Kevin Le
CSE 803
Hw 1

1. Camera Projection Matrix

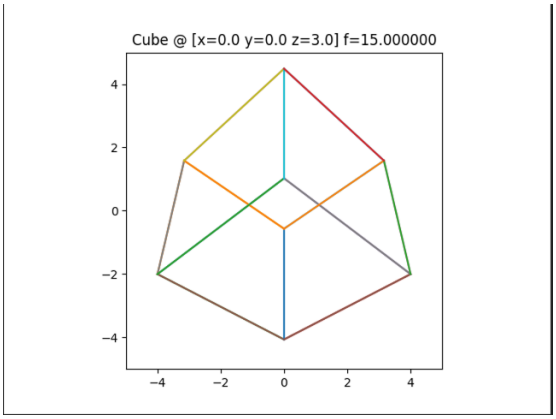
a.) rotY is a function that returns a matrix used as a factor in a dot product with the coordinates of a unit cube. The product of which would create the next coordinate points for the rotation of the cube. The matrix was denoted from the wikipedia resource presented in the homework instructions.

```
32  def rotY(theta):
33      return [[math.cos(theta), 0, math.sin(theta)], [0, 1, 0], [(-1 * math.sin(theta)), 0, math.cos(theta)]]
```

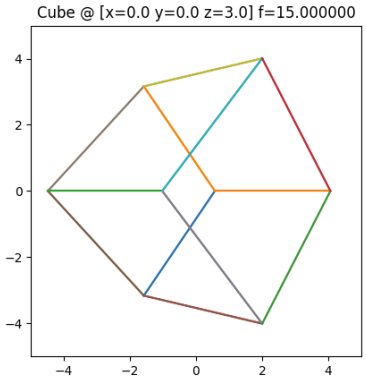
b.) rotX was similar to rotY.

```
35  def rotX(theta):
36      return [[1, 0, 0], [0, math.cos(theta), -1 * math.sin(theta)], [0, math.sin(theta), math.cos(theta)]]
```

With theta being $\frac{\Pi}{4}$, rotating X then Y by theta, would get this picture:



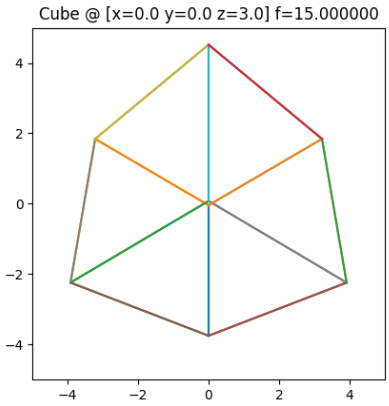
Conversely, rotating Y then X by theta would get this picture:



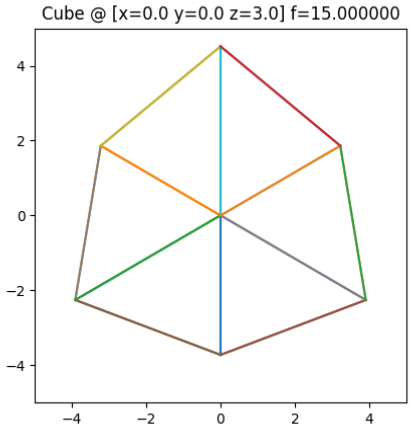
It should be obvious that matrix multiplication is not commutative. To have it rotate X, then Y, you would have to multiply the rotation X matrix by the rotation Y matrix. The reverse of the factors (rotate Y then X) would give a completely different ending result, shown with the previous pictures.

Kevin Le
CSE 803
Hw 1

c.) After seeing the rotation off the cube by $\frac{\Pi}{4}$ by X then Y. It was determined that the cube rotated too much along the x-axis. By reducing the amount of rotation to $\frac{\Pi}{5}$, it can be seen that the cube is very close to being aligned, with there being a little gap showing it is still a little over rotated along the x axis:



By slightly increasing the denominator by 0.1, we could then lower the amount of rotation that tiny amount and get the aligned corners as instructed:



Thus, the X rotation should be $\frac{\Pi}{5.1}$ and the Y rotation should be $\frac{\Pi}{4}$:

```
44    def centerDiag():
45        return np.matmul(rotX(np.pi/5.1), rotY(np.pi/4))
```

Note: This function was called in render cube for the R(rotation) parameter to generate the image

d.) To make an orthogonal view of the cube, the render cube function is first called with the same rotation as 1.c.

```
47  def orthogdraw():
48      fname = "1d"
49      renderCube(f=15, t=(0,0,3), R=centerDiag(), O=True)
50      plt.savefig(fname)
51      plt.close()
```

An “O” parameter is added to the render cube function to create a branch that allows the cube to be drawn in an orthogonal view.

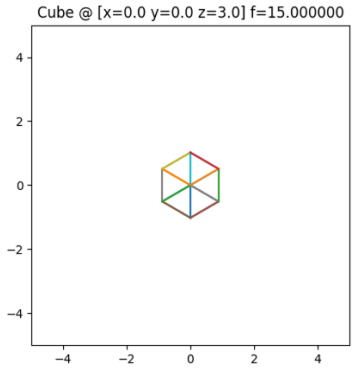
```
80  def renderCube(f=1,scaleFToSize=None,t=(0,0,1),R=np.eye(3), O = False):
81      #Given:
82      #   f -- the focal length
83      #   scaleFToSize -- a target size on the retina (sqrt of area)
84      #   t -- where the cube is with respect to the camera axes
85      #   R -- a rotation
86      #Render the cube
87      L = generateCube()
88      t = np.array(t)
89      if O:
90          pL = projectLinesOrthogonal(f,R,t,L)
91      else:
92          pL = projectLines(f,R,t,L)
93
94      if scaleFToSize is not None:
95      #then adjust f so that the image is the right size
96          xRange, yRange = xyrange(pL)
97          geoMean = (xRange*yRange)**0.5
98          f = (f / geoMean)*scaleFToSize
99          #re-render with the right focal length
100         pL = projectLines(f,R,t,L)
101
102     plt.figure()
103     plt.title("Cube @ [x=%.1f y=%.1f z=%.1f] f=%f" % (t[0],t[1],t[2],f))
104     for i in range(pL.shape[0]):
105         u1, v1, u2, v2 = pL[i,:]
106         print(u1, v1, u2, v2)
107         plt.plot((u1,u2),(v1,v2))
108
109     plt.axis('square')
110     plt.xlim(-5,5); plt.ylim(-5,5)
```

A “projectLinesOrthogonal” function is then made to create a projection with an orthogonal view:

Kevin Le
CSE 803
Hw 1

```
54  def projectLinesOrthogonal(f,R,t,L):
55      #Given:
56      #  Nx6 lines [x1,y1,z1,x2,y2,z2]
57      #  3x3 rotation matrix R
58      #  3x1 translation t
59      #  scalar focal length f
60      #Return:
61      #  Nx4 projection of lines [u1,v1,u2,v2]
62      pL = np.zeros((L.shape[0],4))
63
64      p_matrix = [
65          [1, 0, 0],
66          [0, 1, 0],
67          [0, 0, 0]
68      ]
69      sf = .25
70
71      for i in range(L.shape[0]):
72          #rotate and translate
73          p = np.dot(p_matrix, np.dot(R,L[i,:3])) + t
74          pp = np.dot(p_matrix, np.dot(R,L[i,3:])) + t
75          #apply projection u = x*f/z; v = y*f/z
76          pL[i,:2] = sf * (p[0]*f/p[2]), sf * p[1]*f/p[2]
77          pL[i,2:] = sf * pp[0]*f/pp[2], sf * pp[1]*f/pp[2]
78      return np.vstack(pL)
```

The “p_matrix” is the projection matrix that was denoted in the lecture slides and was put in as a factor for dot product with the rotation seen on lines 73 and 74 of figure _. The sf (scale factor) variable was then estimated to scale the projection down by $\frac{1}{4}$th of its size to get an approximation that matches the problem. By multiplying by sf, we reduce the size of what is rendered giving us:



Another way to change the size of the render would be to change the focal length. By decreasing the focal length, you would simulate the same thing as multiplying by a scale factor to the points. You could also render the cube rotating in the orthogonal view if you change the p_matrix by adding a 1 in the 3rd element of the 2nd and 3rd row. This gif could be found in Answers/Q1/1d_rotation.gif.

Kevin Le
CSE 803
Hw 1

2. Prokudin-Gorskii: Color from grayscale photographs

a.) To combine the grayscale pictures, the picture is first parsed into 3rds shown in the following figure.

```python
def parse(im):
    h, w = im.shape
    cut = h // 3

    #create blue picture
    b_im = im[: cut, : ]

    #create green picture
    g_im = im[cut :cut * 2, : ]

    #create red picture
    r_im = im[cut * 2 : cut * 3, : ]

    return b_im, g_im, r_im
```

Then, the outputs are stacked to with the following function and saved:

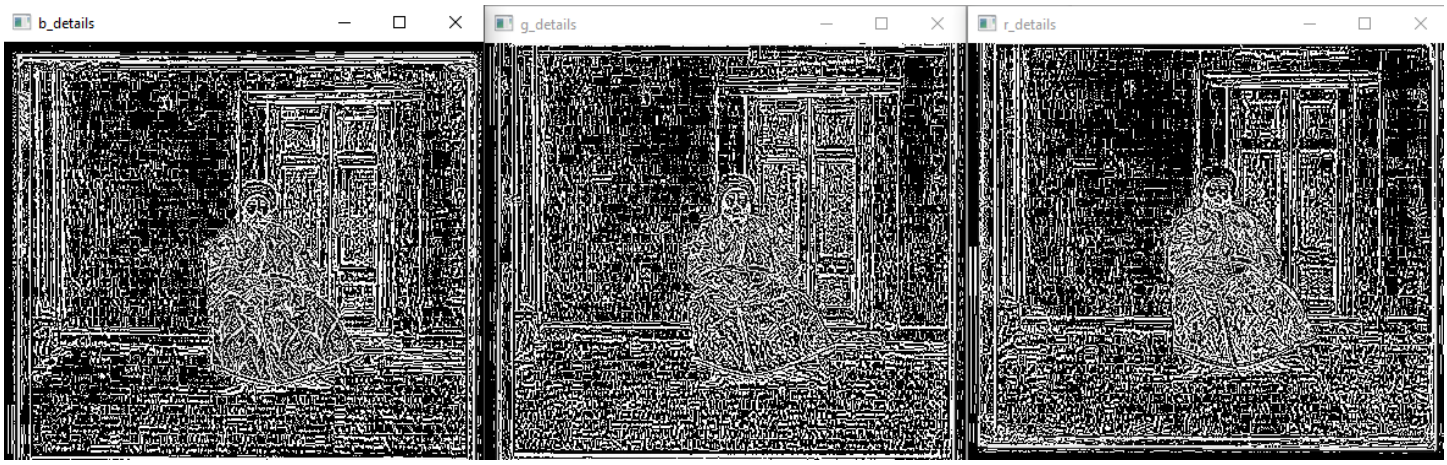
```python
def combine(im_name):
    #set up picture
    p = plt.imread("prokudin-gorskii/" + im_name + ".jpg")

    b_im, g_im, r_im = parse(p)

    combined = np.dstack((r_im, g_im, b_im))
    plt.imsave("Answers/Q2/combined.jpg", combined)
```

The resulting image gives a colored image that is not properly aligned.

b.) To ensure alignment, the following function is used:

```
77  def combine_alignment(im, im_name, write = True, start = [(0,0), (0,0)]):
78      b_im, g_im, r_im = parse(im)
79
80      b_blur = cv.GaussianBlur(b_im, (3,3), 0)
81      b_details =  b_im - b_blur
82
83      g_blur = cv.GaussianBlur(g_im, (3,3), 0)
84      g_details = g_im - g_blur
85
86      r_blur = cv.GaussianBlur(r_im, (3,3), 0)
87      r_details =  r_im - r_blur
88
89      g_im_offset= align(b_details, g_details, start[0])
90      aligned_g_im = roll_XY(g_im, g_im_offset[0], g_im_offset[1])
91
92      r_im_offset= align(b_details, r_details, start[1])
93      aligned_r_im = roll_XY(r_im, r_im_offset[0], r_im_offset[1])
94
95      aligned_im = cv.merge([b_im, aligned_g_im, aligned_r_im])
96
97      if write:
98          cv.imwrite(f"Answers/Q2/aligned_"+ im_name +".jpg", aligned_im)
99      return g_im_offset, r_im_offset
```

Due to contrasts in lighting, alignment may not be exact, like in pictures with the Russian Noble. To mitigate this, a detail map is created on each color channel to normalize the image and align them better. This can be shown on lines similar to 81 in the previous figure.



Then, with the blue channel being the anchor, an alignment function is implemented to find the alignment offset.

```
112 def align(ref_color, color, start):
113     min_val = sys.maxsize
114     offset = start
115     for i in range (start[0] - 15, start[0] + 16):
116         for j in range(start[1] - 15, start[1] + 16):
117             metric = np.sum((roll_XY(color, i, j) - ref_color) ** 2)
118             if metric < min_val:
119                 min_val = metric
120                 offset = (i,j)
121
122     return offset
```

Kevin Le
CSE 803
Hw 1

The alignment searches on the interval of [-15,15]. A starting point is initialized at (0,0), but can be set for pyramid stacking. The alignment uses the sum squared difference (SSD) as a metric to determine what offset gives the smallest SSD for the pixels. The “roll_XY” function is used to shift the channel for the comparison:

```
106  def roll_XY(img, x, y):
107      #roll image via x-axis
108      rolled = np.roll(img, x, axis = 1)
109      #return rolled image via y-axis
110      return np.roll(rolled, y, axis = 0)
```

The channel offsets are then returned as shown in the combine_alignment function on line 99 for usage in building the image pyramids in a later task. The following are the aligned images from running the algorithm.

R Offset: (1, 10)
G Offset: (-1, 5)

R Offset: (2, 9)
G Offset: (2, 4)

R Offset: (4, 14)
G Offset: (2, 7)

R Offset: (1, 13)
G Offset: (0, 4)

R Offset: (4, 11)
G Offset: (3, 5)

R Offset: (1, 5)
G Offset: (0, 0)

Kevin Le
CSE 803
Hw 1

R Offset: (-5, 0)
G Offset: (5, 0)

Kevin Le
CSE 803
Hw 1

c.) The pyramid uses all the previous functions but first reduces the size of the image and feeds into the alignment function to create an intermediate offset. This offset is then fed into the full resolution alignment to set a starting point for alignment of the whole picture, which could be seen on line 115 and 116 of the align function.

```
128 def pyramid(im_name):
129     p = cv.imread(im_name + ".jpg")
130 
131     # Convert for correct shape and color channels
132     p = cv.cvtColor(p, cv.COLOR_BGR2GRAY)
133 
134     h, w = p.shape
135     p_low = cv.resize(p, (w//2,h//2))
136     g_offset_lower, r_offset_lower = combine_alignment(p_low, im_name + "_lower", False)
137     print(f"Coarse G Channel Offset: {g_offset_lower}")
138     print(f"Coarse R Channel Offset: {r_offset_lower}")
139 
140     g_offset_full, r_offset_full = combine_alignment(p, im_name, True, [g_offset_lower, r_offset_lower] )
141     print(f"Full Resolution G Channel Offset: {g_offset_full}")
142     print(f"Full Resolution R Channel Offset: {r_offset_full}")
```

The following are the output for the seoul and vancouver_tableau

Coarse R Offset: (0, 1), Full Resolution R Offset: (-1, 2)
Coarse G Offset: (3, 1), Full Resolution G Offset: (5, 2)

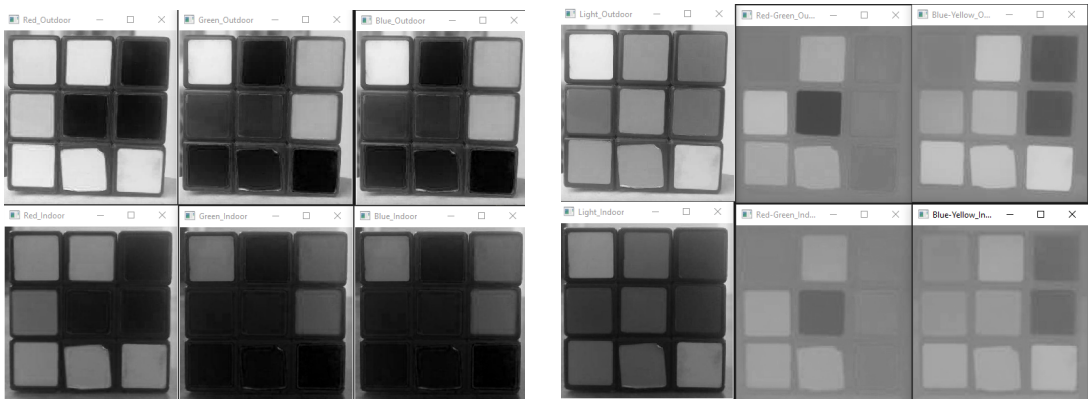
Coarse R Offset: (-6, -5), Full Resolution R Offset: (10, -9)
Coarse G Offset: (5, -5), Full Resolution G Offset: (12, -8)

Kevin Le
CSE 803
Hw 1

3. Color Spaces and illuminance

a.) The indoor and outdoor picture was loaded in and split into RGB and LAB color spaces with the following code with the following outputs:

```
149  def split_plot():
150      p = cv.imread("indoor.png")
151      b_im, g_im, r_im = cv.split(p)
152      cv.imshow("Red_Indoor", r_im)
153      cv.imshow("Blue_Indoor", b_im)
154      cv.imshow("Green_Indoor", b_im)
155
156      p = cv.cvtColor(p, cv.COLOR_BGR2LAB)
157      l, a, b = cv.split(p)
158      cv.imshow("Light_Indoor", l)
159      cv.imshow("Red-Green_Indoor", a)
160      cv.imshow("Blue-Yellow_Indoor", b)
161
162      p = cv.imread("outdoor.png")
163      b_im, g_im, r_im = cv.split(p)
164      cv.imshow("Red_Outdoor", r_im)
165      cv.imshow("Blue_Outdoor", b_im)
166      cv.imshow("Green_Outdoor", b_im)
167
168      p = cv.cvtColor(p, cv.COLOR_BGR2LAB)
169      l, a, b = cv.split(p)
170      cv.imshow("Light_Outdoor", l)
171      cv.imshow("Red-Green_Outdoor", a)
172      cv.imshow("Blue-Yellow_Outdoor", b)
173
174      cv.waitKey(0)
```



The top row is the outdoor and the bottom is the indoor. The right figure is the image split into RGB and the left figure is the image split into LAB.

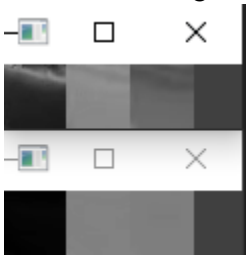
b.) From the outputs of the 2 different color spaces, it's notable that in the RGB split, the images from indoor and outdoor are quite different on each channel. This is mostly due to lighting being captured inside the color value itself. In comparison to LAB, the lighting is captured separately in the "l" (luminance) channel and the "a" and "b" channels are pretty much the same for both pictures. Thus, it's obvious that changes in light are better separated in LAB, being that it has its own designated channel, whereas in RGB, the light data is captured across the various color channels of RGB.

c.) For this task, the object for the image is a boxing glove in a setting next to a white light, and another in a dark room with low yellow light coming from behind. Both pictures were cropped and resized to 256x256.


For im1, the coordinate (60,93) was chosen, on the thumb, where there was a small white reflection. A similar location for im2 was chosen with the coordinate of (55,106). With doing a lab split of the 32x32 patch of the image with the following code:

```
176  def compare():
177      coord1 = (60,93)
178      coord2 = (55,106)
179
180      p1 = cv.imread("Answers/Q3/im1.jpg")
181      p1_patch = p1[coord1[1] : coord1[1]+32, coord1[0] : coord1[0]+32]
182
183      p2 = cv.imread("Answers/Q3/im2.jpg")
184      p2_patch = p2[coord2[1] : coord2[1]+32, coord2[0] : coord2[0]+32]
185
186      #Compare LAB colorspace
187      p1_patch = cv.cvtColor(p1_patch, cv.COLOR_BGR2LAB)
188      p1_l, p1_a, p1_b = cv.split(p1_patch)
189
190      p1_concat = cv.hconcat([p1_l, p1_a, p1_b])
191      cv.imshow("P1 Patch", p1_concat)
192
193      p2_patch = cv.cvtColor(p2_patch, cv.COLOR_BGR2LAB)
194      p2_l, p2_a, p2_b = cv.split(p2_patch)
195
196      p2_concat = cv.hconcat([p2_l, p2_a, p2_b])
197      cv.imshow("P2 Patch", p2_concat)
198
199      cv.waitKey(0)
```

The output from the split is as such, top being im1 and bottom being im2:



Kevin Le
CSE 803
Hw 1

The channels are in LAB order, and in im2, the “a” and “b” channels are very similar, with a small negligible difference, most likely due to the darkness muddling the contrast of the glove. im1 “a” and “b” channels are similar to im2, but there is more of a difference between the “a” and “b” of im1. The reflection on im1 may have been captured in the color data of “a” and “b”. The light channel is the most obvious due to the difference in lighting.