Homework 3 Writeup

Instructions

- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- **Please make this document anonymous.**

Bayer Image Interpolation

First part of this project is to make the code that interpolating the given bayer image into proper RGB image.

```
def bayer_to_rgb_bilinear(bayer_img):

    height, width = bayer_img.shape
    rgb_img = np.zeros((height + 2, width + 2, 3), dtype=np.uint8)
    # added zero padding of 1 pixel width around the image for
    calculation efficiency
    # Extract bayer image into each channel of rgb image
    # Range starts from 1 due to the padding
    rgb_img[1:-1:2, 1:-1:2, 0] = bayer_img[0::2, 0::2]
    rgb_img[1:-1:2, 2:-1:2, 1] = bayer_img[0::2, 1::2]
    rgb_img[2:-1:2, 1:-1:2, 1] = bayer_img[1::2, 0::2]
    rgb_img[2:-1:2, 2:-1:2, 2] = bayer_img[1::2, 1::2]

    # For the R channel
    rgb_img[1:height:2, 2:width+1:2, 0] = (rgb_img[1:height:2, 1:width
    :2,0]//2 +
    rgb_img[1:height:2, 3:width+2:2,0]//2)
    rgb_img[2:height+1:2, 1:width:2, 0] = (rgb_img[1:height:2, 1:width
    :2,0]//2 +
    rgb_img[3:height+2:2, 1:width:2,0]//2)

    rgb_img[2:height+1:2, 2:width+1:2, 0] = (
        rgb_img[1:height:2, 1:width:2, 0]//4 + rgb_img[1:height:2, 3:
    width+2:2, 0]//4 +
        rgb_img[3:height+2:2, 1:width:2, 0]//4 + rgb_img[3:height+2:2,
     3:width+2:2, 0]//4
    )
    #For the G channel
    rgb_img[1:-1:2, 1:-1:2, 1] = (
        rgb_img[1:-1:2, 0:-2:2, 1]//4 +
        rgb_img[0:-2:2, 1:-1:2, 1]//4 +
        rgb_img[2::2, 1:-1:2, 1]//4 +
        rgb_img[1:-1:2, 2::2, 1]//4
    )
    rgb_img[2:height+1:2, 2:width+1:2, 1] = (
```

```
        rgb_img[2:height+1:2, 1:width:2, 1]//4 +
        rgb_img[1:height:2, 2:width+1:2, 1]//4 +
        rgb_img[3:height+2:2, 2:width+1:2, 1]//4 +
        rgb_img[2:height+1:2, 3:width+2:2, 1]//4
    )

    # For the B channel
    rgb_img[2::2, 1:-2:2, 2] = (rgb_img[2::2, 0:-3:2,2]//2 +
    rgb_img[2::2, 2:-1:2,2]//2)

    rgb_img[1:-2:2, 2::2, 2] = (rgb_img[0:-3:2, 2::2,2]//2 +
    rgb_img[2:-1:2, 2::2,2]//2)

    rgb_img[1:-2:2, 1:-2:2, 2] = (
        rgb_img[0:-3:2, 0:-3:2, 2]//4 +
        rgb_img[0:-3:2, 2:-1:2, 2]//4 +
        rgb_img[2:-1:2, 0:-3:2, 2]//4 +
        rgb_img[2:-1:2, 2:-1:2, 2]//4
    )
    #Remove Padding
    rgb_img = rgb_img[1:-1, 1:-1, :]

    return rgb_img

```

Interesting Implementation Detail

I'll explain my code step by step.

1. First, I made the empty array for result image with 2 pixels bigger than bayer image. This is because I wanted to add zero padding of 1 pixel around the image, for edge calculation.
2. Then I brought the each R,G,B pixels from the bayer image and applied to channels in R,G,B order. To apply padding, i left the first and last pixels zero and filled the pixel value starting from 1.
3. For each channel, used bilinear interpolation to fill the missing pixel values. I used indexing and slicing of Numpy to do this, since it is much faster than naive for loops.
4. For the pixels in edges, I just used the same ways with other pixels, the only difference is that averaged the zero padding together. This makes the edge value smaller than intended, but it could the code more precise.
5. Finally, I removed the zero padding by slicing the result image from the first pixel to the second last pixel.

These are two images obatained from bilinear interpolation.

Figure 1: Images obtained from bilienear interpolation

The problem in this implementation is as mentioned, the edge control. I think this could be resolved by using other methods for edges, such as reflection.

Fundamental Matrix Calculation

In this section, with given 8 points on both images I obtained the fundamental matrix, using eight-point algorithm.

```
def calculate_fundamental_matrix(pts1, pts2):
    assert pts1.shape[1] == 2 and pts2.shape[1] == 2
    assert pts1.shape[0] == pts2.shape[0]

    n = pts1.shape[0]

    # Normalization
    mean1 = np.mean(pts1, axis=0)
    mean2 = np.mean(pts2, axis=0)

    scale1 = np.sqrt(2) / np.std(pts1)
    scale2 = np.sqrt(2) / np.std(pts2)

    T1 = np.array([
        [scale1, 0, -scale1 * mean1[0]],
        [0, scale1, -scale1 * mean1[1]],
        [0, 0, 1]
    ])

    T2 = np.array([
        [scale2, 0, -scale2 * mean2[0]],
        [0, scale2, -scale2 * mean2[1]],
        [0, 0, 1]
    ])

    pts1 = np.column_stack([pts1, np.ones(n)])
    pts2 = np.column_stack([pts2, np.ones(n)])

    pts1_normalized = (T1 @ pts1.T).T
    pts2_normalized = (T2 @ pts2.T).T

    A = np.zeros((n, 9))
    for i in range(n):
        x1, y1, _ = pts1_normalized[i]
```

```
        x2, y2, _ = pts2_normalized[i]
        A[i] = [x1*x2, x2*y1, x2, y2*x1, y1*y2, y2, x1, y1, 1]

    U, S, Vt = np.linalg.svd(A)
    f = Vt[-1]
    F = f.reshape(3, 3)

    U, S, Vt = np.linalg.svd(F)
    S[-1] = 0
    F = U @ np.diag(S) @ Vt

    # Denormalize
    F = T2.T @ F @ T1

    return F

def transform_fundamental_matrix(F, h1, h2):

    F_mod = np.linalg.inv(h2).T @ F @ np.linalg.inv(h1)

    return F_mod
```

1. **Normalization:** First I normalized the given points using the built-in fuction $normalize\_points()$.
2. **Constructing Matrix A:** A matrix A is constructed using the normalized points. Each row of A is constructed as $[x'_1x_2, x'_1y_2, x'_1, y'_1x_2, y'_1y_2, y'_1, x_2, y_2, 1]$.
3. **Singular Value Decomposition:** SVD is performed on matrix A to find the Fundamental Matrix F.
4. **Constraining F:** As F is only determined up to a scale, and should have rank 2, the smallest singular value is set to zero to enforce this property.
5. **Denormalization:** Finally, the Fundamental Matrix F is denormalized to bring it back to the scale of the original points.

1. Result 1 was a total failure, because...
2. Result 2 (Figure 2, left) was surprising, because...
3. Result 3 (Figure 2, right) blew my socks off, because...

My results are summarized in Table 1.

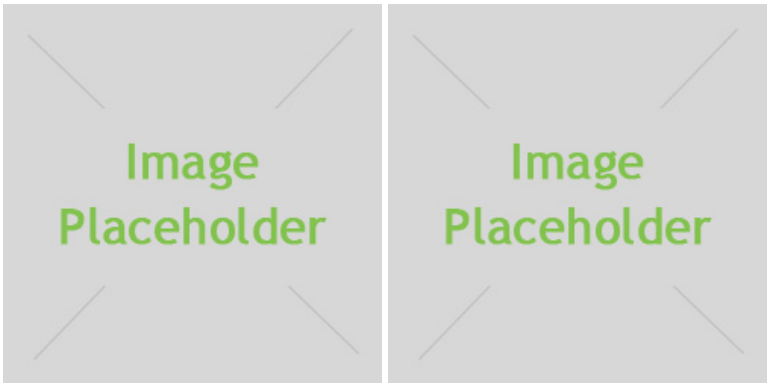

Figure 2: *Left:* My result was spectacular. *Right:* Curious.

Condition	Time (seconds)
Test 1	1
Test 2	1000

Table 1: Stunning revelation about the efficiency of my code.