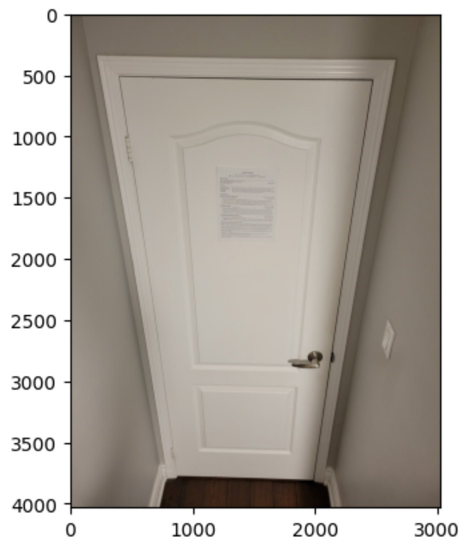
Question 1

```
!pip install opencv-contrib-python

    Requirement already satisfied: opencv-contrib-python in /usr/local/lib/python3.10/dist-packages (4.8.0.76)
    Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from opencv-contrib-python) (1.25.2
```

```
import numpy as np
import cv2
import random
from scipy.ndimage import gaussian_filter
from matplotlib import pyplot as plt
```

```
from google.colab import drive
drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tr
```

```
test = cv2.imread("/content/drive/MyDrive/landscape_1.jpg")
gray_test = cv2.cvtColor(test, cv2.COLOR_BGR2GRAY)

test2 = cv2.imread("/content/drive/MyDrive/landscape_2.jpg")
gray_test2 = cv2.cvtColor(test2, cv2.COLOR_BGR2GRAY)

q1 = cv2.imread("/content/drive/MyDrive/q1.jpg")
gray_q1 = cv2.cvtColor(q1, cv2.COLOR_BGR2GRAY)
print("The height and width of the image is: " + str(gray_q1.shape))
q1_height, q1_width = gray_q1.shape

plt.imshow(cv2.cvtColor(q1, cv2.COLOR_BGR2RGB))

    The height and width of the image is: (4032, 3024)
    <matplotlib.image.AxesImage at 0x7eb4a4e22bf0>
```



```
def get_homography(kps_1, kps_2, matches):
    final_matrix = []
    for _, i, j in matches:
        final_matrix.append(list(kps_1[i].pt) + [1, 0, 0, 0] + [-kps_2[j].pt[0] * kps_1[i].pt[0], -kps_2[j].pt[0] * kps_1[i].pt[
        final_matrix.append([0, 0, 0] + list(kps_1[i].pt) + [1, -kps_2[j].pt[1] * kps_1[i].pt[0], -kps_2[j].pt[1] * kps_1[i].pt[

    final_matrix = np.array(final_matrix, dtype=np.float32)
    eigenvalues, eigenvectors = np.linalg.eig(final_matrix.T @ final_matrix)
    min_eigenvalue_index = np.argmin(eigenvalues)
    min_eigenvalue = eigenvalues[min_eigenvalue_index]
    min_eigenvector = eigenvectors[:, min_eigenvalue_index]

    return min_eigenvector[:9].reshape((3, 3)) / min_eigenvector[:9].reshape((3, 3))[-1][-1]


def get_homography_no_kps(kps_1, kps_2, matches):
    final_matrix = []
    for _, i, j in matches:
        final_matrix.append(list(kps_1[i]) + [1, 0, 0, 0] + [-kps_2[j][0] * kps_1[i][0], -kps_2[j][0] * kps_1[i][1], -kps_2[j][0
        final_matrix.append([0, 0, 0] + list(kps_1[i]) + [1, -kps_2[j][1] * kps_1[i][0], -kps_2[j][1] * kps_1[i][1], -kps_2[j][1

    final_matrix = np.array(final_matrix, dtype=np.float32)
    eigenvalues, eigenvectors = np.linalg.eig(final_matrix.T @ final_matrix)
    min_eigenvalue_index = np.argmin(eigenvalues)
    min_eigenvalue = eigenvalues[min_eigenvalue_index]
    min_eigenvector = eigenvectors[:, min_eigenvalue_index]

    return min_eigenvector[:9].reshape((3, 3)) / min_eigenvector[:9].reshape((3, 3))[-1][-1]


source_points_q1 = [
    [397, 522],
    [2461, 552],
    [868, 3782],
    [1976, 3817]
]
destination_points_q1 = [
    [0, 0],
    [q1_width - 1, 0],
    [0, q1_height - 1],
    [q1_width - 1, q1_height - 1]
]
paper_points = [
    [1186, 1254],
    [1700, 1265],
    [1201, 1857],
    [1670, 1874]
]

matches_q1 = [(None, i, i) for i in range(4)]

homography_q1 = get_homography_no_kps(source_points_q1, destination_points_q1, matches_q1)

transformed_points = cv2.perspectiveTransform(np.array([paper_points], dtype=np.float32), homography_q1)
print("The source points of the paper are:\n", np.array(paper_points, dtype=np.float32).reshape(1, 4, 2), "\n and the destinatic
warped_image = cv2.warpPerspective(q1, homography_q1, (q1_width, q1_height))
plt.imshow(cv2.cvtColor(warped_image, cv2.COLOR_BGR2RGB))
```

```
The source points of the paper are:
 [[[1186. 1254.]
  [1700. 1265.]
  [1201. 1857.]
  [1670. 1874.]]]
 and the destination points of the paper are:
 [[[1124.6738  533.1061]
  [1963.3591  534.4147]
  [1112.0011 1082.3035]
  [1958.499  1089.2811]]]
<matplotlib.image.AxesImage at 0x7eb4a4f9dd50>
```

In the results above, the first 4 points are acquired by taking the top, left, top right, bottom left, bottom right corners of the paper in the imag and transforming the points with the homography acquired. Thus, the first point is the top left corner of the paper in the transformed image, the second is the top right, the third is the bottom left and the fourth is the bottom right. We can do some math to estimate the height and width of the door. Let's create the following relation for the width of the door:

(width of image) / (distance between top two corners of paper) = (width of door) / (width of paper)

$\Rightarrow \frac{3024}{\sqrt{(1124.6738-1963.3591)^2+(533.1061-534.4147)^2}} = \frac{w_{door}}{8.5}$

$\Rightarrow 8.5 \cdot \frac{3024}{\sqrt{(1124.6738-1963.3591)^2+(533.1061-534.4147)^2}} = w_{door}$

$\Rightarrow w_{door} = 30.65 \text{ "}= 77.85cm$

The height of the door can be calculated similarly:

(height of image) / (distance between left two corners of paper) = (height of door) / (height of paper)

$\Rightarrow \frac{4032}{\sqrt{(1124.6738-1112.0011)^2+(533.1061-1082.3035)^2}} = \frac{w_{door}}{11}$

$\Rightarrow 11 \cdot \frac{3024}{\sqrt{(1124.6738-1963.3591)^2+(533.1061-534.4147)^2}} = w_{door}$

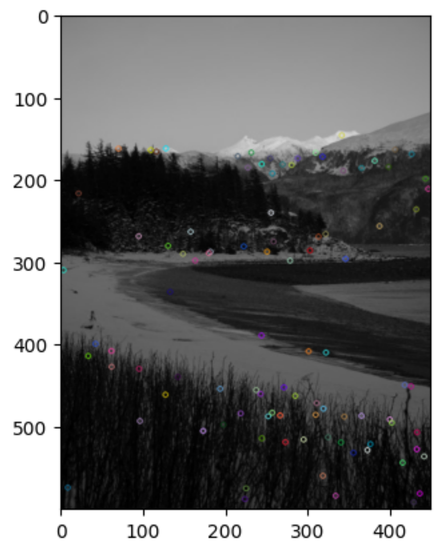
$\Rightarrow w_{door} = 80.74 \text{ "}= 205.08cm$

So the width of the door is $205.08cm$ and the height is $156.87cm$.

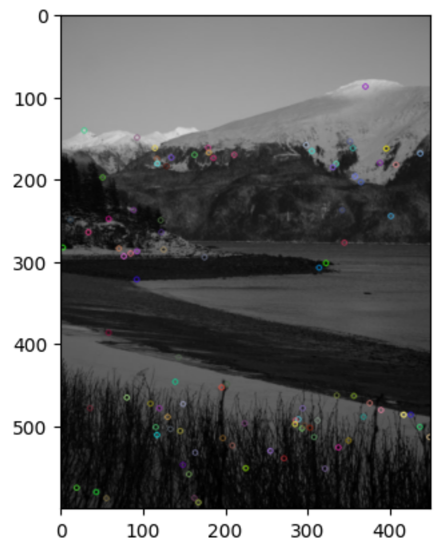
Question 2

```
def visualize_100(kps, des, image):
    kps_reference1, des_reference1 = [], []
    for _ in range(100):
        i = random.randint(0, len(kps) - 1)
        kps_reference1.append(kps[i])
        des_reference1.append(des[i])

    img_kps = cv2.drawKeypoints(image, kps_reference1[:], outImage=None)
    plt.imshow(img_kps)
```

```
sift = cv2.xfeatures2d.SIFT_create()

kps_test, des_test = sift.detectAndCompute(gray_test, None)
visualize_100(kps_test, des_test, gray_test)
```



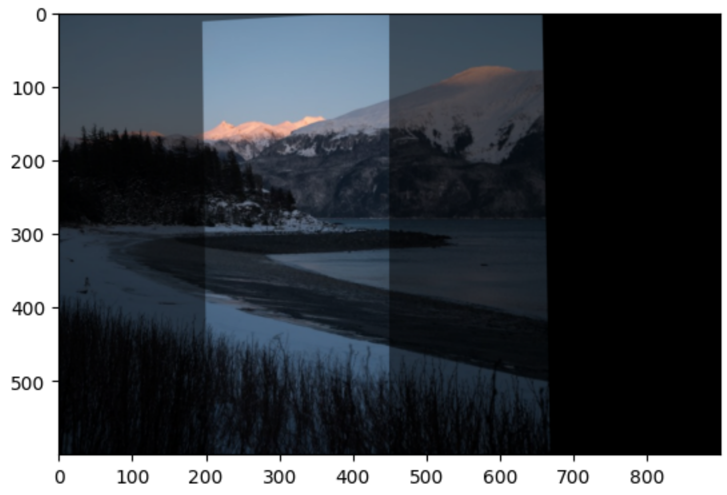
```
kps_test2, des_test2 = sift.detectAndCompute(gray_test2, None)
visualize_100(kps_test2, des_test2, gray_test2)
```



```
def matching(threshold, descriptor_1, descriptor_2):
    matches = []
    for i, descriptor_ref in enumerate(descriptor_1):
        first_best_match_index, second_best_match_index = None, None
        first_best_match, second_best_match = float('inf'), float('inf')

        for j, descriptor_test in enumerate(descriptor_2):
            euclidean_distance = np.linalg.norm(descriptor_ref - descriptor_test)

            if euclidean_distance < first_best_match:
                second_best_match, second_best_match_index = first_best_match, first_best_match_index
                first_best_match, first_best_match_index = euclidean_distance, j
            elif euclidean_distance < second_best_match:
                second_best_match, second_best_match_index = euclidean_distance, j

        phi_ratio = first_best_match / second_best_match
        if phi_ratio < threshold:
            matches.append((first_best_match, i, first_best_match_index))

    return matches


top_matches_1 = matching(0.8, des_test, des_test2)


def ransac(kps_1, kps_2, matches, num_trials, threshold):
    top_inlier = 0
    top_homography = None
    for _ in range(num_trials):
        random_indices = [random.randint(0, len(matches) - 1) for _ in range(4)]
        new_matches = [(None, i, i) for i in range(4)]

        new_matches = [matches[i] for i in random_indices]
        source_points = [kps_1[i] for _, i, _ in matches]
        source_points_1 = [kps_1[i].pt for _, i, _ in matches]
        destination_points = [kps_2[j] for _, _, j in matches]
        destination_points_1 = [kps_2[j].pt for _, _, j in matches]

        homography = get_homography(kps_1, kps_2, new_matches)
        transformed_points = cv2.perspectiveTransform(np.array([source_points_1]), homography)

        inliers = 0
        for match_ in matches:
            source_point = np.array(kps_1[match_[1]].pt)
            destination_point = np.array(kps_2[match_[2]].pt)

            source_point = np.append(source_point, 1)
            destination_point = np.append(destination_point, 1)

            my_destination = homography @ source_point.T
            my_destination = my_destination / my_destination[-1]
            euclidean_distance = np.linalg.norm(my_destination - np.array([destination_point[0], destination_point[1], 1]))
            if euclidean_distance < threshold:
                inliers += 1

        if inliers > top_inlier:
            top_inlier = inliers
            top_homography = homography

    return top_homography


def merge_image_panorama(image_1, image_2, homography):
    height_1, width_1 = image_1.shape[:2]
    height_2, width_2 = image_2.shape[:2]

    warped_image_2 = cv2.warpPerspective(image_2, np.linalg.inv(homography), (width_1 + width_2, height_2)) # size = (width_1 +

    image = np.zeros((height_2, width_1 + width_2, 3), dtype=np.uint8)
    image[:height_1, :width_1] = image_1
    image = cv2.addWeighted(image, 0.5, warped_image_2, 0.5, 0)
    return image
```

```
homography = ransac(kps_test, kps_test2, top_matches_1, 1000, 1)
homography = homography / homography[-1][-1]
print(homography)
merged_image = merge_image_panorama(test, test2, homography)
plt.imshow(cv2.cvtColor(merged_image, cv2.COLOR_BGR2RGB))
```

```
[[ 1.07865214e+00 -1.23999482e-02 -2.10859360e+02]
 [ 6.00340813e-02  1.05924571e+00 -2.49841938e+01]
 [ 1.66045735e-04  1.14457525e-05  1.00000000e+00]]
<matplotlib.image.AxesImage at 0x7eb4a4e215d0>
```



Question 3

Part (a)

The internal camera parameter matrix can be given by the general equation:

$$K = \begin{pmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{pmatrix}$$

Where (p_x, p_y) is the principle point. Subbing in the values given, we have:

$$K = \begin{pmatrix} 0.7215 & 0 & 609.6 \\ 0 & 0.7215 & 172.9 \\ 0 & 0 & 1 \end{pmatrix}$$

Part (b)

We are given the principle point $P = (609.6, 172.9)$, and we also know that the camera is $1.7m$ above the ground. Since we know that the camera plane is orthogonal to the ground plane, we have that vector $\overrightarrow{PQ} = \overrightarrow{n} = (0, -1.7)$ is the normal vector to the ground plane. This will then give that the equation of the plane is simply $y = -1.7$.

Part (c)

We have a point $p = (x, y)$ in 2D, and we know that the point in 3D is on the ground. This allows us to set the value of the z-axis to the height of the camera. In the given question, it is $1.7m$ above the ground, however we can represent this by $z_h = -height$ to generalize the method.

To bring a 3D coordinate to 2D, we have the following mapping:

$$D = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = w \cdot K^{-1} \begin{pmatrix} vp_x \\ vp_y \\ 1 \end{pmatrix}$$

Where K is the internal camera parameter matrix, $(vp_x, vp_y$ is our given (x, y) coordinate.

We know that the point is on the ground plane, which means that our 3D points y coordinate $Y = -1.7$. So we have:

$$\begin{pmatrix} X \\ -1.7 \\ Z \end{pmatrix} = w \cdot \begin{pmatrix} 0.7215 & 0 & 609.6 \\ 0 & 0.7215 & 172.9 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} vp_x \\ vp_y \\ 1 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} X \\ -1.7 \\ Z \end{pmatrix} = w \cdot \begin{pmatrix} 1.386 & 0 & -841.16 \\ 0 & 1.386 & -239.63 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} X \\ -1.7 \\ Z \end{pmatrix} = \begin{pmatrix} w \cdot (1.386x - 841.16) \\ w \cdot (1.386y - 239.63) \\ w \end{pmatrix}$$

$$\Rightarrow w = \frac{-1.7}{1.386y - 239.62}$$

Now that w is solved for, we can say the 3D point is:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} \frac{-1.7 \cdot (1.386x - 841.16)}{(1.386y - 239.63)} \\ -1.7 \\ \frac{-1.7}{1.386y - 239.62} \end{pmatrix}$$

As desired.

Question 4

The pseudocode to compute disparity for a pair of stereo cameras is given by:

1. Assuming the images are on the same epipolar line, calculate the SSD of a feature.
2. Find the minimized SSD value, this gives you the disparity.
3. Do this for each feature in one of the images

The complexity of this algorithm is given my $O(W \cdot H \cdot P^2)$, where W is the width of the left image without loss of generality, H is the height of the left image, and P is the area of the given patch. The patch will be a square.

To compute the depth of each pixel, we can take a pixel (x, y), calculate it's disparaity. Then we can use the formula:

$$Z = \frac{f \cdot T}{d}$$

Where Z is the depth, f is the focal length, and T the baseline distance.

Double-click (or enter) to edit