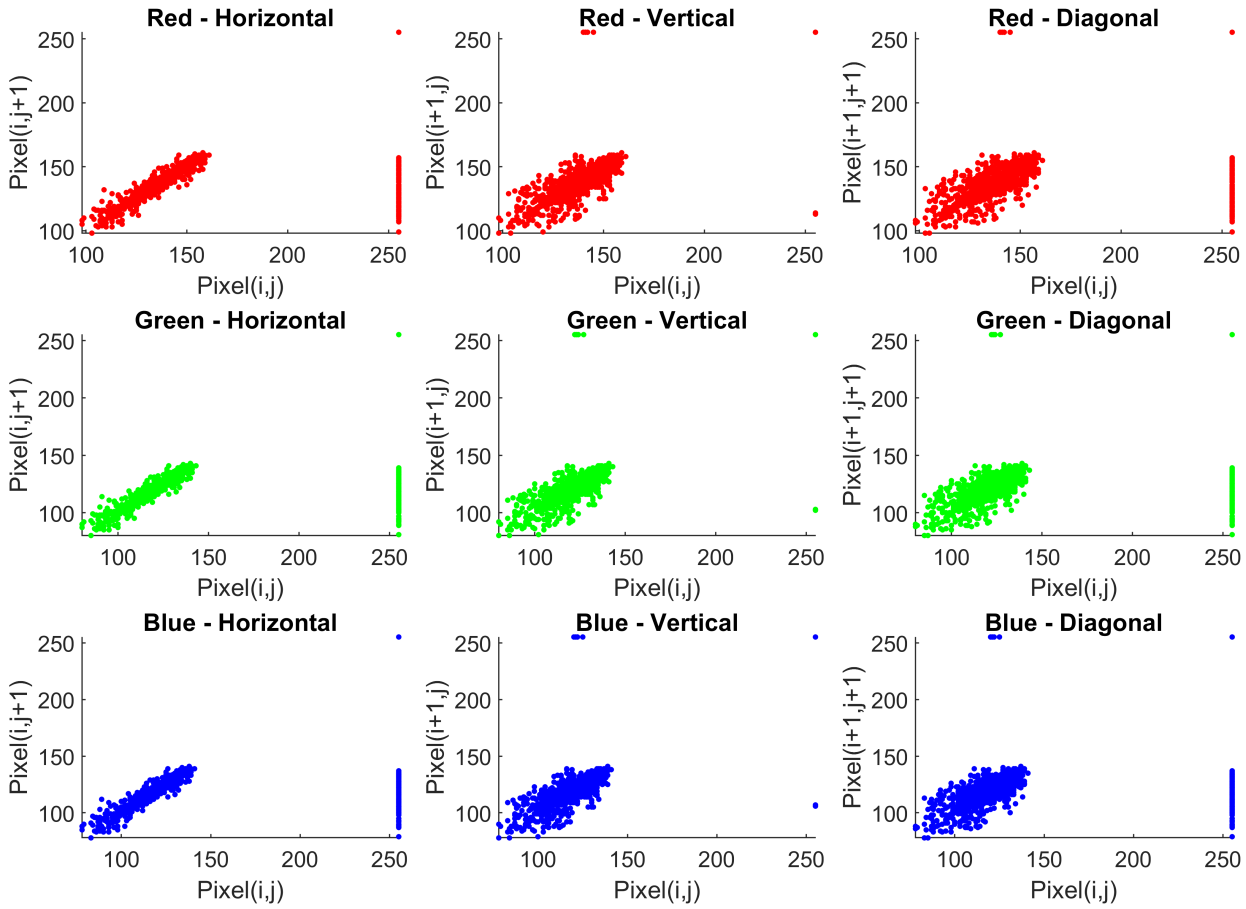


After Reverse Diffusion



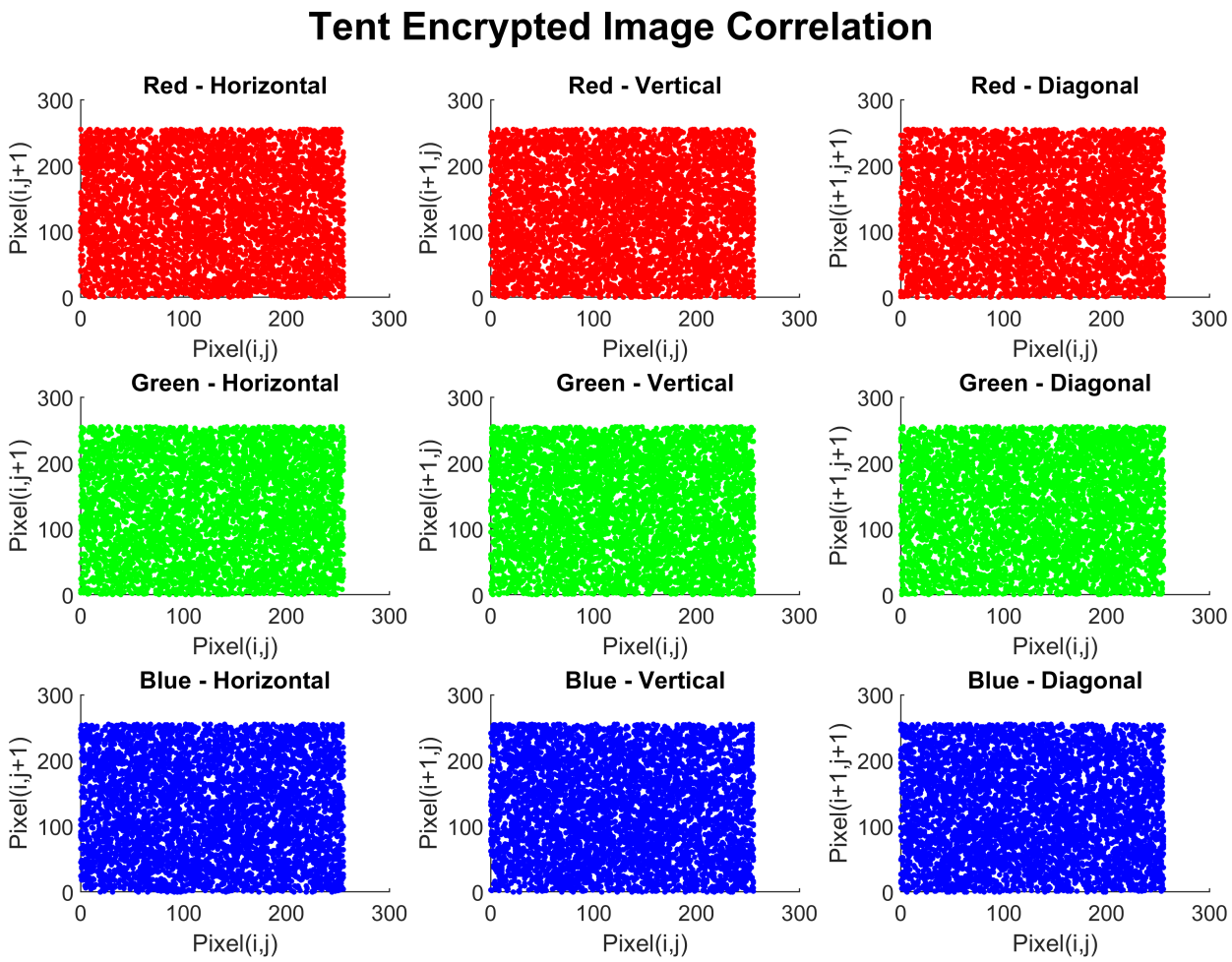
Decrypted Image

```
%% Correlation Analysis:
correlation_analysis(img, 'Original Image Correlation');
```




```
================= Correlation Summary: Original Image Correlation =================
              Horizontal     Vertical       Diagonal
-----------------------------------------------------------------
Red     :     0.9663        0.9568         0.9191
Green   :     0.9670        0.9587         0.9221
Blue    :     0.9688        0.9608         0.9263
=================================================================
```

```
correlation_analysis(encrypted_tent, 'Tent Encrypted Image Correlation');
```

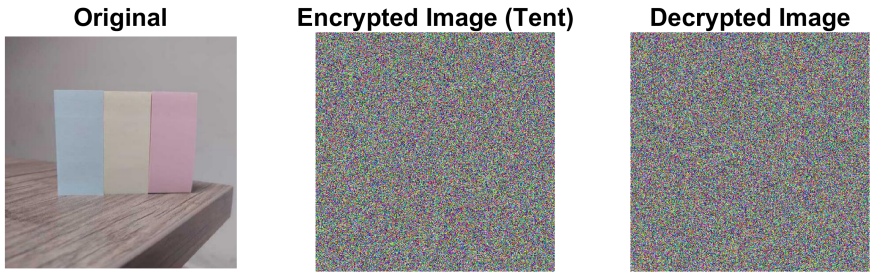



```
================= Correlation Summary: Tent Encrypted Image Correlation =================
              Horizontal     Vertical       Diagonal
-----------------------------------------------------------------
Red     :     0.0012        0.0111         0.0018
Green   :     -0.0017        0.0060         -0.0010
Blue    :     -0.0044        0.0101         -0.0023
=================================================================
```

```
%% Sensitivity of Key for Decryption process:
%% For Tent:
% Encryption Key
alpha = 1.99;
x0 = 0.52;
% Generate chaotic sequence
tent_seq = generate_tent_map(alpha, x0, N);

%% CONFUSION USING TENT
confused_tent = confusion_stage(img, tent_seq);

%% DIFFUSION USING TENT
encrypted_tent = diffusion_stage(confused_tent, tent_seq);
```

```
figure;
subplot(131), imshow(img), title('Original');
subplot(132), imshow(encrypted_tent), title('Encrypted Image (Tent)');

%% Decryption key:
alpha = 1.99;
x0 = 0.520000001;

% Generate chaotic sequence
tent_seq = generate_tent_map(alpha, x0, N);
% Step 1: Reverse diffusion
after_diffusion = reverse_diffusion(encrypted_tent, tent_seq );

% Step 2: Reverse confusion
decrypted_tent= reverse_confusion(after_diffusion, tent_seq  );
% Display
subplot(133), imshow(decrypted_tent), title('Decrypted Image');
sgtitle('Encryption key: 0.52 & Decryption key: 0.520000001')
```

Encryption key: 0.52 & Decryption key: 0.520000001




```
%% %% Timing Analysis for Tent Map Encryption
fprintf('=== Tent Map Encryption Timing Analysis ===\n');
```

```
=== Tent Map Encryption Timing Analysis ===
```

```
% Start timer
tic;
% Generate chaotic sequence
```

```
tent_seq = generate_tent_map(alpha, x0, N);

% Confusion
confused_tent = confusion_stage(img, tent_seq);

% Diffusion
encrypted_tent = diffusion_stage(confused_tent, tent_seq);
% Stop timer
time_tent = toc;

fprintf('Tent Map Encryption Time: %.6f seconds\n', time_tent);
```

```
Tent Map Encryption Time: 0.054383 seconds
```

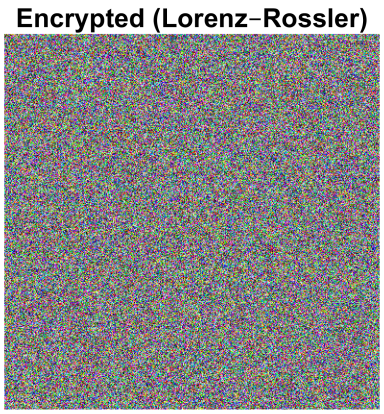
```
%% Optional: Repeat multiple times for average time
num_runs = 100;
times_tent = zeros(1,num_runs);
for k = 1:num_runs
    tic;
    tent_seq = generate_tent_map(alpha, x0, N);
    confused_tent = confusion_stage(img, tent_seq);
    encrypted_tent = diffusion_stage(confused_tent, tent_seq);
    times_tent(k) = toc;
end
avg_time_tent = mean(times_tent);
fprintf('Average Tent Map Encryption Time over %d runs: %.6f seconds\n', num_runs, avg_time_ten
```

```
Average Tent Map Encryption Time over 100 runs: 0.041369 seconds
```

Lorentz Rossler Method

```
%% Chaotic Maps Image Encryption
clc; clear; close all;
%% Read the image
img =imread(' C:\Users\Control Lab\Downloads\codes\codes\state of art methods\paper.PNG');
img = uint8(img);

[R, C, ~] = size(img);
N = R*C;

%% Lorenz-Rossler parameters
delta = 20;
r = 20;
a = 9;
beta = 8.5;
b = 0;
c = 8;

% initial conditions
x0 = 0.001;  y0 = 0.001;  z0 = 0.1;
```

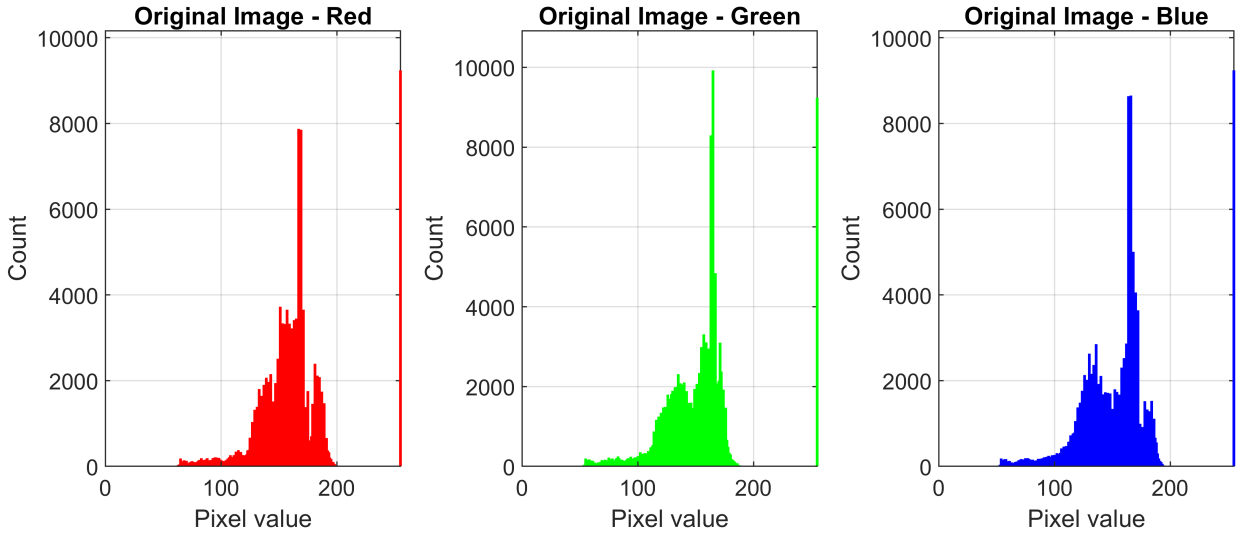
```
%% === 1. Generate chaotic sequences ===
[X, Y, Z] = lorenz_rossler_system(delta, r, a, beta, b, c, [x0; y0; z0], N);

%% === 2. Normalize chaotic sequences ===
Xn = uint8(mod(abs(X)*1e14, 256));
Yn = uint8(mod(abs(Y)*1e14, 256));
Zn = uint8(mod(abs(Z)*1e14, 256));

%% === 3. Split image ===
Rch = img(:,:,1);
Gch = img(:,:,2);
Bch = img(:,:,3);

Rvec = Rch(:);
Gvec = Gch(:);
Bvec = Bch(:);

%% === 4. Sort sequences and obtain confusion index ===
[~, idxX] = sort(Xn);
[~, idxY] = sort(Yn);
[~, idxZ] = sort(Zn);

%% === 5. Confusion (rearranging pixels) ===
shfR = Rvec(idxX);
shfG = Gvec(idxY);
shfB = Bvec(idxZ);

%% === 6. Diffusion (XOR) ===
encR = bitxor(shfR, Xn);
encG = bitxor(shfG, Yn);
encB = bitxor(shfB, Zn);

%% === 7. Merge encrypted channels ===
enc_img = zeros(R,C,3,'uint8');
enc_img(:,:,1) = reshape(encR, [R,C]);
enc_img(:,:,2) = reshape(encG, [R,C]);
enc_img(:,:,3) = reshape(encB, [R,C]);

%% Display
figure;
subplot(1,2,1), imshow(img), title('Original');
subplot(1,2,2), imshow(enc_img), title('Encrypted (Lorenz-Rossler)');
```




```
%% %% Histogram Analysis for Lorentz Rossler Map Encryption
fprintf('=== Lorentz Rossler Map Encryption Histogram Analysis ===\n');
```

```
=== Lorentz Rossler Map Encryption Histogram Analysis ===
```

```
% Histogram Analysis
plot_hist(img, 'Original Image');
```




```
plot_hist(enc_img, 'Encrypted Image  (Lorenz-Rossler)');
```


```
chi_Image   = chi_square_hist(img, 'Original Image');
```

```
Chi-Square Values (Original Image):
Red Channel   : 629925.9294
Green Channel : 660452.6480
Blue Channel  : 616476.2014
```

```
chi_LR    = chi_square_hist(enc_img,'Lorenz-Rossler' );
```

```
Chi-Square Values (Lorenz-Rossler):
Red Channel   : 215.0393
Green Channel : 256.0580
Blue Channel  : 266.2122
```

```
%% === DECRYPTION ===
% Input: enc_img (encrypted image)
% Output: dec_img (decrypted image)

[R, C, ~] = size(enc_img);
N = R * C;

%% 1. Regenerate chaotic sequences (must be identical)
[X, Y, Z] = lorenz_rossler_system(delta, r, a, beta, b, c, [x0; y0; z0], N);

%% 2. Normalize sequences (same as encryption)
Xn = uint8(mod(abs(X)*1e14, 256));
Yn = uint8(mod(abs(Y)*1e14, 256));
Zn = uint8(mod(abs(Z)*1e14, 256));

%% 3. Split encrypted image into channels
encR = enc_img(:,:,1); encG = enc_img(:,:,2); encB = enc_img(:,:,3);

encRvec = encR(:); encGvec = encG(:); encBvec = encB(:);

%% 4. Undo diffusion (XOR again with same sequences)
shfR = bitxor(encRvec, Xn);   % reverses encR = bitxor(shfR, Xn)
shfG = bitxor(encGvec, Yn);
```

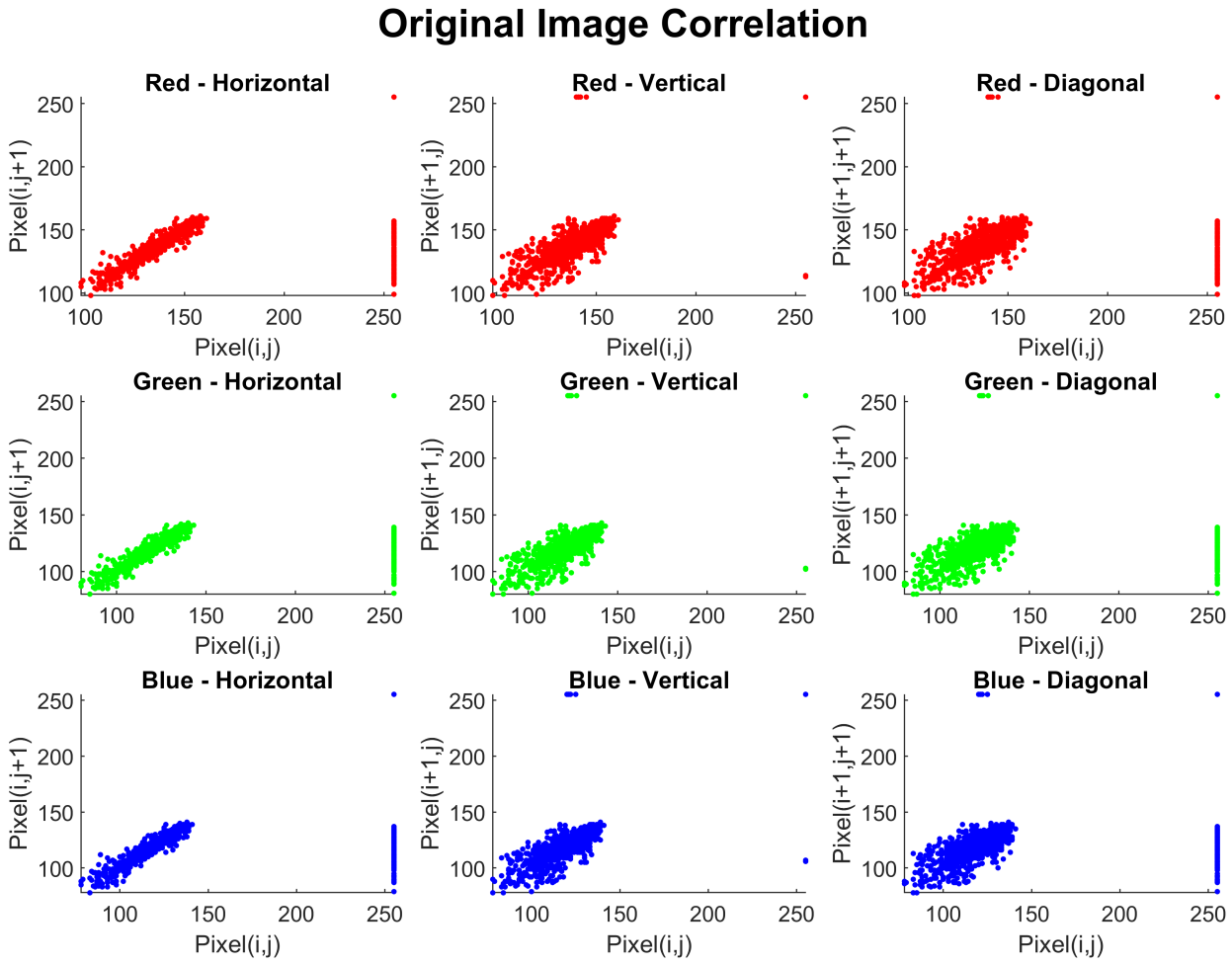
```
shfB = bitxor(encBvec, Zn);

%% 5. Undo confusion (reorder pixels back to original)
[~, idxX] = sort(Xn);   % same index used in encryption
[~, idxY] = sort(Yn);
[~, idxZ] = sort(Zn);

invR = zeros(N,1,'uint8');  invG = zeros(N,1,'uint8');  invB = zeros(N,1,'uint8');

invR(idxX) = shfR;   % place pixels back using the index
invG(idxY) = shfG;
invB(idxZ) = shfB;

%% 6. Merge channels to reconstruct original image
dec_img = zeros(R,C,3,'uint8');
dec_img(:,:,1) = reshape(invR, [R,C]);
dec_img(:,:,2) = reshape(invG, [R,C]);
dec_img(:,:,3) = reshape(invB, [R,C]);

%% 7. Display
figure;
subplot(1,3,1), imshow(enc_img), title('Encrypted Image');
subplot(1,3,2), imshow(dec_img), title('Decrypted Image');
subplot(1,3,3), imshow(img), title('Original Image');
```


```
% Optional check
if isequal(uint8(img), dec_img)
    disp('Decryption successful: original image recovered.');
```

```
else
    disp('Decryption failed: mismatch detected.');
end
```

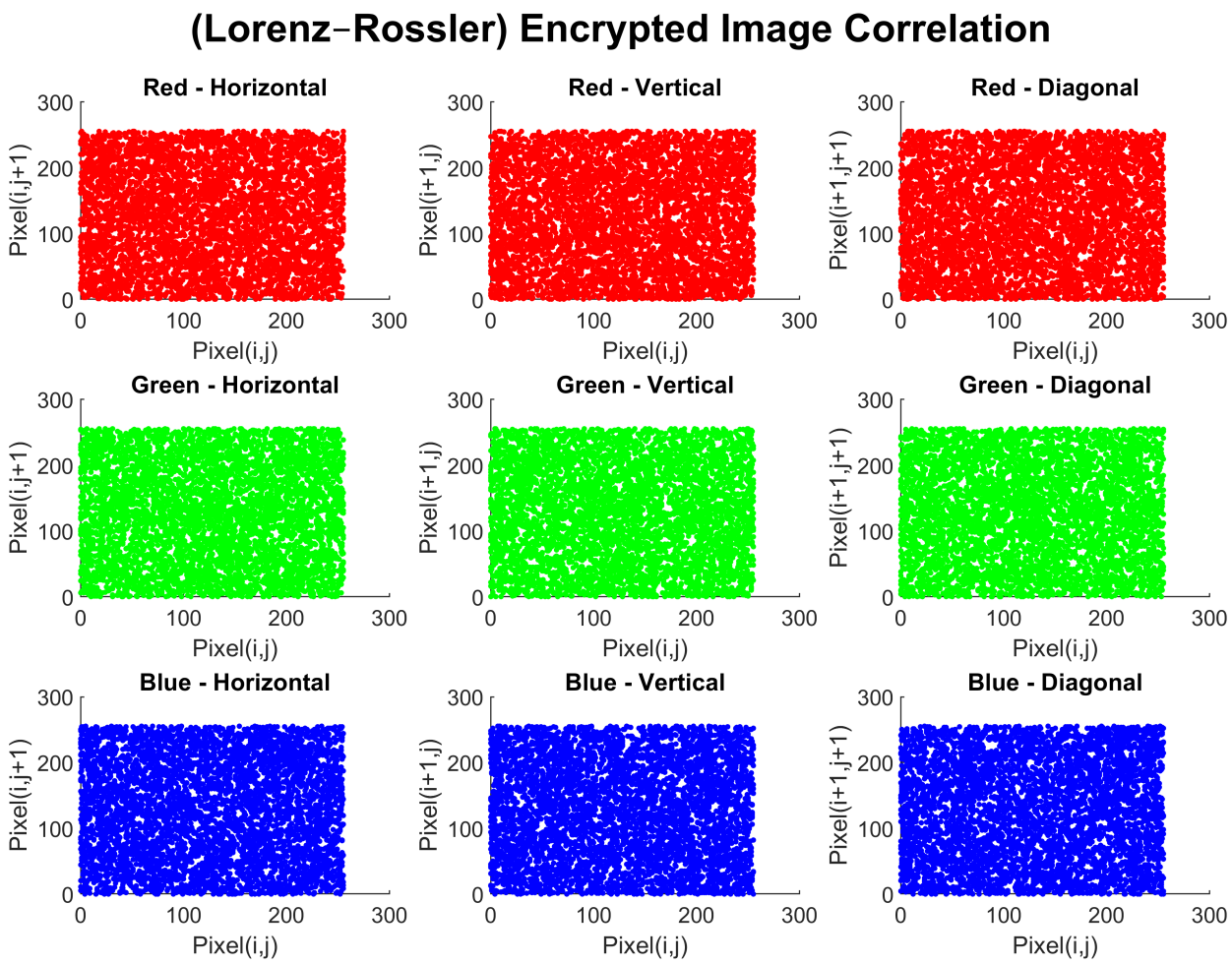
```
Decryption successful: original image recovered.
```

```
%% Correlation Analysis:
correlation_analysis(img, 'Original Image Correlation');
```




```
================= Correlation Summary: Original Image Correlation =================
              Horizontal     Vertical       Diagonal
------------------------------------------------------------------
Red     :     0.9663         0.9568         0.9191
Green   :     0.9670         0.9587         0.9221
Blue    :     0.9688         0.9608         0.9263
==================================================================
```

```
correlation_analysis(enc_img, '(Lorenz-Rossler) Encrypted Image Correlation');
```




```
================= Correlation Summary: (Lorenz–Rossler) Encrypted Image Correlation =================
              Horizontal     Vertical       Diagonal
-------------------------------------------------------------
Red     :     -0.0021        0.0031         0.0035
Green   :     0.0004        -0.0035         0.0045
Blue    :     0.0049        -0.0031         0.0003
=============================================================
```

```
%% %% Timing Analysis for Lorenz–Rossler Encryption
fprintf('=== Lorenz–Rossler Encryption Timing Analysis ===\n');
```

```
=== Lorenz–Rossler Encryption Timing Analysis ===
```

```
% Start timer
tic;

% 1. Generate chaotic sequences
[X, Y, Z] = lorenz_rossler_system(delta, r, a, beta, b, c, [x0; y0; z0], N);

% 2. Normalize chaotic sequences
Xn = uint8(mod(abs(X)*1e14, 256));
Yn = uint8(mod(abs(Y)*1e14, 256));
Zn = uint8(mod(abs(Z)*1e14, 256));
```

```
% 3. Split image
Rch = img(:,:,1); Gch = img(:,:,2); Bch = img(:,:,3);
Rvec = Rch(:); Gvec = Gch(:); Bvec = Bch(:);

% 4. Confusion
[~, idxX] = sort(Xn); [~, idxY] = sort(Yn); [~, idxZ] = sort(Zn);
shfR = Rvec(idxX); shfG = Gvec(idxY); shfB = Bvec(idxZ);

% 5. Diffusion (XOR)
encR = bitxor(shfR, Xn);
encG = bitxor(shfG, Yn);
encB = bitxor(shfB, Zn);

% 6. Merge channels
enc_img = zeros(R,C,3,'uint8');
enc_img(:,:,1) = reshape(encR, [R,C]);
enc_img(:,:,2) = reshape(encG, [R,C]);
enc_img(:,:,3) = reshape(encB, [R,C]);

% Stop timer
time_LR = toc;
fprintf('Lorenz-Rossler Encryption Time: %.6f seconds\n', time_LR);
```

```
Lorenz-Rossler Encryption Time: 0.269954 seconds
```

```
%% Optional: Repeat multiple times for average time
num_runs = 100;
times_LR = zeros(1,num_runs);
for k = 1:num_runs
    tic;
    [X, Y, Z] = lorenz_rossler_system(delta, r, a, beta, b, c, [x0; y0; z0], N);
  Xn = uint8(mod(abs(X)*1e14, 256));
    Yn = uint8(mod(abs(Y)*1e14, 256));
Zn = uint8(mod(abs(Z)*1e14, 256));

% Extract and flatten image channels
Rvec = img(:,:,1);
Rvec = Rvec(:);

Gvec = img(:,:,2);
Gvec = Gvec(:);

Bvec = img(:,:,3);
Bvec = Bvec(:);

    [~, idxX] = sort(Xn); [~, idxY] = sort(Yn); [~, idxZ] = sort(Zn);
    shfR = Rvec(idxX); shfG = Gvec(idxY); shfB = Bvec(idxZ);
    encR = bitxor(shfR, Xn); encG = bitxor(shfG, Yn); encB = bitxor(shfB, Zn);
    enc_img = zeros(R,C,3,'uint8'); enc_img(:,:,1) = reshape(encR,[R,C]); enc_img(:,:,2) = resh
    times_LR(k) = toc;
end
avg_time_LR = mean(times_LR);
```

```
fprintf('Average Lorenz-Rossler Encryption Time over %d runs: %.6f seconds\n', num_runs, avg_ti
```

```
Average Lorenz-Rossler Encryption Time over 100 runs: 0.211372 seconds
```