Aidan Fischer CS 532 Homework 4 5/6/2022

I pledge my honor that I have abided by the Stevens Honor System

Code

main.py - Runs all problems

```
# Get indices of maximum n entries in array
def get_n_max_indices(array, n):
    indices = np.zeros((n, len(array.shape)))
    arr = array.copy()
    for i in range(n):
        indmax = np.unravel_index(np.argmax(arr), array.shape)
        indices[i] = np.array(indmax)
        arr[indmax] = np.min(arr)
    return indices

# Take ONLY the entries in array specified by indices
# Leave rest of entries in output at 0
def indices_select(array, indices):
    out = np.zeros(array.shape)
    for y, x in indices:
        a = int(y)
        b = int(x)
        out[a, b] = array[a, b]
    return out


def main():
    P1_images = [
        Image.get_image("../data/problem1/rgb1.png", True),
        Image.get_image("../data/problem1/rgb2.png", True),
        Image.get_image("../data/problem1/rgb3.png", True)
    ]
    P1_corners = [
        nms(harris(P1_images[0])),
        nms(harris(P1_images[1])),
        nms(harris(P1_images[2]))
    ]
    P1_depthmaps = [
        Image.get_image("../data/problem1/depth1.png"),
        Image.get_image("../data/problem1/depth2.png"),
        Image.get_image("../data/problem1/depth3.png")
    ]
    # Get top 100 corners
    P1_corners = [
        indices_select(P1_corners[0], get_n_max_indices(P1_corners[0],
100)),
        indices_select(P1_corners[1], get_n_max_indices(P1_corners[1],
100)),
        indices_select(P1_corners[2], get_n_max_indices(P1_corners[2],
100))
    ]
    # Get depths
    P1_depths = [
        corners_to_depths(P1_corners[0], Filter.crop_to(P1_depthmaps[0],
P1_corners[0]), int((P1_depthmaps[0].shape[0] - P1_corners[0].shape[0]) /
2)),
        corners_to_depths(P1_corners[1], Filter.crop_to(P1_depthmaps[1],
P1_corners[1]), int((P1_depthmaps[1].shape[0] - P1_corners[1].shape[0]) /
2)),
        corners_to_depths(P1_corners[2], Filter.crop_to(P1_depthmaps[2],
P1_corners[2]), int((P1_depthmaps[2].shape[0] - P1_corners[2].shape[0]) /
2))
    ]
    P1_ranktransform = [
        Filter.crop_to(rank_transform(P1_images[0]), P1_corners[0]),
        Filter.crop_to(rank_transform(P1_images[1]), P1_corners[1]),
        Filter.crop_to(rank_transform(P1_images[2]), P1_corners[2])
    ]
    P1_cornermatch_21 = match_corners(P1_ranktransform[1],
                                      P1_depths[1],
                                      P1_ranktransform[0],
                                      P1_depths[0])
    P1_cornermatch_23 = match_corners(P1_ranktransform[1],
                                      P1_depths[1],
                                      P1_ranktransform[2],
                                      P1_depths[2])
    P1_cornermatch_21 = sorted(P1_cornermatch_21, key = lambda tup:
tup[2])[:10]
```

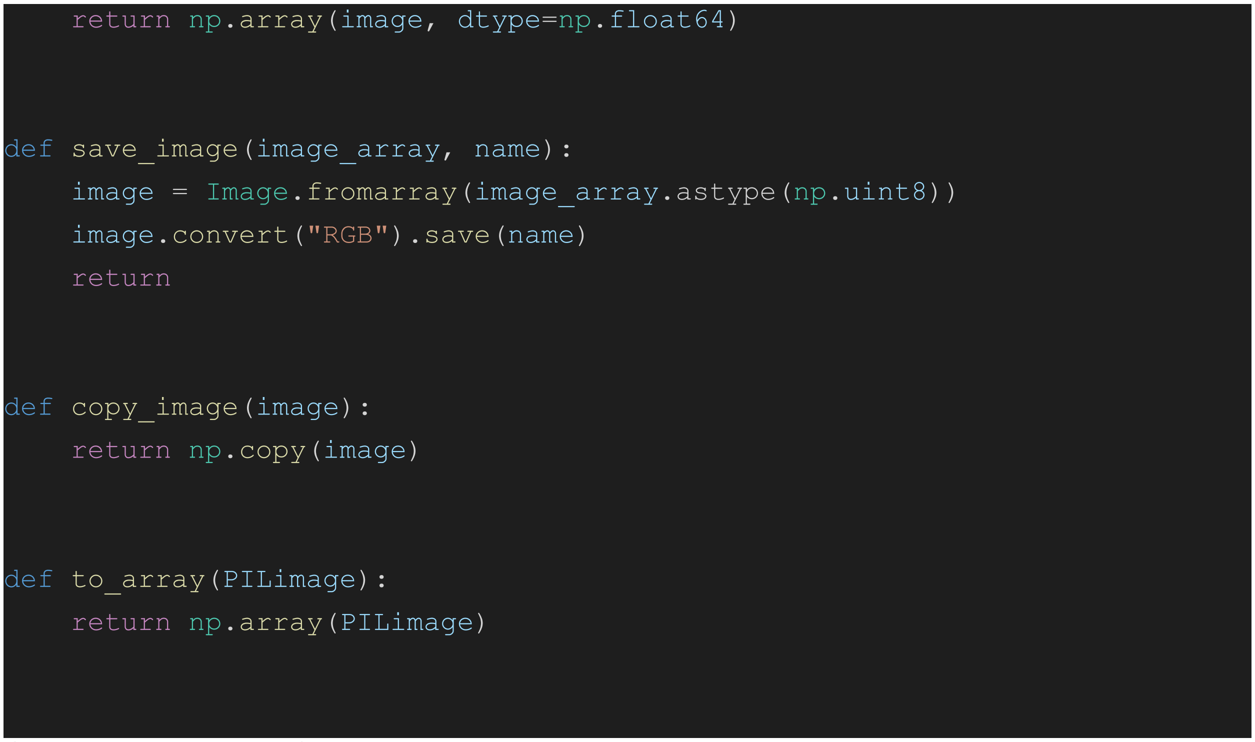
```
    P1_cornermatch_23 = sorted(P1_cornermatch_23, key = lambda tup:
tup[2])[:10]
    R12, t12 = RANSAC(P1_cornermatch_21)
    R32, t32 = RANSAC(P1_cornermatch_23)
    P1_coloredimages = [
        Image.get_image("../data/problem1/rgb1.png"),
        Image.get_image("../data/problem1/rgb2.png"),
        Image.get_image("../data/problem1/rgb3.png")
    ]
    P1_depths_colored = [
        color_map(P1_coloredimages[0], to_depths(P1_images[0],
P1_depthmaps[0], 0)),
        color_map(P1_coloredimages[1], to_depths(P1_images[1],
P1_depthmaps[1], 0)),
        color_map(P1_coloredimages[2], to_depths(P1_images[2],
P1_depthmaps[2], 0)),
    ]
    P1_depths_colored_transformed = [
        image_map(P1_depths_colored[0], R12, t12),
        P1_depths_colored[1],
        image_map(P1_depths_colored[2], R32, t32)
    ]
    ply_write(P1_depths_colored_transformed, "../Output/model.ply")
    P2_image = Image.get_image("../data/problem2/rgbn.png")
    P2_depthmap = Image.get_image("../data/problem2/depthn.png")
    P2_depths = to_depths(P2_image, P2_depthmap, 0)
    P2_normals = compute_normals(P2_depths, P2_image)
    Image.save_image(P2_normals, "../output/normals.png")



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

Image.py - image abstraction

```
def get_image(path, gray=False):
    image = Image.open(path)
    if gray:
        image = ImageOps.grayscale(image)
```



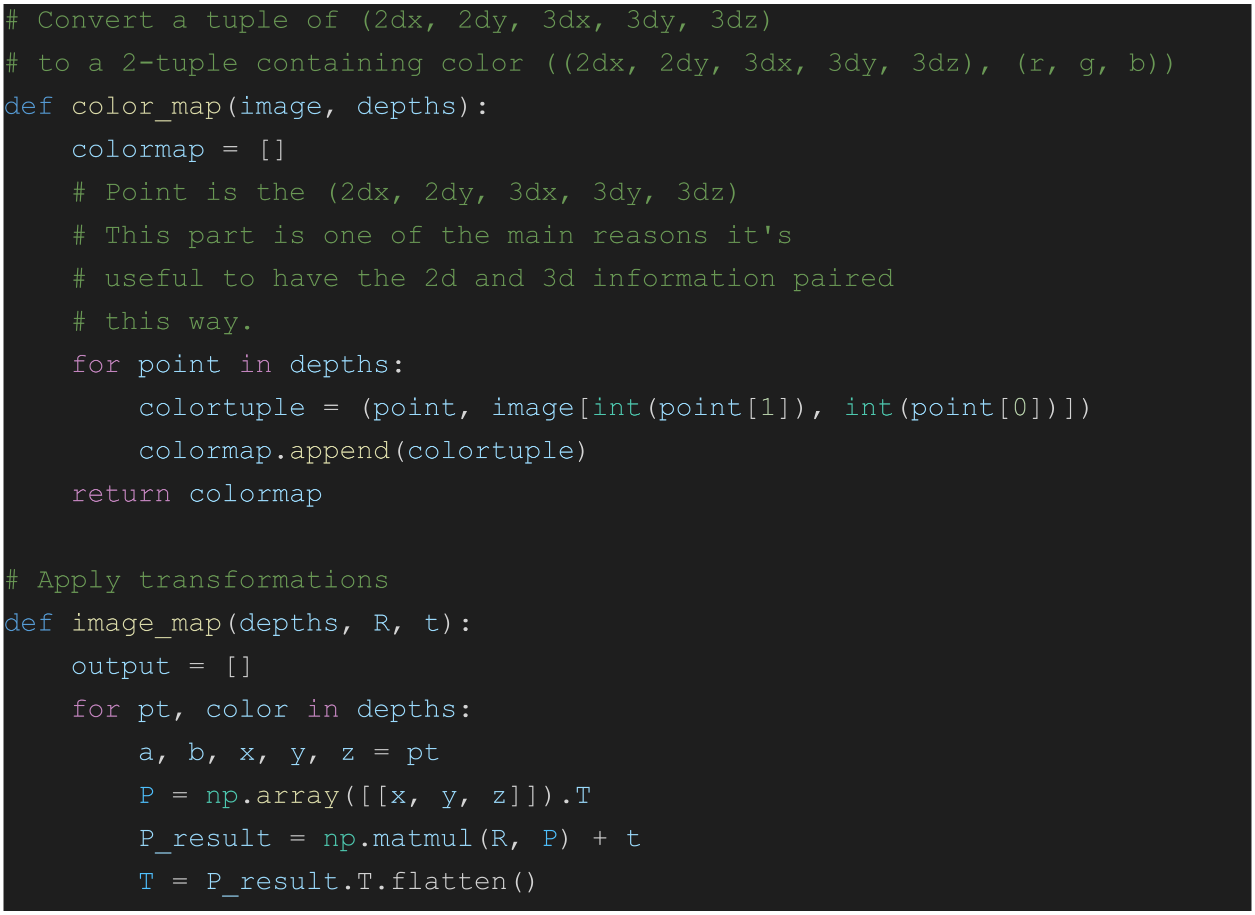
```
    return np.array(image, dtype=np.float64)


def save_image(image_array, name):
    image = Image.fromarray(image_array.astype(np.uint8))
    image.convert("RGB").save(name)
    return


def copy_image(image):
    return np.copy(image)


def to_array(PILimage):
    return np.array(PILimage)
```

Colors.py - Problem 1 file that handles colors in the ply output



```
# Convert a tuple of (2dx, 2dy, 3dx, 3dy, 3dz)
# to a 2-tuple containing color ((2dx, 2dy, 3dx, 3dy, 3dz), (r, g, b))
def color_map(image, depths):
    colormap = []
    # Point is the (2dx, 2dy, 3dx, 3dy, 3dz)
    # This part is one of the main reasons it's
    # useful to have the 2d and 3d information paired
    # this way.
    for point in depths:
        colortuple = (point, image[int(point[1]), int(point[0])])
        colormap.append(colortuple)
    return colormap

# Apply transformations
def image_map(depths, R, t):
    output = []
    for pt, color in depths:
        a, b, x, y, z = pt
        P = np.array([[x, y, z]]).T
        P_result = np.matmul(R, P) + t
        T = P_result.T.flatten()
        output.append(((a, b, T[0], T[1], T[2]), color))
    return output
```

Depth.py - file used in both problems, converts an image-depthmap pair to a list of (2dx, 2dy, 3dx, 3dy, 3dz)

```
# Depth calculation
S = 5000

K = np.matrix([
    [525.0, 0, 319.5],
    [0, 525.0, 239.5],
    [0, 0, 1]
])

# Converts corners to 5-tuples
# [2dx, 2dy, 3dx, 3dy, 3dz]
# Useful to have both 2d and 3d information available in later steps
# offset included because changing image sizes puts reference frame
# out of original. Equal to (orig size - new size) / 2
# offset used only in context of depth estimation
def corners_to_depths(corners, depthmap, offset):
    pixels = np.argwhere(corners)
    pts = np.zeros((100,5), dtype=np.float64)
    i = 0
    for y, x in pixels:
        depth = depthmap[y, x]
        if depth == 0:
            continue
        pt = 1 / S * depth * np.matmul(np.linalg.inv(K),np.matrix([x +
offset, y + offset, 1]).T)
        pts[i] = [int(x), int(y), pt.T.flat[0], pt.T.flat[1],
pt.T.flat[2]]
        i += 1
    return pts[:i]

def to_depths(image, depthmap, offset):
    pts = np.zeros((image.shape[0] * image.shape[1],5), dtype=np.float64)
    i = 0
    for y in trange(image.shape[0], leave=False, desc="Determining all
depths"):
        for x in range(image.shape[1]):
            depth = depthmap[y, x]
            if depth == 0:
                continue
            pt = 1 / S * depth * np.matmul(np.linalg.inv(K),np.matrix([x +
offset, y + offset, 1]).T)
            pts[i] = [int(x), int(y), pt.T.flat[0], pt.T.flat[1],
pt.T.flat[2]]
            i += 1
    return pts[:i]
```

Filter.py - applies filters on images

```
# Define filters used

Ix = np.matrix([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]
])

Iy = Ix.T

# Used to reduce size by ignoring outer pixels

gaussian = 1 / 273 * np.matrix([
    [1, 4, 7, 4, 1],
    [4, 16, 26, 16, 4],
    [7, 26, 41, 26, 7],
    [4, 16, 26, 16, 4],
    [1, 4, 7, 4, 1]
])

# Define filter function

def filter_im(input, filter):
    offset = int((filter.shape[0] - 1) / 2)
    output = np.zeros((input.shape[0] - 2 * offset, input.shape[1] - 2 *
offset))
```

```
    for y in trange(offset, input.shape[0] - offset, desc="Filtering", leave=False):
        for x in range(offset, input.shape[1] - offset):
            output[y - offset, x - offset] = np.sum(np.multiply(input[y - offset: y + offset+1, x - offset: x + offset + 1], filter))
    return output

# crop image to size of other.
# Difference must be even
def crop_to(image, other):
    s1 = other.shape[0]
    s2 = image.shape[0]
    diff = s2 - s1
    if diff < 0 or diff % 2 == 1:
        print("Other image not smaller or difference not even.")
        return
    output = image.copy()
    diff = int(diff / 2)
    output = output[diff: -diff, diff: -diff]
    return output
```

Harris.py - Implements Harris corner detector

```
# Harris corner detection
def harris(image):
    prog = tqdm(total=13, desc="Applying Filters")
    # Compute derivatives
    Ix = Filter.filter_im(image, Filter.Ix)
    prog.update()
    Iy = Filter.filter_im(image, Filter.Iy)
    prog.update()
    Ix2 = Filter.filter_im(Ix, Filter.Ix)
    prog.update()
    Iy2 = Filter.filter_im(Iy, Filter.Iy)
    prog.update()
    Ixy = Filter.filter_im(Ix, Filter.Iy)
    prog.update()
    # Apply gaussians
    Ix = Filter.filter_im(Ix, Filter.gaussian)
    prog.update()
    Iy = Filter.filter_im(Iy, Filter.gaussian)
    prog.update()
    Ix2 = Filter.filter_im(Ix2, Filter.gaussian)
    prog.update()
    Iy2 = Filter.filter_im(Iy2, Filter.gaussian)
    prog.update()
    Ixy = Filter.filter_im(Ixy, Filter.gaussian)
    prog.update()
    # Reduce sizes to smallest to get same shapes
    Ix = Filter.crop_to(Ix, Ixy)
    prog.update()
    Iy = Filter.crop_to(Iy, Ixy)
    prog.update()
    im = image.copy()
    im = Filter.crop_to(im, Ixy)
    prog.update()
    # Because the gaussians are already applied,
    # The second moment matrices are embedded in
    # the variables.
    R = np.zeros(im.shape)
    for y in range(im.shape[0]):
        for x in range(im.shape[1]):
            M = np.matrix([
                [Ix2[y, x], Ixy[y ,x]],
                [Ixy[y, x], Iy2[y, x]]
            ])
            R[y, x] = np.linalg.det(M) - 0.05 * (np.trace(M) ** 2)
            # Threshold
            R[y, x] = R[y, x] if R[y, x] > 10 else 0
    return R
```

MatchCorners.py - implements corner matching

```
# Match corners, using SAD on rank transformed images
# r1, r2 = rank transformed images
# c1, c2 = list of corners with depth
def match_corners(r1, c1, r2, c2):
    output = []
    for x1, y1, x31, y31, z31 in c1:
        x1 = int(x1)
        y1 = int(y1)
        min_dist = None
        min_corner = None
        for x2, y2, x32, y32, z32 in c2:
            x2 = int(x2)
            y2 = int(y2)
            dist = np.sum(np.abs(r1[y1 - 5: y1 + 6, x1 - 5: x1 + 6] - \
                                 r2[y2 - 5: y2 + 6, x2 - 5: x2 + 6]))
            if min_dist is None or dist < min_dist:
                min_dist = dist
                min_corner = (x2, y2, x32, y32, z32)
        output.append([(x1, y1, x31, y31, z31), min_corner, min_dist])
    return output
```

NMS.py - non-maximum suppression function

```
# Non-maximum suppression
def nms(image):
    output = np.zeros((image.shape[0] - 2, image.shape[1] - 2))
    for y in trange(1, image.shape[0] - 1, desc="Non-maximum suppression",
leave=False):
        for x in range(1, image.shape[1] - 1):
            pixel = image[y, x]
            output[y - 1, x - 1] = pixel if pixel == np.max(image[y - 1: y
+ 2, x - 1: x + 2]) else 0
    return output
```

ply.py - writes problem 1 output

```
# Output the ply file

def ply_write(transformed_colored_pts, filename):
    total = len(transformed_colored_pts[0]) + \
            len(transformed_colored_pts[1]) + \
            len(transformed_colored_pts[2])
    with open(filename, "w+") as plyf:
        plyf.write("ply\n")
        plyf.write("format ascii 1.0\n")
        plyf.write(f"element vertex {total}\n")
        plyf.write("property float x\n")
        plyf.write("property float y\n")
        plyf.write("property float z\n")
        plyf.write("property uchar red\n")
        plyf.write("property uchar green\n")
        plyf.write("property uchar blue\n")
        plyf.write("element face 0\n")
        plyf.write("end_header\n")
        for i in range(3):
            for j in range(len(transformed_colored_pts[i])):
                pt, color = transformed_colored_pts[i][j]
                _, _, X, Y, Z = pt
                r, g, b = color
                plyf.write("{0} {1} {2} {3} {4} {5}\n".format(float(X),
float(Y),
                                                              float(Z),
int(r),
                                                              int(g),
int(b)))
        plyf.write("\n")
```

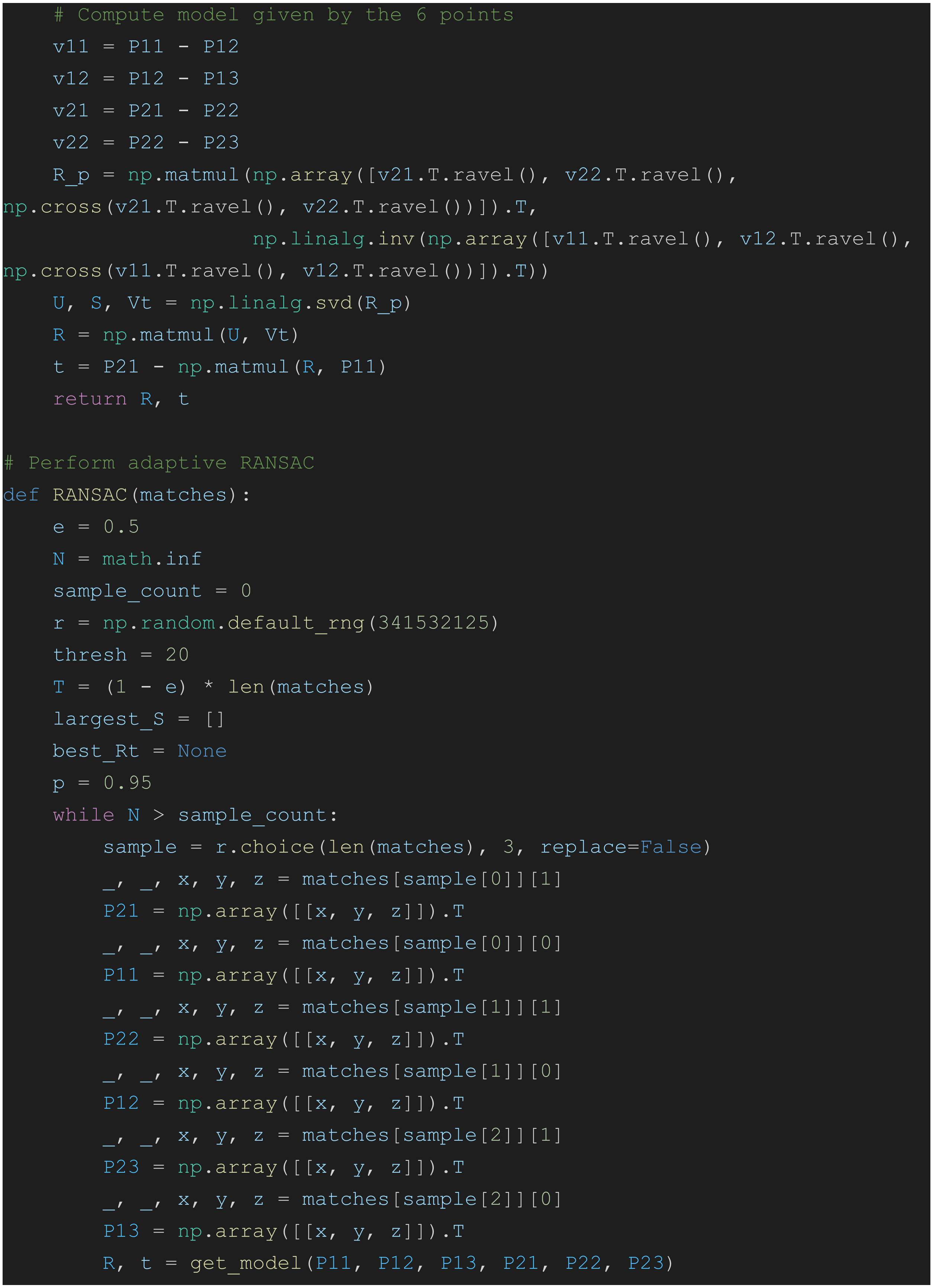
Rank.py - rank transform function

```
# Compute the rank transform
def rank_transform(image):
    output = np.zeros((image.shape[0] - 4, image.shape[1] - 4))
    for y in range(2, image.shape[0] - 2):
        for x in range(2, image.shape[1] - 2):
            output[y - 2, x - 2] = np.sum(image[y - 2: y + 3, x - 2: x +
3] < image[y, x])
    return output
```

RANSAC.py - implements RANSAC

```
def distance(R, t, P1, P2):
    # Get expected P2 from given P1, R, t, then determine
    # distance from actual P2
    P2_p = np.matmul(R, P1) + t
    dist = math.sqrt(math.pow(P2[0, 0] - P2_p[0, 0], 2) + \
                     math.pow(P2[1, 0] - P2_p[1, 0], 2) + \
                     math.pow(P2[2, 0] - P2_p[2, 0], 2))
    return dist

def get_model(P11, P12, P13, P21, P22, P23):
```



```
    # Compute model given by the 6 points
    v11 = P11 - P12
    v12 = P12 - P13
    v21 = P21 - P22
    v22 = P22 - P23
    R_p = np.matmul(np.array([v21.T.ravel(), v22.T.ravel(),
np.cross(v21.T.ravel(), v22.T.ravel())]).T,
                    np.linalg.inv(np.array([v11.T.ravel(), v12.T.ravel(),
np.cross(v11.T.ravel(), v12.T.ravel())]).T))
    U, S, Vt = np.linalg.svd(R_p)
    R = np.matmul(U, Vt)
    t = P21 - np.matmul(R, P11)
    return R, t

# Perform adaptive RANSAC
def RANSAC(matches):
    e = 0.5
    N = math.inf
    sample_count = 0
    r = np.random.default_rng(341532125)
    thresh = 20
    T = (1 - e) * len(matches)
    largest_S = []
    best_Rt = None
    p = 0.95
    while N > sample_count:
        sample = r.choice(len(matches), 3, replace=False)
        _, _, x, y, z = matches[sample[0]][1]
        P21 = np.array([[x, y, z]]).T
        _, _, x, y, z = matches[sample[0]][0]
        P11 = np.array([[x, y, z]]).T
        _, _, x, y, z = matches[sample[1]][1]
        P22 = np.array([[x, y, z]]).T
        _, _, x, y, z = matches[sample[1]][0]
        P12 = np.array([[x, y, z]]).T
        _, _, x, y, z = matches[sample[2]][1]
        P23 = np.array([[x, y, z]]).T
        _, _, x, y, z = matches[sample[2]][0]
        P13 = np.array([[x, y, z]]).T
        R, t = get_model(P11, P12, P13, P21, P22, P23)
        inliers = []
        for match in matches:
            _, _, x, y, z = match[1]
            P1 = np.array([[x, y, z]]).T
            _, _, x, y, z = match[0]
            P2 = np.array([[x, y, z]]).T
            if distance(R, t, P1, P2) < thresh:
                inliers.append(matches)
        if len(inliers) > T:
            return R, t
        else:
            if len(inliers) > len(largest_S):
                largest_S = inliers
                best_Rt = R, t
        e = 1 - (len(inliers) / len(matches))
        N = math.log(1 - p) / math.log(1 - math.pow(1 - e, sample_count))
        sample_count += 1
        T = (1 - e) * len(matches)
    return best_Rt
```

Normal.py - estimates point normals

```
# Determine normals from (2dx, 2dy, 3dx, 3dy, 3dz)
# Output ((2dx, 2dy, 3dx, 3dy, 3dz), (a, b, c))
def compute_normals(depths, image):
    pts = np.zeros((image.shape[0], image.shape[1], 3))
    # Convert depths list to depths array
    for x, y, X, Y, Z in depths:
        pts[int(y), int(x)] = [X, Y, Z]
    normals = np.zeros((image.shape[0], image.shape[1], 3))
    for y in trange(image.shape[0], desc="Determining normals"):
        for x in range(image.shape[1]):
            area = pts[max(0, y - 3): min(y + 4, image.shape[0] - 1),
max(0, x - 3): min(x + 4, image.shape[1] - 1)]
            c = 0
            # Count number of actual depths available in area
            # Also build point matrix
            A = np.zeros((area.shape[0] * area.shape[1], 3))

            for a in range(area.shape[0]):
                for b in range(area.shape[1]):
                    X, Y, Z = area[a, b]
                    if X == 0 and Y == 0 and Z == 0:
                        continue
                    A[c] = [X, Y, Z]
                    c += 1
            # Abort if less than 3 points
            if c < 3:
                continue
            A = A[:c]
            points = Points(A)
            try:
                plane = Plane.best_fit(points)
            except ValueError:
                # Points colinear
                continue
            normal = np.array([plane.normal.round(3)]).T
            normals[y, x] = (((normal / (2 * np.linalg.norm(normal))) +
np.array([[0.5, 0.5, 0.5]]).T) * 255).T.flatten()
    return normals
```

Problem 2 output