



⚡ Grok 3 ▾






Grok 3



i have two similar documents which were photographed under very similar, yet not exactly the same lighting conditions. a color standard with 6 primary colors, white, dark gray and black. and a gray scale standard with 19 graduations were included in each photograph. using only the color standards as reference to infer differences in lighting conditions, propose a method for developing an algorithm that when applied to the 1st image would cause it to appear as if it was photographed in the same lighting conditions as the 2nd image.

 Thought for 1m 28s

Expand for details



To address your query, I'll propose a method to develop an algorithm that adjusts the first image of a document to appear as if it was photographed under the same lighting conditions as the second image, using only the color standards (six primary colors, white, dark gray, and black) as reference. The color standards provide a consistent basis to infer and correct for lighting differences, and I'll outline a practical approach using an affine transformation, which is well-suited for this task given the similarity in lighting conditions.

## Problem Understanding

You have two photographs of similar documents, each containing:

- A **color standard** with six primary colors (e.g., possibly red, green, blue, cyan, magenta, yellow, though not specified), white, dark gray, and black—totaling 9 distinct color patches.
- A **gray scale standard** with 19 graduations, which we won't use since the task specifies using only the color standards.

The lighting conditions in the two images are very similar but not identical, causing slight differences in how colors appear. The goal is to transform the first image so its colors match

differences in how colors appear. The goal is to transform the first image so its colors match those of the second image, as if both were taken under the same lighting.



## Proposed Method

The method involves computing and applying an **affine transformation**—a combination of a linear transformation (via a 3x3 matrix) and a translation (via a 3x1 vector)—to adjust the RGB values of the first image based on the color standards. Here's how it works:

### Step 1: Extract Color Standard Data

- **Identify the Color Patches:** In each image, locate the 9 color standards (six primary colors, white, dark gray, black).
- **Measure RGB Values:**
  - For the first image, extract the RGB values of these 9 patches, forming a 9x3 matrix called (  $C_1$  ), where each row is an RGB triplet (e.g., [R, G, B]).
  - For the second image, do the same, forming a 9x3 matrix called (  $C_2$  ).
- These values reflect how the true colors of the standards appear under each image's lighting conditions.

### Step 2: Compute the Affine Transformation

- **Objective:** Find a transformation that maps the colors in (  $C_1$  ) to those in (  $C_2$  ), effectively adjusting for the lighting difference.
- **Model:** Use an affine transformation of the form  $p' = T \cdot p + b$ , where:
  - (  $p$  ) is an RGB pixel value from the first image (a 3x1 vector),
  - (  $T$  ) is a 3x3 matrix (linear transformation),
  - (  $b$  ) is a 3x1 vector (translation),
  - (  $p'$  ) is the transformed RGB value matching the second image's lighting.
- **Formulation:**
  - Transpose (  $C_1$  ) and (  $C_2$  ) to  $X = C_1^T$  (3x9) and  $Y = C_2^T$  (3x9), where each column

is an RGB vector.

×

⚡ Grok 3

- Augment (  $X$  ) with a row of ones to account for the translation:  $Z = \begin{bmatrix} X \\ \text{ones}(1, 9) \end{bmatrix}$ , a 4x9 matrix.
- Solve for a 3x4 matrix (  $W$  ) such that  $W \cdot Z \approx Y$ , using least squares:  $W = Y \cdot \text{pinv}(Z)$ , where  $\text{pinv}(Z)$  is the pseudoinverse of (  $Z$  ).
- Extract  $T = W[:, 0:3]$  (columns 0 to 2, a 3x3 matrix) and  $b = W[:, 3]$  (column 3, a 3x1 vector).
- This transformation minimizes the difference between the transformed colors of the first image and the colors of the second image across all 9 standards.

### Step 3: Apply the Transformation

- **Transform the Image:** For each pixel (  $p$  ) in the first image (a 3x1 RGB vector), compute the new pixel value:

$$p' = T \cdot p + b$$

- **Ensure Valid Range:** Clip the resulting RGB values to [0, 255] (assuming an 8-bit image) to prevent overflow or invalid colors.
- **Output:** The transformed first image should now appear as if photographed under the same lighting as the second image.

## Algorithm in Pseudocode

Here's a concrete implementation in pseudocode (easily translatable to languages like Python):

python

# Input:

# - image1: First image (height x width x 3)

# - C1: 9x3 matrix of RGB values of color standards from image1

# - C2: 9x3 matrix of RGB values of color standards from image2

# Step 1: Compute transformation

X = transpose(C1) # 3x9

Y = transpose(C2) # 3x9

W = Y \* pinv(X) # 3x4 matrix

```

Z = stack(X, ones(1,9))    # 4x9, augment with row of ones
W = Y * pseudoinverse(Z)   # 3x4
T = W[:, 0:3]              # 3x3 matrix
b = W[:, 3]                # 3x1 vector

# Step 2: Apply to image1
transformed_image = zeros_like(image1)
for i from 0 to height-1:
    for j from 0 to width-1:
        p = image1[i, j]    # 3x1 RGB vector
        p_transformed = T * p + b    # Affine transformation
        p_transformed = clip(p_transformed, 0, 255) # Keep in valid range
        transformed_image[i, j] = p_transformed

# Output: transformed_image

```

Grok 3



## Why This Works

- **Sufficient Data:** With 9 color standards (27 data points: 9 RGB triplets), we can robustly estimate the 12 parameters of the affine transformation (9 for (  $T$  ), 3 for (  $b$  )).
- **Lighting Adjustment:** The transformation captures both color shifts (e.g., color casts) via (  $T$  ) and brightness/contrast changes via (  $b$  ), suitable for “very similar” lighting differences.
- **Simplicity:** An affine model assumes a linear relationship between the color spaces under different lighting, which is reasonable given the problem’s context.

## Assumptions and Considerations

- **Uniform Lighting:** The method assumes lighting is consistent across each image, as the color standards are used globally.
- **Color Space:** RGB values are treated as-is (e.g., sRGB). For more precision, convert to linear RGB before transforming and back afterward, though this may be unnecessary given the similarity in lighting.
- **Patch Identification:** Accurate extraction of the color standard RGB values from both

Images is critical.



⚡ Grok 3 ▾



## Conclusion

This method—extracting color standard RGB values, computing an affine transformation via least squares, and applying it to the first image—effectively adjusts the first image to match the lighting conditions of the second. It leverages the 9 color standards to infer and correct lighting differences, fulfilling the query's requirements efficiently and accurately.

```
write the python code to complete this. inputs from
command line: C1 ( name of file containing CSV 9x3 RGB
values) C2 (name of similar file) image1 in .jpg or .png
encoding. output transformed .png image. add an optional
verbose flag which enables standard output of intermediate
calculations: X, Y, Z, W, T, b
```

Below is a Python script that implements the method described in the previous response. It takes command-line inputs for:

- **C1** : Path to a CSV file containing a 9x3 matrix of RGB values for the color standards in the first image.
- **C2** : Path to a CSV file containing a 9x3 matrix of RGB values for the color standards in the second image.
- **image1** : Path to the first image (in .jpg or .png format).
- Outputs a transformed image as a .png file.
- Includes an optional **--verbose** flag to print intermediate calculations (matrices X, Y, Z, W, T, b).

The script uses NumPy for matrix operations and OpenCV for image processing, as they are efficient and standard for such tasks.

python

✕

⚡ Grok 3 ▾



```
import numpy as np
import cv2
import argparse
import os
```

```
def parse_args():
    parser = argparse.ArgumentParser(description="Transform image1 to match the lighting of image2")
    parser.add_argument("C1", type=str, help="Path to CSV file with 9x3 RGB values for image1")
    parser.add_argument("C2", type=str, help="Path to CSV file with 9x3 RGB values for image2")
    parser.add_argument("image1", type=str, help="Path to input image1 (.jpg or .png)")
    parser.add_argument("--verbose", action="store_true", help="Enable verbose output")
    return parser.parse_args()
```

```
def load_csv_matrix(file_path):
    """Load a 9x3 CSV file into a NumPy array."""
    try:
        matrix = np.loadtxt(file_path, delimiter=',', dtype=np.float32)
        if matrix.shape != (9, 3):
            raise ValueError(f"CSV file {file_path} must contain a 9x3 matrix.")
        return matrix
    except Exception as e:
        print(f"Error loading {file_path}: {e}")
        exit(1)
```

```
def transform_image(image, T, b):
    """Apply affine transformation (T * p + b) to each pixel in the image."""
    height, width, _ = image.shape
    transformed_image = np.zeros_like(image, dtype=np.float32)

    # Reshape image for vectorized operation: (height*width, 3)
    pixels = image.reshape(-1, 3).astype(np.float32)
    # Apply transformation: T * p + b
    transformed_pixels = pixels @ T.T + b
    # Clip to valid RGB range [0, 255]
    transformed_pixels = np.clip(transformed_pixels, 0, 255)
    # Reshape back to image dimensions
    transformed_image = transformed_pixels.reshape(height, width, 3).astype(np.uint8)
    return transformed_image
```

```
def main():
    args = parse_args()

    # Load input files
    C1 = load_csv_matrix(args.C1)
    C2 = load_csv_matrix(args.C2)
    image1 = cv2.imread(args.image1)
    if image1 is None:
```

✕ `print(f"Error: Could not load image {args.image1}. Ensure it's a valid .jpg or  
exit(1)`

⚡ Grok 3 ▾

↑ 📖 ↺ 📄

```
# Convert image1 to RGB (OpenCV loads in BGR)
image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2RGB)

# Step 1: Compute transformation
# X = C1^T (3x9), Y = C2^T (3x9)
X = C1.T
Y = C2.T
# Z = [X; ones(1,9)] (4x9)
Z = np.vstack([X, np.ones((1, 9))])
# Solve W = Y * pinv(Z) (3x4)
W = Y @ np.linalg.pinv(Z)
# Extract T (3x3) and b (3x1)
T = W[:, 0:3]
b = W[:, 3]

# Verbose output
if args.verbose:
    print("Intermediate calculations:")
    print("X (C1^T):\n", X)
    print("Y (C2^T):\n", Y)
    print("Z ([X; ones]):\n", Z)
    print("W (Y * pinv(Z)):\n", W)
    print("T (linear transformation):\n", T)
    print("b (translation):\n", b)

# Step 2: Apply transformation to image1
transformed_image = transform_image(image1, T, b)

# Save output as PNG
output_path = "transformed_image.png"
transformed_image_bgr = cv2.cvtColor(transformed_image, cv2.COLOR_RGB2BGR)
cv2.imwrite(output_path, transformed_image_bgr)
print(f"Transformed image saved as {output_path}")

if __name__ == "__main__":
    main()
```

## How to Use

### 1. Requirements: Install required libraries:

```
bash
```

```
pip install numpy opencv-python
```



✖ **CSV File Format:** Each CSV file ( `C1.csv`, `C2.csv` ) should contain 9 rows and 3 columns of RGB values (0-255), e.g.:

```
255,0,0
0,255,0
0,0,255
0,255,255
255,0,255
255,255,0
255,255,255
64,64,64
0,0,0
```

### 3. Run the Script:

- Basic usage:

```
bash

python script.py C1.csv C2.csv image1.jpg
```

- With verbose output:

```
bash

python script.py C1.csv C2.csv image1.jpg --verbose
```

4. **Output:** The script generates `transformed_image.png` in the working directory.

## Features

- **Input Validation:** Checks for valid CSV files (9x3 matrices) and image formats (.jpg or .png).
- **Efficient Transformation:** Uses vectorized operations for applying the affine transformation to the entire image.
- **Verbose Mode:** Prints matrices X, Y, Z, W, T, and b when `--verbose` is specified.
- **Color Space Handling:** Converts between BGR (OpenCV) and RGB to ensure correct color processing.
- **Clipping:** Ensures transformed RGB values stay within [0, 255].

## Notes

⚡ Grok 3 ▾



- **Assumptions:** The input image is in sRGB, and RGB values in the CSV files are in the range [0, 255]. If the color standards use a different range or color space, you may need to adjust the input data or add normalization.
- **Performance:** Vectorized operations ensure efficiency even for large images.
- **Error Handling:** The script includes basic error checking for file loading and matrix shapes.

This code fulfills the requirements and provides a robust solution to transform the first image to match the lighting conditions of the second, using the color standards as reference.

implement the above but without using OpenCV

Below is a Python script that implements the image transformation without using OpenCV, relying instead on **PIL** (Pillow) for image handling and NumPy for matrix operations. The script takes the same command-line inputs:

- **C1** : Path to a CSV file with a 9x3 matrix of RGB values for color standards in the first image.
- **C2** : Path to a CSV file with a 9x3 matrix of RGB values for color standards in the second image.
- **image1** : Path to the input image (.jpg or .png).
- Outputs a transformed image as **transformed\_image.png** .
- Includes an optional **--verbose** flag to print intermediate calculations (matrices X, Y, Z, W, T, b).

## Implementation

```
python
```

```
import numpy as np
from PIL import Image
```

```

import argparse
import os

def parse_args():
    parser = argparse.ArgumentParser(description="Transform image1 to match the lighti
    parser.add_argument("C1", type=str, help="Path to CSV file with 9x3 RGB values for
    parser.add_argument("C2", type=str, help="Path to CSV file with 9x3 RGB values for
    parser.add_argument("image1", type=str, help="Path to input image1 (.jpg or .png)"
    parser.add_argument("--verbose", action="store_true", help="Enable verbose output (
    return parser.parse_args()

def load_csv_matrix(file_path):
    """Load a 9x3 CSV file into a NumPy array."""
    try:
        matrix = np.loadtxt(file_path, delimiter=',', dtype=np.float32)
        if matrix.shape != (9, 3):
            raise ValueError(f"CSV file {file_path} must contain a 9x3 matrix.")
        return matrix
    except Exception as e:
        print(f"Error loading {file_path}: {e}")
        exit(1)

def transform_image(image_array, T, b):
    """Apply affine transformation (T * p + b) to each pixel in the image array."""
    height, width, _ = image_array.shape
    # Reshape image for vectorized operation: (height*width, 3)
    pixels = image_array.reshape(-1, 3).astype(np.float32)
    # Apply transformation: T * p + b
    transformed_pixels = pixels @ T.T + b
    # Clip to valid RGB range [0, 255]
    transformed_pixels = np.clip(transformed_pixels, 0, 255)
    # Reshape back to image dimensions
    transformed_image = transformed_pixels.reshape(height, width, 3).astype(np.uint8)
    return transformed_image

def main():
    args = parse_args()

    # Load input files
    C1 = load_csv_matrix(args.C1)
    C2 = load_csv_matrix(args.C2)

    # Load image1 using PIL
    try:
        image1 = Image.open(args.image1)
        if image1.mode != 'RGB':
            image1 = image1.convert('RGB')
        image_array = np.array(image1, dtype=np.float32)
    except Exception as e:
        print(f"Error: Could not load image {args.image1}. Ensure it's a valid image or

```

```
print(f"ERROR: Could not load image {args.image1}. Ensure it's a valid .jpg or .png file.")
exit(1)
```



⚡ Grok 3



```
# Step 1: Compute transformation
# X = C1^T (3x9), Y = C2^T (3x9)
X = C1.T
Y = C2.T
# Z = [X; ones(1,9)] (4x9)
Z = np.vstack([X, np.ones((1, 9))])
# Solve W = Y * pinv(Z) (3x4)
W = Y @ np.linalg.pinv(Z)
# Extract T (3x3) and b (3x1)
T = W[:, 0:3]
b = W[:, 3]

# Verbose output
if args.verbose:
    print("Intermediate calculations:")
    print("X (C1^T):\n", X)
    print("Y (C2^T):\n", Y)
    print("Z ([X; ones]):\n", Z)
    print("W (Y * pinv(Z)):\n", W)
    print("T (linear transformation):\n", T)
    print("b (translation):\n", b)

# Step 2: Apply transformation to image1
transformed_array = transform_image(image_array, T, b)

# Save output as PNG using PIL
output_path = "transformed_image.png"
transformed_image = Image.fromarray(transformed_array, mode='RGB')
transformed_image.save(output_path, format='PNG')
print(f"Transformed image saved as {output_path}")

if __name__ == "__main__":
    main()
```

## How to Use

### 1. Requirements: Install required libraries:

```
bash
```

```
pip install numpy pillow
```

### 2. CSV File Format: Each CSV file ( **C1.csv** , **C2.csv** ) should contain 9 rows and 3 columns of RGB values (0-255) as follows:

RGB values (0-255), e.g.:



⚡ Grok 3 ▾



```
255,0,0
0,255,0
0,0,255
0,255,255
255,0,255
255,255,0
255,255,255
64,64,64
0,0,0
```

### 3. Run the Script:

- Basic usage:

```
bash
```

```
python script.py C1.csv C2.csv image1.jpg
```

- With verbose output:

```
bash
```

```
python script.py C1.csv C2.csv image1.jpg --verbose
```

4. **Output:** The script generates `transformed_image.png` in the working directory.

## Key Differences from OpenCV Version

- **Image Handling:** Uses `PIL` instead of OpenCV. `PIL` loads images in RGB format directly, eliminating the need for color space conversion (BGR to RGB).
- **Image Loading/Saving:** `PIL.Image.open` loads the image, and `Image.fromarray` converts the transformed array back to an image for saving as PNG.
- **Array Handling:** The image is converted to a NumPy array for processing, consistent with the previous approach.
- **Dependencies:** Only requires Pillow and NumPy, which are lightweight and widely used.

## Features

- **Input Validation:** Checks for valid 9x3 CSV files and supported image formats (.jpg or .png).

⚡ Grok 3 ▼



- **Efficient Transformation:** Uses vectorized NumPy operations for fast processing of the entire image.
- **Verbose Mode:** Prints matrices X, Y, Z, W, T, and b when `--verbose` is specified.
- **Color Space:** Assumes RGB input (Pillow's default), with CSV values in [0, 255].
- **Error Handling:** Includes checks for file loading and image format issues.

## Notes

- **Assumptions:** RGB values in CSV files and the image are in the range [0, 255] and in sRGB color space. Adjust the script if a different range or color space is used (e.g., normalize or convert to linear RGB).
- **Performance:** Vectorized operations ensure efficiency for large images.
- **Image Format:** Outputs a PNG file, which is lossless and suitable for the transformed image.

This script provides a robust, OpenCV-free solution to transform the first image to match the lighting conditions of the second, using the 9 color standards as reference, while maintaining all functionality from the previous implementation.

modify that code to allow an arbitrary number of color comparison values rather than requiring exactly 9 values. maybe error out if less than 5 are specified for good measure. also have the output image file be a variation of the input filename with "\_xformed" appended to the root filename.

Below is a modified version of the previous Python script that:

- Allows an arbitrary number of color comparison values in the CSV files ( `C1.csv` and `C2.csv` ), instead of requiring exactly 9.
- Enforces a minimum of 5 color standards for robust transformation estimation.

- ✗ Generates the output image filename by appending `_xformed` to the root of the input filename (e.g., `image1.jpg` becomes `image1_xformed.png`).
- Retains all other functionality, including the `--verbose` flag for printing intermediate calculations (X, Y, Z, W, T, b) and uses `PIL` and `NumPy` without `OpenCV`.

## Modified Python Code

python





```
import numpy as np
from PIL import Image
import argparse
import os

def parse_args():
    parser = argparse.ArgumentParser(description="Transform image1 to match the lighti
    parser.add_argument("C1", type=str, help="Path to CSV file with Nx3 RGB values for
    parser.add_argument("C2", type=str, help="Path to CSV file with Nx3 RGB values for
    parser.add_argument("image1", type=str, help="Path to input image1 (.jpg or .png)"
    parser.add_argument("--verbose", action="store_true", help="Enable verbose output
    return parser.parse_args()

def load_csv_matrix(file_path):
    """Load an Nx3 CSV file into a NumPy array, ensuring at least 5 rows."""
    try:
        matrix = np.loadtxt(file_path, delimiter=',', dtype=np.float32)
        if matrix.shape[1] != 3:
            raise ValueError(f"CSV file {file_path} must have exactly 3 columns (RGB v
        if matrix.shape[0] < 5:
            raise ValueError(f"CSV file {file_path} must contain at least 5 rows of co
        return matrix
    except Exception as e:
        print(f"Error loading {file_path}: {e}")
        exit(1)

def transform_image(image_array, T, b):
    """Apply affine transformation (T * p + b) to each pixel in the image array."""
    height, width, _ = image_array.shape
    # Reshape image for vectorized operation: (height*width, 3)
    pixels = image_array.reshape(-1, 3).astype(np.float32)
    # Apply transformation: T * p + b
    transformed_pixels = pixels @ T.T + b
    # Clip to valid RGB range [0, 255]
    transformed_pixels = np.clip(transformed_pixels, 0, 255)
    # Reshape back to image dimensions
    . . . . .
```

transformed\_image = transformed\_pixels.reshape(height, width, 3).astype(np.uint8)

× `return transformed_image` ⚡ Grok 3 ▾    

```
def get_output_filename(input_filename):
    """Generate output filename by appending '_xformed' to the root of the input file:
    base, _ = os.path.splitext(input_filename)
    return f"{base}_xformed.png"

def main():
    args = parse_args()

    # Load input files
    C1 = load_csv_matrix(args.C1)
    C2 = load_csv_matrix(args.C2)

    # Ensure C1 and C2 have the same number of rows
    if C1.shape[0] != C2.shape[0]:
        print(f"Error: C1 has {C1.shape[0]} rows, but C2 has {C2.shape[0]} rows. They r
        exit(1)

    # Load image1 using PIL
    try:
        image1 = Image.open(args.image1)
        if image1.mode != 'RGB':
            image1 = image1.convert('RGB')
        image_array = np.array(image1, dtype=np.float32)
    except Exception as e:
        print(f"Error: Could not load image {args.image1}. Ensure it's a valid .jpg or
        exit(1)

    # Step 1: Compute transformation
    #  $X = C1^T$  (3xN),  $Y = C2^T$  (3xN)
    X = C1.T
    Y = C2.T
    #  $Z = [X; \text{ones}(1, N)]$  (4xN)
    Z = np.vstack([X, np.ones((1, C1.shape[0]))])
    # Solve  $W = Y * \text{pinv}(Z)$  (3x4)
    W = Y @ np.linalg.pinv(Z)
    # Extract  $T$  (3x3) and  $b$  (3x1)
    T = W[:, 0:3]
    b = W[:, 3]

    # Verbose output
    if args.verbose:
        print("Intermediate calculations:")
        print("X ( $C1^T$ ):\n", X)
        print("Y ( $C2^T$ ):\n", Y)
        print("Z ([X; ones]):\n", Z)
        print("W ( $Y * \text{pinv}(Z)$ ):\n", W)
        print("T (linear transformation):\n", T)
```



× `print(f"Linear transformation: {T}, {b}")`  
`print("b (translation):\n", b)`

⚡ Grok 3 ✓



*# Step 2: Apply transformation to image1*

`transformed_array = transform_image(image_array, T, b)`

*# Save output as PNG using PIL*

`output_path = get_output_filename(args.image1)`

`transformed_image = Image.fromarray(transformed_array, mode='RGB')`

`transformed_image.save(output_path, format='PNG')`

`print(f"Transformed image saved as {output_path}")`

`if __name__ == "__main__":`  
`main()`

## Changes Made

### 1. Arbitrary Number of Color Standards:

- Modified `load_csv_matrix` to accept an Nx3 matrix (where  $N \geq 5$ ) instead of requiring exactly 9 rows.
- Added a check to ensure at least 5 color standards are provided, as fewer may lead to an underdetermined system for the affine transformation (12 parameters require sufficient data points).
- Updated the transformation computation to use `C1.shape[0]` for the number of color standards, making it flexible for any  $N \geq 5$ .
- Added a check to ensure `C1` and `C2` have the same number of rows.

### 2. Output Filename:

- Added the `get_output_filename` function to generate the output filename by appending `_xformed` to the root of the input filename (e.g., `image1.jpg` → `image1_xformed.png`).
- Uses `os.path.splitext` to split the input filename into its root and extension.

### 3. Error Handling:

- Added a check to ensure `C1` and `C2` have the same number of rows.
- Enforces a minimum of 5 color standards in `load_csv_matrix`.
- Retained existing error handling for invalid CSV files and image loading.

## ✕ How to Use

⚡ Grok 3 ▾



### 1. Requirements:

```
bash
```

```
pip install numpy pillow
```

### 2. CSV File Format: Each CSV file ( **C1.csv** , **C2.csv** ) should contain Nx3 RGB values ( $N \geq 5$ , 0-255), e.g.:

```
255,0,0
0,255,0
0,0,255
0,255,255
255,0,255
```

### 3. Run the Script:

- Basic usage:

```
bash
```

```
python script.py C1.csv C2.csv image1.jpg
```

- With verbose output:

```
bash
```

```
python script.py C1.csv C2.csv image1.jpg --verbose
```

### 4. Output: The transformed image is saved as **<input\_filename>\_xformed.png** (e.g., **image1\_xformed.png** ).

## Notes

- **Minimum 5 Colors:** Requiring at least 5 color standards ensures enough data points (15 values) to robustly estimate the 12 parameters of the affine transformation (3x3 matrix  $T$  + 3x1 vector  $b$ ).

- **Output Format:** The output is always PNG for lossless quality, regardless of input format

× (.jpg or .png).

⚡ Grok 3 ▾



- **Assumptions:** RGB values in CSV files and the image are in [0, 255] and sRGB color space. Adjust if needed for other ranges or color spaces.
- **Performance:** Vectorized NumPy operations ensure efficiency, and **PIL** handles image I/O efficiently.

This script meets all requirements, allowing an arbitrary number of color standards ( $\geq 5$ ) and generating a dynamically named output file while preserving the original functionality.

💡 Think Harder



Ask anything



DeepSearch



Think

