Segmentation

February 21, 2025

[ ]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment3/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/assignment3
```

1 Clustering and Segmentation

This notebook includes both coding and written questions. Please hand in this notebook file with all the outputs and your answers to the written questions.

This assignment covers K-Means and HAC methods for clustering and image segmentation.

[1]:
```
# Setup
from __future__ import print_function
from time import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from skimage import io

%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 12.0) # set default size of plots
```

```
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
%load_ext autoreload
%autoreload 2
```

1.1 Introduction

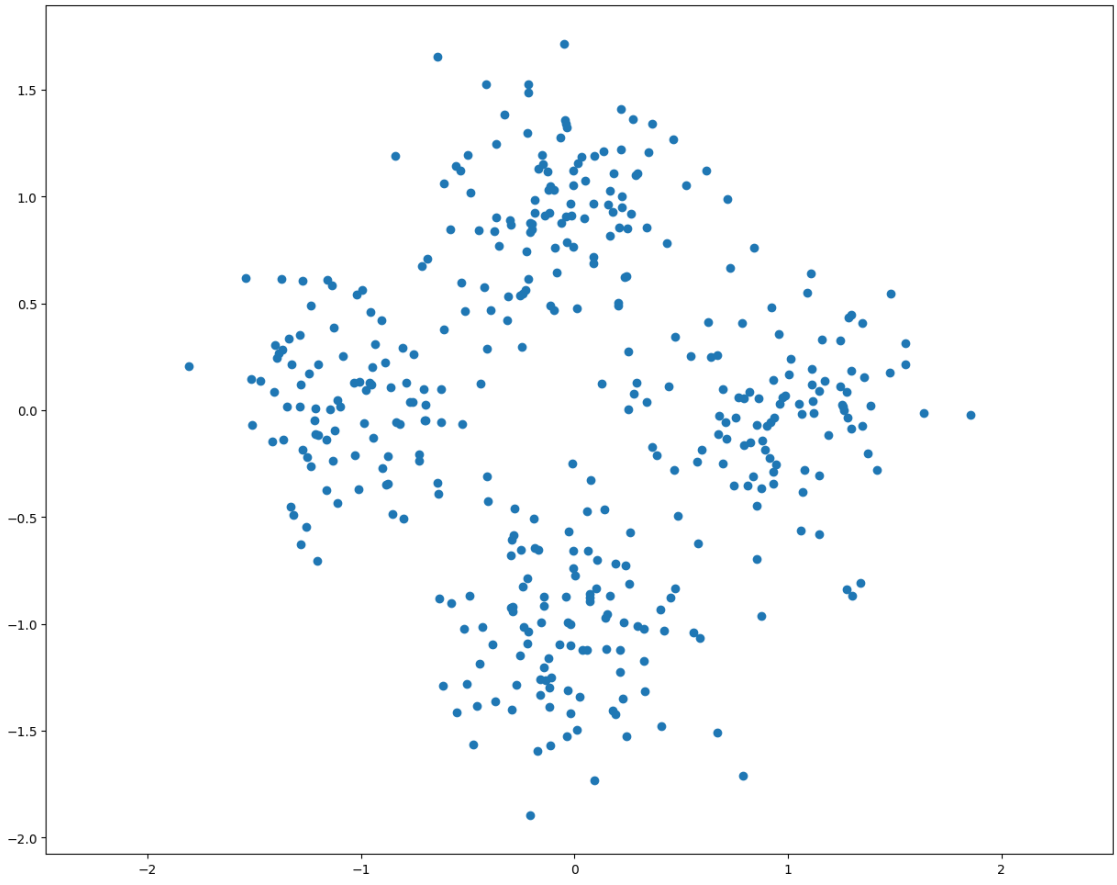
In this assignment, you will use clustering algorithms to segment images. You will then use these segmentations to identify foreground and background objects.

Your assignment will involve the following subtasks: - **Clustering algorithms**: Implement K-Means clustering and Hierarchical Agglomerative Clustering. - **Pixel-level features**: Implement a feature vector that combines color and position information and implement feature normalization. - **Quantitative Evaluation**: Evaluate segmentation algorithms with a variety of parameter settings by comparing your computed segmentations against a dataset of ground-truth segmentations.

1.2 1 Clustering Algorithms (40 points)

[2]:
```
# Generate random data points for clustering

# Set seed for consistency
np.random.seed(0)

# Cluster 1
mean1 = [-1, 0]
cov1 = [[0.1, 0], [0, 0.1]]
X1 = np.random.multivariate_normal(mean1, cov1, 100)

# Cluster 2
mean2 = [0, 1]
cov2 = [[0.1, 0], [0, 0.1]]
X2 = np.random.multivariate_normal(mean2, cov2, 100)

# Cluster 3
mean3 = [1, 0]
cov3 = [[0.1, 0], [0, 0.1]]
X3 = np.random.multivariate_normal(mean3, cov3, 100)

# Cluster 4
mean4 = [0, -1]
cov4 = [[0.1, 0], [0, 0.1]]
X4 = np.random.multivariate_normal(mean4, cov4, 100)

# Merge two sets of data points
X = np.concatenate((X1, X2, X3, X4))
```

```
# Plot data points
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal')
plt.show()
```



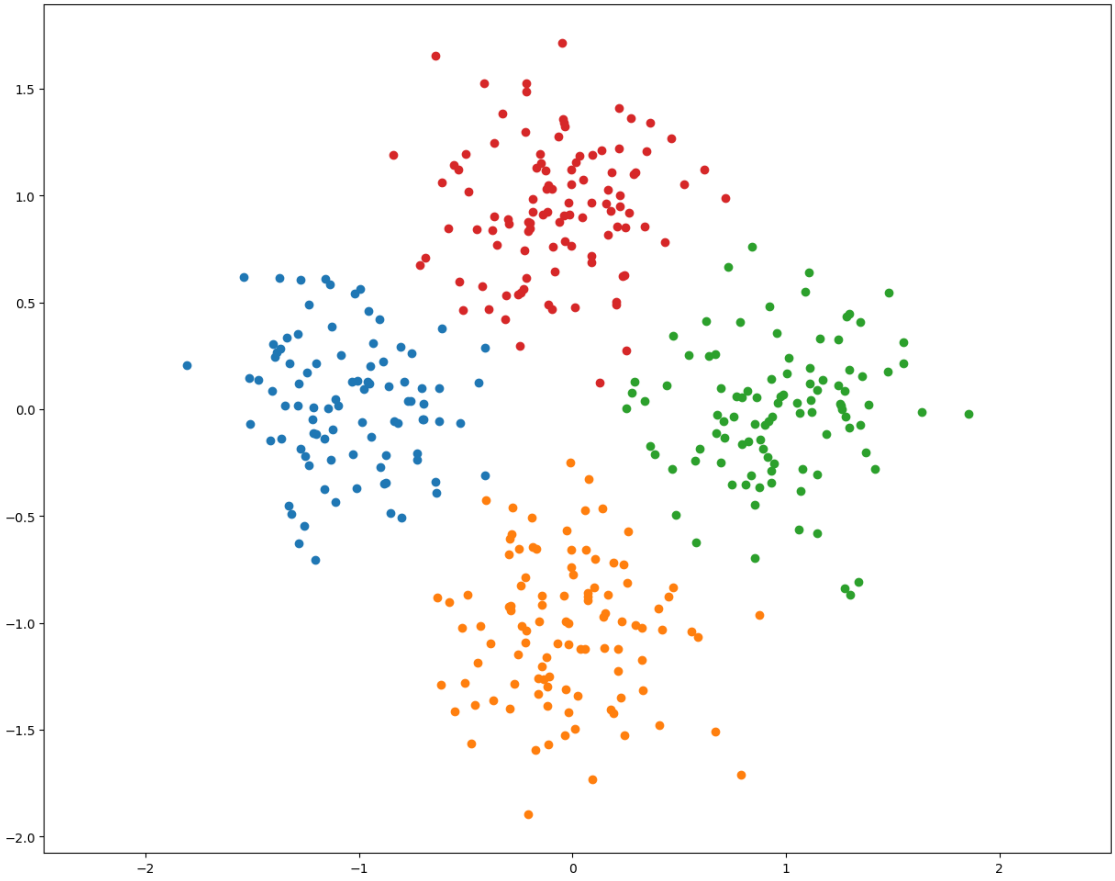
1.2.1 1.1 K-Means Clustering (20 points)

As discussed in class, K-Means is one of the most popular clustering algorithms. We have provided skeleton code for K-Means clustering in the file `segmentation.py`. Your first task is to finish implementing **`kmeans`** in `segmentation.py`. This version uses nested for loops to assign points to the closest centroid and compute a new mean for each cluster.

[3]:
```
from segmentation import kmeans

np.random.seed(0)
start = time()
assignments = kmeans(X, 4)
end = time()
```

```
kmeans_runtime = end - start

print("kmeans running time: %f seconds." % kmeans_runtime)

for i in range(4):
    cluster_i = X[assignments==i]
    plt.scatter(cluster_i[:, 0], cluster_i[:, 1])

plt.axis('equal')
plt.show()
```

```
kmeans running time: 0.379930 seconds.
```

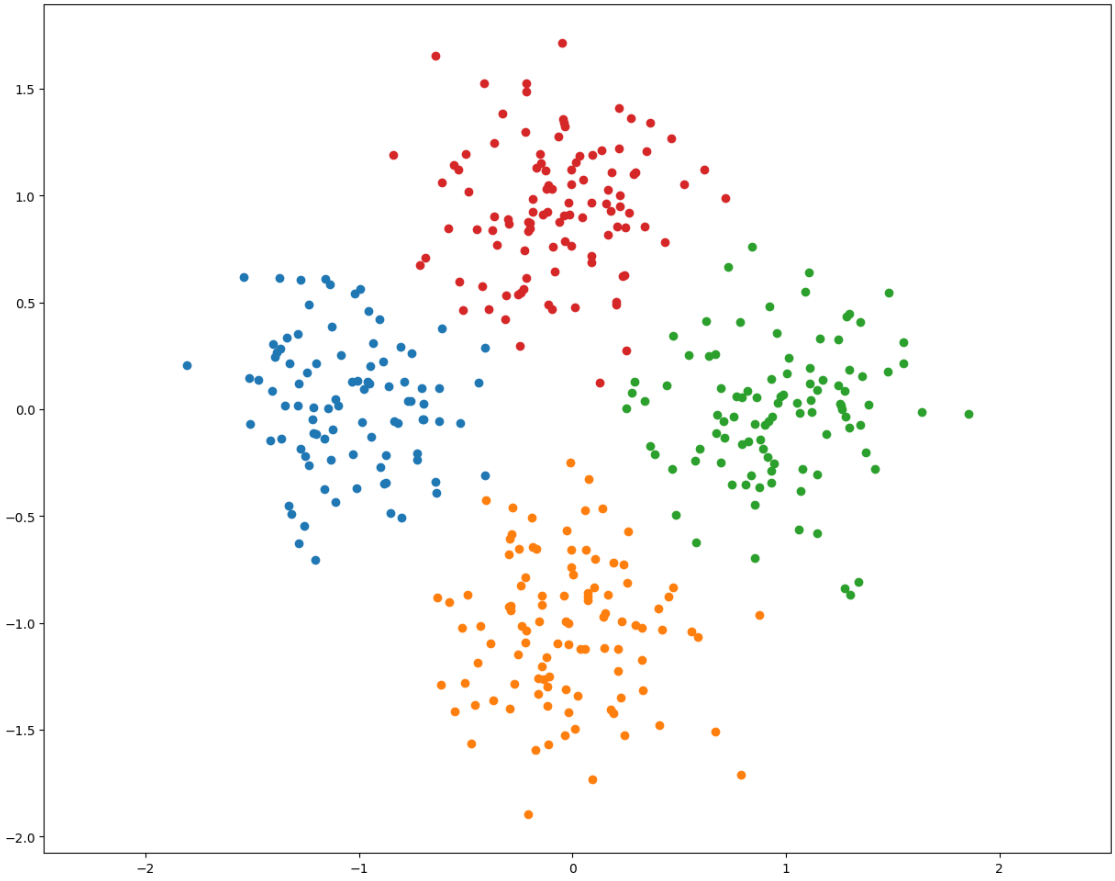


We can use numpy functions and broadcasting to make K-Means faster. Implement **`kmeans_fast`** in `segmentation.py`. This should run at least 10 times faster than the previous implementation.

[4]:
```
from segmentation import kmeans_fast

np.random.seed(0)
start = time()
```

```
assignments = kmeans_fast(X, 4)
end = time()

kmeans_fast_runtime = end - start
print("kmeans running time: %f seconds." % kmeans_fast_runtime)
print("%f times faster!" % (kmeans_runtime / kmeans_fast_runtime))

for i in range(4):
    cluster_i = X[assignments==i]
    plt.scatter(cluster_i[:, 0], cluster_i[:, 1])

plt.axis('equal')
plt.show()
```

```
kmeans running time: 0.006526 seconds.
58.222287 times faster!
```



1.2.2 1.2 K-Means Convergence (10 points)

Implementations of the K-Means algorithm will often have the parameter `num_iters` to define the maximum number of iterations the algorithm should run for. Consider that we opt to not include this upper bound on the number of iterations, and that we define the termination criterion of the algorithm to be when the cost L stops changing.

Recall that L is defined as the sum of squared distance between all points x and their nearest cluster center c:

$$L = \sum_{i \in clusters} \sum_{x \in cluster_i} (x - c_i)^2$$

Show that for any set of points D and any number of clusters k, the K-Means algorithm will terminate in a finite number of iterations.

Your answer here: Given $|D| = n$, there are $\binom{n}{k}$ possible cluster centers, and every point is assigned to precisely one cluster, which gives k^n total cluster assignments for each fixed set of k centers. So there are $\binom{n}{k} \cdot k^n$ total possibilities. K-means iteratively finds the points that are closest to each center, and then after that it re-computes the clusters. So after each iteration, this is decreasing the loss function. It is like a greedy algorithm. However, since there are only $\binom{n}{k} \cdot k^n$ total possibilities, it cannot keep decreasing the loss function for infinite time, so eventually the algorithm terminates.

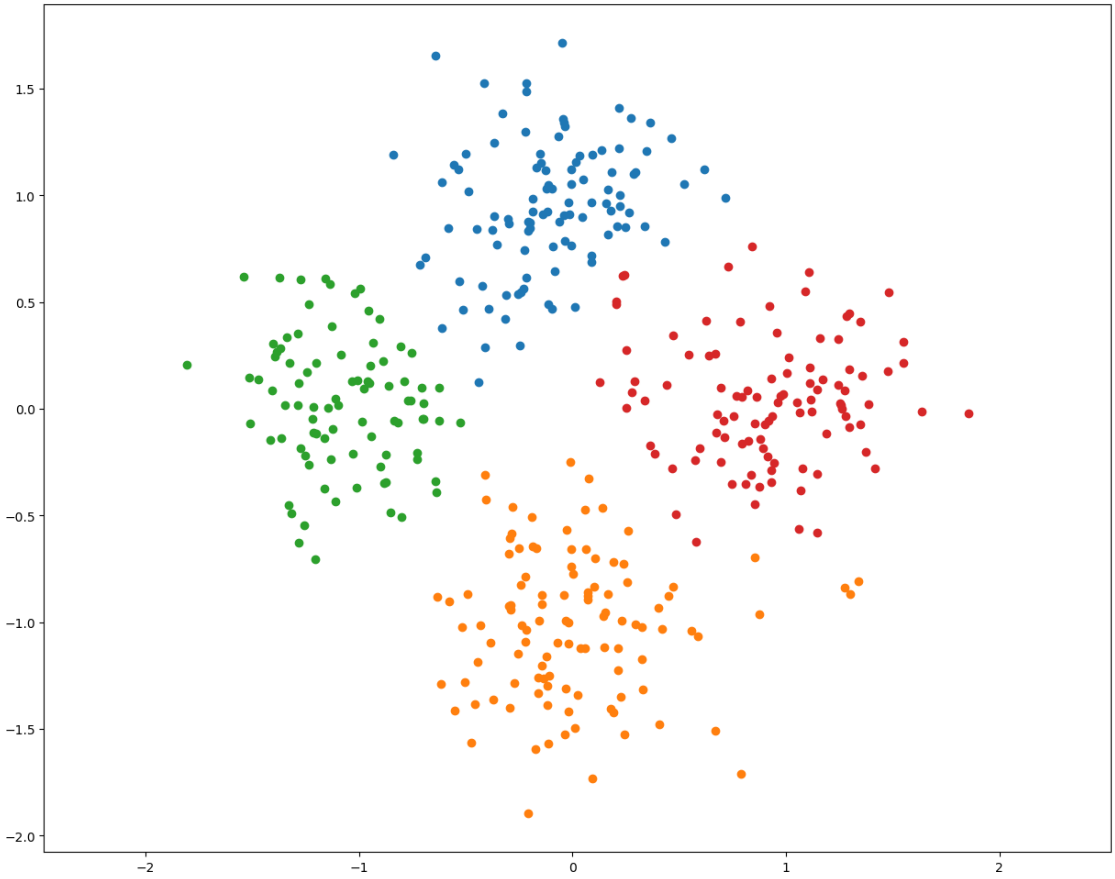
1.2.3 1.2 Hierarchical Agglomerative Clustering (10 points)

Another simple clustering algorithm is Hieararchical Agglomerative Clustering, which is somtimes abbreviated as HAC. In this algorithm, each point is initially assigned to its own cluster. Then cluster pairs are merged until we are left with the desired number of predetermined clusters (see Algorithm 1).

Implement **`hiererachical_clustering`** in `segmentation.py`.

[76]:
```
from segmentation import hierarchical_clustering

start = time()
assignments = hierarchical_clustering(X, 4)
end = time()

print("hierarchical_clustering running time: %f seconds." % (end - start))

for i in range(4):
    cluster_i = X[assignments==i]
    plt.scatter(cluster_i[:, 0], cluster_i[:, 1])

plt.axis('equal')
plt.show()
```

```
hierarchical_clustering running time: 0.041678 seconds.
```



1.3 2 Pixel-Level Features (30 points)

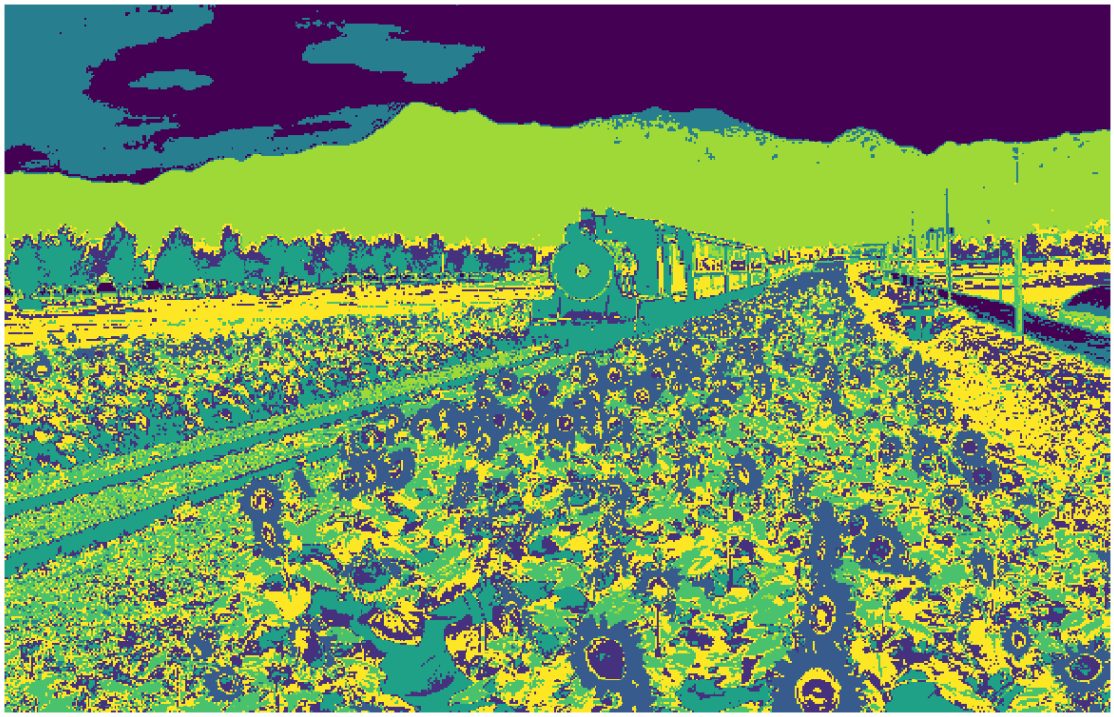
Before we can use a clustering algorithm to segment an image, we must compute some *feature vector* for each pixel. The feature vector for each pixel should encode the qualities that we care about in a good segmentation. More concretely, for a pair of pixels p_i and p_j with corresponding feature vectors f_i and f_j, the distance between f_i and f_j should be small if we believe that p_i and p_j should be placed in the same segment and large otherwise.

[77]:
```
# Load and display image
img = io.imread('train.jpg')
H, W, C = img.shape

plt.imshow(img)
plt.axis('off')
plt.show()
```

1.3.1 2.1 Color Features (15 points)

One of the simplest possible feature vectors for a pixel is simply the vector of colors for that pixel. Implement **`color_features`** in `segmentation.py`. Output should look like the following:

[78]:
```
from segmentation import color_features
np.random.seed(0)

features = color_features(img)

# Sanity checks
assert features.shape == (H * W, C),\
    "Incorrect shape! Check your implementation."

assert features.dtype == np.float64,\
    "dtype of color_features should be float."

assignments = kmeans_fast(features, 8)
segments = assignments.reshape((H, W))

# Display segmentation
plt.imshow(segments, cmap='viridis')
plt.axis('off')
plt.show()
```



In the cell below, we visualize each segment as the mean color of pixels in the segment.

[79]:
```
from utils import visualize_mean_color_image
visualize_mean_color_image(img, segments)
```

1.3.2 2.2 Color and Position Features (15 points)

Another simple feature vector for a pixel is to concatenate its color and position within the image. In other words, for a pixel of color (r, g, b) located at position (x, y) in the image, its feature vector would be (r, g, b, x, y). However, the color and position features may have drastically different ranges; for example each color channel of an image may be in the range $[0, 1)$, while the position of each pixel may have a much wider range. Uneven scaling between different features in the feature vector may cause clustering algorithms to behave poorly.

One way to correct for uneven scaling between different features is to apply some sort of normalization to the feature vector. One of the simplest types of normalization is to force each feature to have zero mean and unit variance.

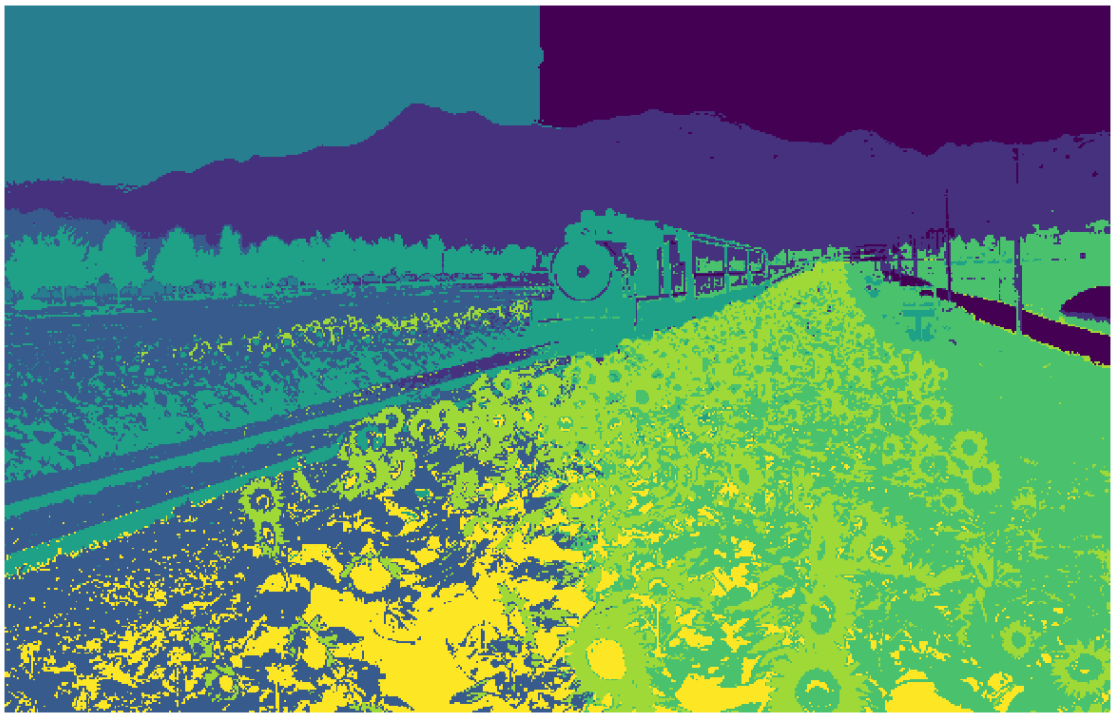
Implement **`color_position_features`** in `segmentation.py`.

Output segmentation should look like the following:

[83]:
```
from segmentation import color_position_features
np.random.seed(0)

features = color_position_features(img)

# Sanity checks
assert features.shape == (H * W, C + 2),\
    "Incorrect shape! Check your implementation."
```

```
assert features.dtype == np.float64,\
    "dtype of color_features should be float."

assignments = kmeans_fast(features, 8)
segments = assignments.reshape((H, W))

# Display segmentation
plt.imshow(segments, cmap='viridis')
plt.axis('off')
plt.show()
```

```
(399, 624, 5)
```



[84]:
```
visualize_mean_color_image(img, segments)
```

1.3.3 Extra Credit: Implement Your Own Feature

For this programming assignment we have asked you to implement a very simple feature transform for each pixel. While it is not required, you should feel free to experiment with other feature transforms. Could your final segmentations be improved by adding gradients, edges, SIFT descriptors, or other information to your feature vectors? Could a different type of normalization give better results?

Implement your feature extractor **`my_features`** in **`segmentation.py`**

Depending on the creativity of your approach and the quality of your writeup, implementing extra feature vectors can be worth extra credit (up to 1% of final grade).

Describe your approach: (YOUR APPROACH)

[ ]:
```
from segmentation import my_features

# Feel free to experiment with different images
# and varying number of segments
img = io.imread('train.jpg')
num_segments = 8

H, W, C = img.shape

# Extract pixel-level features
features = my_features(img)
```

```
# Run clustering algorithm
assignments = kmeans_fast(features, num_segments)

segments = assignments.reshape((H, W))

# Display segmentation
plt.imshow(segments, cmap='viridis')
plt.axis('off')
plt.show()
```

1.4 3 Quantitative Evaluation (30 points)

Looking at images is a good way to get an idea for how well an algorithm is working, but the best way to evaluate an algorithm is to have some quantitative measure of its performance.

For this project we have supplied a small dataset of cat images and ground truth segmentations of these images into foreground (cats) and background (everything else). We will quantitatively evaluate different segmentation methods (features and clustering methods) on this dataset.

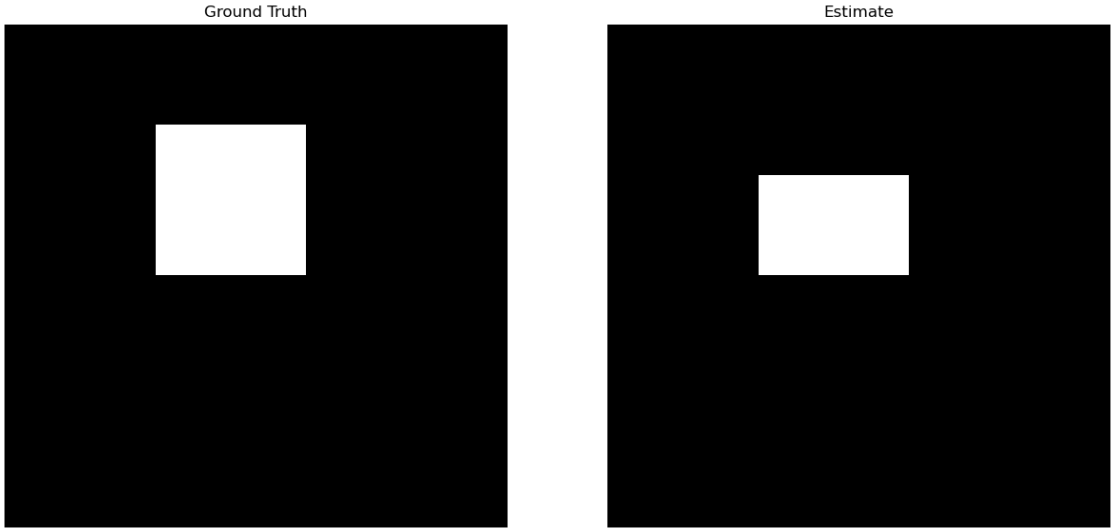
We can cast the segmentation task into a binary classification problem, where we need to classify each pixel in an image into either foreground (positive) or background (negative). Given the ground-truth labels, the accuracy of a segmentation is $(TP + TN)/(P + N)$.

Implement **compute_accuracy** in **segmentation.py**.

[85]:
```
from segmentation import compute_accuracy

mask_gt = np.zeros((100, 100))
mask = np.zeros((100, 100))

# Test compute_accracy function
mask_gt[20:50, 30:60] = 1
mask[30:50, 30:60] = 1

accuracy = compute_accuracy(mask_gt, mask)

print('Accuracy: %0.2f' % (accuracy))
if accuracy != 0.97:
    print('Check your implementation!')

plt.subplot(121)
plt.imshow(mask_gt)
plt.title('Ground Truth')
plt.axis('off')

plt.subplot(122)
plt.imshow(mask)
plt.title('Estimate')
```

```
plt.axis('off')

plt.show()
```

```
Accuracy: 0.97
```



You can use the script below to evaluate a segmentation method's ability to separate foreground from background on the entire provided dataset. Use this script as a starting point to evaluate a variety of segmentation parameters.

[110]:
```
from scipy.sparse.linalg import svds

def low_rank_approximation_to_channel(channel, k):
    U, S, Vt = svds(channel, k)
    return U @ np.diag(S) @ Vt

def low_rank_approximation(img, k):
    img = img.astype(np.float64) / 255.0
    approx = np.dstack([low_rank_approximation_to_channel(img[:, :, i], k) for␣
 ↪i in range(3)])
    approx = np.clip(approx, 0, 1) * 255.0
    return approx.round().astype(np.uint8)
```

[115]:
```
from utils import load_dataset, compute_segmentation
from segmentation import evaluate_segmentation

# Load a small segmentation dataset
imgs, gt_masks = load_dataset('./data')

# Set the parameters for segmentation.
```

```
# testing: 3, 5, 7
num_segments = 3
clustering_fn = kmeans_fast
# testing: color_features, color_position_features
feature_fn = color_position_features
# testing: 0.25, 0.5, 0.75
scale = 0.5

mean_accuracy = 0.0

segmentations = []

k = 5

for i, (img, gt_mask) in enumerate(zip(imgs, gt_masks)):
    # low-rank approx of the image
    img = low_rank_approximation(img, k)

    # Compute a segmentation for this image
    segments = compute_segmentation(img, num_segments,
                                    clustering_fn=clustering_fn,
                                    feature_fn=feature_fn,
                                    scale=scale)

    segmentations.append(segments)

    # Evaluate segmentation
    accuracy = evaluate_segmentation(gt_mask, segments)

    print('Accuracy for image %d: %0.4f' %(i, accuracy))
    mean_accuracy += accuracy

mean_accuracy = mean_accuracy / len(imgs)
print('Mean accuracy: %0.4f' % mean_accuracy)
```

```
Accuracy for image 0: 0.8056
Accuracy for image 1: 0.9216
Accuracy for image 2: 0.9734
Accuracy for image 3: 0.6997
Accuracy for image 4: 0.6890
Accuracy for image 5: 0.6847
Accuracy for image 6: 0.5919
Accuracy for image 7: 0.5818
Accuracy for image 8: 0.9129
Accuracy for image 9: 0.9759
Accuracy for image 10: 0.8920
Accuracy for image 11: 0.8438
Accuracy for image 12: 0.7345
```

```
Accuracy for image 13: 0.6638
Accuracy for image 14: 0.7478
Accuracy for image 15: 0.6000
Mean accuracy: 0.7699
```

[116]:
```python
# Visualize segmentation results

N = len(imgs)
plt.figure(figsize=(15,60))
for i in range(N):

    plt.subplot(N, 3, (i * 3) + 1)
    plt.imshow(low_rank_approximation(imgs[i], k))
    plt.axis('off')

    plt.subplot(N, 3, (i * 3) + 2)
    plt.imshow(gt_masks[i])
    plt.axis('off')

    plt.subplot(N, 3, (i * 3) + 3)
    plt.imshow(segmentations[i], cmap='viridis')
    plt.axis('off')

plt.show()
```

Include a detailed evaluation of the effect of varying segmentation parameters (feature transform, clustering method, number of clusters, resize) on the mean accuracy of foreground-background segmentations on the provided dataset. You should test a minimum of 6 combinations of parameters. To present your results, add rows to the table below (you may delete the first row).

One tip from us is that it's okay to avoid using hierarchical clustering altogether. The HAC algorithm is quite slow for larger scales. It is totally fine to just K-Means and modulate the other parameters of the clustering function!

Feature Transform

Clustering Method

Number of segments

Scale

Mean Accuracy

Color

K-Means

3

0.5

0.79

Color

K-Means

3

0.25

0.77

Color

K-Means

3

0.75

0.79

Color

K-Means

7

0.25

0.74

Color

K-Means

7

0.50

0.73

Color and Pos

K-Means

3

0.50

0.76

Color and Pos

K-Means

3

0.75

0.77

Color and Pos

K-Means

3

0.25

0.74

Observe your results carefully and try to answer the following question: 1. Based on your quantitative experiments, how do each of the segmentation parameters affect the quality of the final foreground-background segmentation? 2. Are some images simply more difficult to segment correctly than others? If so, what are the qualities of these images that cause the segmentation algorithms to perform poorly? 3. Also feel free to point out or discuss any other interesting observations that you made.

Write your analysis in the cell below.

Your answer here:

1. It seems like accuracy increases with increasing scale, but also takes significantly longer. It also decreases pretty dramatically with a higher number of segments. It also seems like just color is better than color and pos.
2. It seems like the algorithm really struggles when the foreground cat has a similar color to the background. This is logical. There is one gray cat standing in grass, which you would think would be really easy. However, I think because the cat has some dark lines, which is more similar to dark lines in the grass, it completely messes up the segmentation. Color and pos specifically: It seems to really struggle when there is more than one thing in the foreground.

For the first image, with two different color cats, it assigns each cat & their corresponding background to one segment. It seems like, for both methods, it really struggles when there is a lot of fine details in the image, such as the one cat jumping over some well-defined grass. This is really highlighted when you choose the scale to be large.

3. When the cat is really well-defined in the image, such as the cat on the white background, it does quite well. I think the color & pos method, when there is only one thing to focus on, is more robust to a similar color background, since all the background would seemingly be in one cluster since they are close, and then it needs to find another cluster in the cat. On the last part from Q2, maybe a more robust approach would be a low-rank approximation of the image (such as SVD), and then running the algorithm. Update: After trying this it does not seem to work very good. I didn't search over many values of k, but it didn't work the way I was hoping. It just ends up blending more colors with the background.

Cameras

February 21, 2025

[1]:
```
# ## This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment3/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Cameras

So far, we've looked at a ton of neat things that we can do with images: filtering, edge detection, stitching, segmentation, and resizing. Look at how far we've come!

In this part of the assignment, we'll explore some of the geometry that underlies how these images are formed.

[2]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt


def hash_numpy(x):
    import hashlib

    return hashlib.sha1(x.view(np.uint8)).hexdigest()


%load_ext autoreload
%autoreload 2
```

1.1 1. Transformations in 3D

In order to make sense of how objects in our world are rendered in a camera, we typically need to understand how they are located relative to the camera. In this question, we'll examine some properties of the transformations that formalize this process by expressing coordinates with respect to multiple frames.

We'll be considering a scene with two frames: a world frame (W) and a camera frame (C).

Notice that: - We have 3D points p, q, r, and s that define a square, which is parallel to the world zy plane - C_z and C_x belong to the plane defined by W_z and W_x - C_y is parallel to W_y

1.1.1 1.1 Reference Frame Definitions

First, we'll take a moment to validate your understanding of 3D reference frames.

Consider creating: - A point w at the origin of the world frame (O_w) - A point c at the origin of the camera frame (O_c)

Examine the x, y, and z axes of each frame, then express these points with respect to the world and camera frames. Fill in **`w_wrt_camera`**, **`w_wrt_world`**, and **`c_wrt_camera`**.

You can consider the length $d = 1$.

[3]:
```
d = 1.0

# Abbreviation note:
# - "wrt" stands for "with respect to", which is ~synonymous with "relative to"

w_wrt_world = np.array([0.0, 0.0, 0.0])  # Done for you
w_wrt_camera = np.array([0, 0, 1])  # Assign me!

c_wrt_world = np.array([1/np.sqrt(2), 0, 1/np.sqrt(2)])  # Assign me!
c_wrt_camera = np.array([0., 0., 0.])  # Assign me!

### YOUR CODE HERE
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
### END YOUR CODE
```

[4]:
```
# Run this cell to check your answers!
# assert (
#     (3,)
#     == w_wrt_world.shape
#     == w_wrt_camera.shape
#     == c_wrt_world.shape
#     == c_wrt_camera.shape
# ), "Wrong shape!"
```

```
assert (
    hash_numpy(w_wrt_world) == "d3399b7262fb56cb9ed053d68db9291c410839c4"
), "Double check your w_wrt_world!"
assert (
    hash_numpy(c_wrt_world) == "a4c525cd853a072d96cade8b989a9eaf1e13ed3d"
), "Double check your c_wrt_world!"
assert (
    hash_numpy(c_wrt_camera) == "d3399b7262fb56cb9ed053d68db9291c410839c4"
), "Double check your c_wrt_camera!"

print("Looks correct!")
```

```
Looks correct!
```

1.1.2 1.2 World Camera Transforms

Derive the homogeneous transformation matrix needed to convert a point expressed with respect to the world frame W in the camera frame C.

Discuss the rotation and translation terms in this matrix and how you determined them, then implement it in `camera_from_world_transform()`.

We've also supplied a set of `assert` statements below to help you check your work.

Hint #1: With rotation matrix $R \in \mathbb{R}^{3\times3}$ and translation vector $t \in \mathbb{R}^{3\times1}$, you can write transformations as 4×4 matrices:

$$\begin{bmatrix} x_C \\ y_C \\ z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ \vec{0}^\top & 1 \end{bmatrix} \begin{bmatrix} x_W \\ y_W \\ z_W \\ 1 \end{bmatrix}$$

Hint #2: Remember our 2D transformation matrix for rotations in the xy plane.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

To apply this to 3D rotations, you might think of this xy plane rotation as holding the z coordinate constant, since that's the axis you're rotating around, and transforming the x and y coordinates as described in the 2D formulation:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

(the Wikipedia page may also be helpful)

Hint #3: In a homogeneous transform, the translation is applied after the rotation.

As a result, you can visualize the translation as an offset in the output frame.

The order matters! You'll end up with a different transformation if you translate and then rotate versus if you rotate first and then translate with the same offsets. In lecture 2 we discussed a formulation for a combinated scaling, rotating, and translating matrix (in that order), which can be a useful starting point.

Your response here: Write your answer in this markdown cell. We are given that the $x-z$ axis of the camera coordinate system is contained within the $x-z$ coordinate system of the word, and a crucial picture, that shows that after spinning the $x-z$ camera coordinate system 135 degrees counterclockwise in the camera coordinate system, we would recover the orientation of the world coordinate system. Using the knowledge that translation is applied after rotations, as discussed above to get the right rotation to go from world coordinates to camera coordinates, we would need to rotate the world coordinate system by 135 degrees clockwise. After that rotation, we would have to translate based on where the world's center is relative to the camera's center, and in this case after that rotation the origin is d forward on the z-axis, so it should get mapped to $(0, 0, d)$. Notice that we have fixed the y coordinate for the rotations, so the rotation matrix R would be:

$$\begin{bmatrix} \cos(-3\pi/4) & 0 & -\sin(-3\pi/4) \\ 0 & 1 & 0 \\ \sin(-3\pi/4) & 0 & \cos(-3\pi/4) \end{bmatrix}.$$

After this rotation, from the cameras perspective, the world's origin is d forward in the z-axis, so it should get mapped to $(0, 0, d)$. This gives the full matrix:

$$\begin{bmatrix} \cos(-3\pi/4) & 0 & -\sin(-3\pi/4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-3\pi/4) & 0 & \cos(-3\pi/4) & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[5]:
```
# Check your answer against 1.1!
from cameras import camera_from_world_transform

T_camera_from_world = camera_from_world_transform()

# Check c_wrt_camera against T_camera_from_world @ w_wrt_world
w_wrt_camera_computed = (T_camera_from_world @ np.append(w_wrt_world, 1.0))[:3]
print(f"w_wrt camera: expected {w_wrt_camera}, computed␣
 ↪{w_wrt_camera_computed}")
assert np.allclose(
    w_wrt_camera, w_wrt_camera_computed
), "Error! (likely bad translation)"
print("Translation components look reasonable!")

# Check w_wrt_camera against T_camera_from_world @ c_wrt_world
c_wrt_camera_computed = (T_camera_from_world @ np.append(c_wrt_world, 1.0))[:3]
print(f"c_wrt camera: expected {c_wrt_camera}, computed␣
 ↪{c_wrt_camera_computed}")
assert np.allclose(
```

```
    c_wrt_camera, c_wrt_camera_computed
), "Error! (likely bad rotation)"
print("Rotation components looks reasonable!")
```

```
w_wrt camera: expected [0 0 1], computed [0. 0. 1.]
Translation components look reasonable!
c_wrt camera: expected [0. 0. 0.], computed [1.11022302e-16 0.00000000e+00
1.11022302e-16]
Rotation components looks reasonable!
```

1.1.3 1.3 Preserving Edge Orientations (Geometric Intuition)

Under the translation and rotation transformation from world coordinates to camera coordinates, which, if any, of the edges of the square retain their orientation and why?

For those that change orientation, how do they change? (e.g. translation x,y,z and rotation in one of our planes).

A sentence or two of geometric intuition is sufficient for each question, such as reasoning about the orientation of the edges and which axes we're rotating and translating about.

Your response here: Write your answer in this markdown cell. Since we rotate about the y-axis, I believe the p-q and s-r, since they are axis-aligned with the y-axis, would keep their left-right orientation, however the $s - r$ edge would definitely be moved. The y-coordinate would stay the same, since the y-axis is untouched, except the axis has been moved in the xz direction. However, the z-axis is pointing 45 degrees downwards, and the x-axis is pointing 45 degrees upwards, and the camera center is at a different place than the world center, so there would be movement in the xz direction. On the other hand, for an edge that is parallel to y, spinning the x-z axis doesn't actually do anything, so it would keep the same look (however it would also be translated since the vertices have a nonzero z coordinate).

1.1.4 1.4 Preserving Edge Orientations (Mathematical Proof)

We'll now connect this geometric intuition to your transformation matrix. Framing transformations as matrix multiplication is useful because it allows us to rewrite the difference between two transformed points as the transformation of the difference between the original points. For example, take points a and b and a transformation matrix T: $Ta - Tb = T(a - b)$.

All of the edges in the p, q, r, s square are axis-aligned, which means each edge has a nonzero component on only one axis. Assume that the square is 1 by 1, and apply your transformation to the edge vectors $bottom = q - p$ and $left = s - p$ to show which of these edges rotate and how.

Notation: You can apply the transformation to vectors representing the direction of each edge. If we transform all 4 corners, then the vector representing the direction of the transformed square's bottom is:

$$\begin{bmatrix} bottom_x{}' \\ bottom_y{}' \\ bottom_z{}' \\ 0 \end{bmatrix} = T \begin{bmatrix} q_x \\ q_y \\ q_z \\ 1 \end{bmatrix} - T \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Using matrix rules, we can rewrite this in terms of the edges of the original square

$$\begin{bmatrix} bottom_x{}' \\ bottom_y{}' \\ bottom_z{}' \\ 0 \end{bmatrix} = T \begin{bmatrix} q_x - p_x \\ q_y - p_y \\ q_z - p_z \\ 0 \end{bmatrix}$$

Eliminate the $q-p$ components that you know to be 0, and then apply your transformation to obtain the vector $bottom' = q' - p'$ defined above. Do the same for $left' = s' - p'$. Which edge rotated, and which one didn't?

Your response here: Write your answer in this markdown cell.

$q - p$ has 0 x coordinate, as do all the points, and the same z-coordinate, so it would look like $(0, 1, 0)$. The transformation matrix I used was:

$$\begin{pmatrix} \cos(-3\pi/4) & 0 & -\sin(-3\pi/4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-3\pi/4) & 0 & \cos(-3\pi/4) & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

As you can see, the second row is $(0, 1, 0, 0)$, which just keeps the same y-coordinate. In particular, $(0, 1, 0)$ gets mapped to the same vector (after adding/removing the extra coordiante of 1), so this edge would remain the same. However, $s - p$ edge would be $(0, 0, 1)$, which, using this same logic, would be rotated. In particular:

$$\begin{pmatrix} \cos(-3\pi/4) & 0 & -\sin(-3\pi/4) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-3\pi/4) & 0 & \cos(-3\pi/4) & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \cos(-3\pi/4) + 1 \\ 0 \end{pmatrix}$$

which is indeed different.

Interesting note: This may remind you of eigenvectors: one of these edges (the one that doesn't rotate) is an eigenvector of our transformation matrix!

1.1.5 1.5 Visualization

Implement **`apply_transform()`** to help us apply a homogeneous transformation to a batch of points.

Then, run the cell below to start visualizing our frames and the world square in PyPlot!

Using your code, we can animate a GIF that shows the transition of the square from its position in world coordinates to a new position in camera coordinates. We transform the perspective continuously from the world coordinate system to the camera coordinate system. Analogous to a homogeneous transform, you can see that we first rotate to match the orientation of the camera coordinate system, then translate to match the position of the camera origin.

If you want to see how the animation was computed or if you want to play around with its configuration, then check out **`animate_transformation`** in **`utils.py`**!

[6]:
```
from cameras import apply_transform
from utils import (
    animate_transformation,
    configure_ax,
    plot_frame,
    plot_square,
)

# Vertices per side of the square
N = 2

# Compute vertices corresponding to each side of the square
vertices_wrt_world = np.concatenate(
    [
        np.vstack([np.zeros(N), np.linspace(1, 2, N), np.ones(N)]),
        np.vstack([np.zeros(N), np.ones(N) + 1, np.linspace(1, 2, N)]),
        np.vstack([np.zeros(N), np.linspace(2, 1, N), np.ones(N) + 1]),
        np.vstack([np.zeros(N), np.ones(N), np.linspace(1, 2, N)]),
    ],
    axis=1,
)

# Visualize our rotation!
animate_transformation(
    "transformation.gif",
    vertices_wrt_world,
    camera_from_world_transform,
    apply_transform,
)

import IPython.display

with open("transformation.gif", "rb") as file:
    display(IPython.display.Image(file.read()))

# Uncomment to compare to staff animation
# with open("solution_transformation.gif", "rb") as file:
#     display(IPython.display.Image(file.read()))
```

…

…

1.2 2. Camera Intrinsics & Vanishing Points

In a pinhole camera, lines that are parallel in 3D rarely remain parallel when projected to the image plane. Instead, parallel lines will meet at a **vanishing point**:

1.2.1 2.1 Homogeneous coordinates (5 points)

Consider a line that is parallel to a world-space direction vector in the set $\{d \in \mathbb{R}^3 : d^\top d = 1\}$. Show that the image coordinates v of the vanishing point can be be written as $v = KRd$.

Hints: - As per the lecture slides, K is the camera calibration matrix and R is the camera extrinsic rotation. - As in the diagram above, the further a point on a 3D line is from the camera origin, the closer its projection will be to the line's 2D vanishing point. - Given a line with direction vector d, you can write a point that's infinitely far away from the camera via a limit: $\lim_{\alpha\to\infty} \alpha d$. - The 3D homogeneous coordinate definition is:

$$\begin{bmatrix} x & y & z & w \end{bmatrix}^\top \iff \begin{bmatrix} x/w & y/w & z/w & 1 \end{bmatrix}^\top$$

You answer here: Write your answer in this markdown cell.

I have interpreted the statement to mean the coordinate v of the vanishing point in 2d homogeneous coordinates is $v = KRd$. For any line $r(\alpha) = x_0 + \alpha d$, where x_0 is the starting point and α is the time, the output coordinate is just:

$$\lim_{\alpha\to\infty} \frac{KR\left(\begin{bmatrix} x_0 \\ 1 \end{bmatrix} + \begin{bmatrix} \alpha d \\ 1 \end{bmatrix}\right)}{KR\left(\begin{bmatrix} x_0 \\ 1 \end{bmatrix} + \begin{bmatrix} \alpha d \\ 1 \end{bmatrix}\right)_3}$$

This equals:

$$\lim_{\alpha \to \infty} \frac{KRx_0 + 2Kt + \alpha KRd}{(KRx_0 + 2Kt + \alpha KRd)_3} = \lim_{\alpha \to \infty} \frac{\alpha KRd}{\alpha (KRd)_3} = KRd/(KRd)_3.$$

The other terms drop out since there is no dependence on value $\alpha \to \infty$ which will remove them in the limit. In the 2d homogeneous coordinate system, this will just equal KRd (we would have to divide by the third coordinate to actually get the value we want in 2d).

1.2.2 2.2 Calibration from vanishing points (5 points)

Let $d_0, d_1, ...$ represent directional vectors for 3D lines in a scene, and $v_0, v_1, ...$ represent their corresponding vanishing points.

Consider the situtation when these lines are orthogonal:

$$d_i^\top d_j = 0, \text{for each } i \neq j$$

Show that:

$$(K^{-1}v_i)^\top (K^{-1}v_j) = 0, \text{for each } i \neq j$$

You answer here: Write your answer in this markdown cell. We just proved that $v_i = KRd_i$. In particular, $K^{-1}v_i = Rd_i$. As R is a rotation matrix, its rows are orthogonal, so $R^T R = RR^T = I$, so: $d_i^T d_j = d_i^T R^T R d_j = (Rd_i)^T (Rd_j) = (K^{-1}v_i)^T (K^{-1}v_j)$.

1.2.3 2.3 Short Response (5 points)

Respond to the following using bullet points:

- In the section above, we eliminated the extrinsic rotation matrix R. Why might this simplify camera calibration?
- Assuming square pixels and no skew, how many vanishing points with mutually orthogonal directions do we now need to solve for our camera's focal length and optical center?
- Assuming square pixels and no skew, how many vanishing points with mutually orthogonal directions do we now need to solve for our camera's focal length when the optical center is known?

You answer here: Write your answer in this markdown cell. - Camera calibration matrix K has 3 parameters, while rotation matrix R has 9. By getting rid of the rotation matrix R, we can vastly simplify our measurements, since we don't actually need to know the 9 parameters of the rotation matrix to solve for K. Equivalently, every rotation of the camera should have the same intrinsic matrix K, so we should be able to do our measurements with the camera in any way we want, and then find the vanishing points, and we are guaranteed to get the same K. - There are 3 parameters we need to find, the focal length f and the intrinsic parameters p_x, p_y. 3 equations in 3 variables yields a unique solution, so we need 3 vanishing points with mutually orthogonal directions to solve for K. - From above, once we know the optical center, (p_x, p_y), we only have to find the single parameter f, which we can do with one set of equations. So we only need 1 vanishing point in this case.

1.3 3. Intrinsic Calibration

Using the vanishing point math from above, we can solve for a camera matrix K!

First, let's load in an image. To make life easier for you, we've hand labeled a set of coordinates on it that we'll use to compute vanishing points.

[7]:
```
# Load image and annotated points; note that:
# > Our image is a PIL image type; you can convert this to NumPy with `np.
  ↪asarray(img)`
# > Points are in (x, y) format, which corresponds to (col, row)!
img = Image.open("images/pressure_cooker.jpg")
print(f"Image is {img.width} x {img.height}")
points = np.array(
    [
        [270.0, 327.0],  # [0]
        [356.0, 647.0],  # [1]
        [610.0, 76.0],  # [2]
        [706.0, 857.0],  # [3]
        [780.0, 585.0],  # [4]
        [1019.0, 226.0],  # [5]
    ]
)

# Visualize image & annotated points
fig, ax = plt.subplots(figsize=(8, 10))
ax.imshow(img)
ax.scatter(points[:, 0], points[:, 1], color="white", marker="x")
for i in range(len(points)):
    ax.annotate(
        f"points[{i}]",
        points[i] + np.array([15.0, 5.0]),
        color="white",
        backgroundcolor=(0, 0, 0, 0.15),
        zorder=0.1,
    )
```

```
Image is 1300 x 975
```

1.3.1 3.1 Finding Vanishing Points

In 2D, notice that a vanishing point can be computing by finding the intersection of two lines that we know are parallel in 3D.

To find the vanishing points in the image, implement **`intersection_from_lines()`**.

Then, run the cell below to check that it's working.

[8]:
```
from cameras import intersection_from_lines

# Python trivia: the following two assert statements are the same.
# > https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists
# > https://numpy.org/doc/stable/reference/arrays.indexing.
  ↪html#integer-array-indexing
assert np.allclose(
    intersection_from_lines(points[0], points[1], points[4], points[0],),
    points[0],
)
assert np.allclose(intersection_from_lines(*points[[0, 1, 4, 0]]), points[0])
print("Looks correct!")
```

```
Looks correct!
```

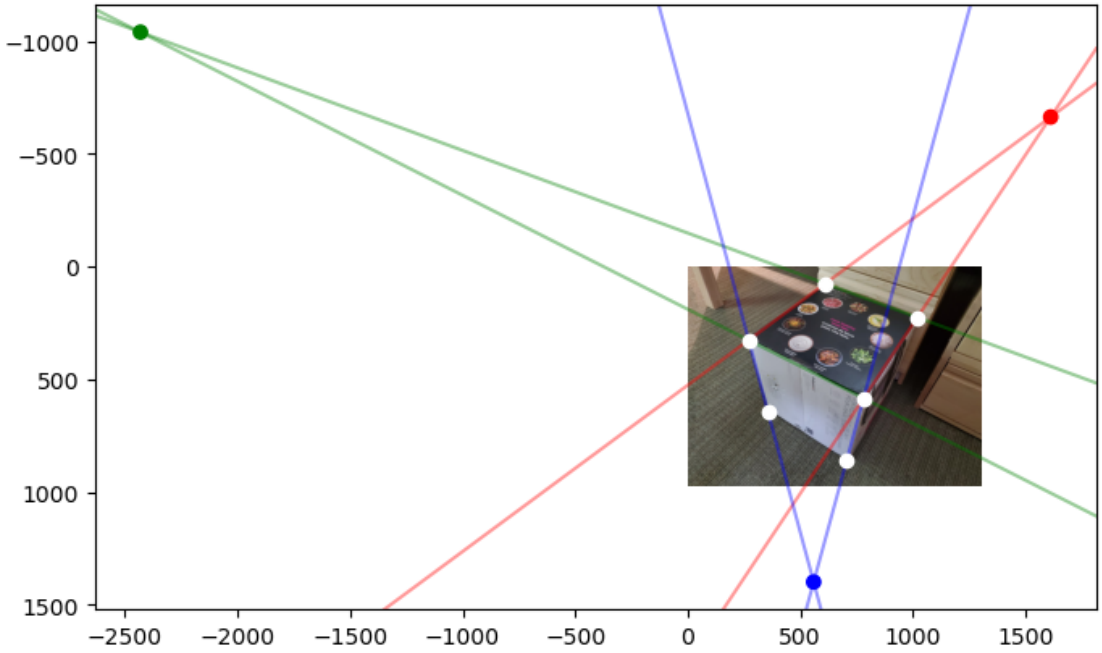
To use the constraint we derived above, we need to find vanishing points that correspond to three orthogonal direction vectors.

Populate `v0_indices`, `v1_indices`, and `v2_indices`, then run the cell below to compute `v`.

You should be able to get an output that looks like this (color ordering does not matter):

[9]:
```python
# Select points used to compute each vanishing point
#
# Each `v*_indices` list should contain four integers, corresponding to
# indices into the `points` array; the first two ints define one line and
# the second two define another line.

### YOUR CODE HERE
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

v0_indices = [0,2,4,5]
v1_indices = [4,0,5,2]
v2_indices = [1,0,3,4]

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
### END YOUR CODE

# Validate indices
assert (
    len(v0_indices) == len(v1_indices) == len(v2_indices) == 4
), "Invalid length!"
for i, j, k in zip(v0_indices, v1_indices, v2_indices):
    assert type(i) == type(j) == type(k) == int, "Invalid type!"

# Compute vanishing points
v = np.zeros((3, 2))
v[:, :2] = np.array(
    [
        intersection_from_lines(*points[v0_indices]),
        intersection_from_lines(*points[v1_indices]),
        intersection_from_lines(*points[v2_indices]),
    ]
)
assert v.shape == (3, 2)

# Display image
fig, ax = plt.subplots(figsize=(8, 10))
ax.imshow(img)

# Display annotated points
ax.scatter(points[:, 0], points[:, 1], color="white")
```

```
# Visualize vanishing points
colors = ["red", "green", "blue"]
for indices, color in zip((v0_indices, v1_indices, v2_indices), colors):
    ax.axline(*points[indices[:2]], zorder=0.1, c=color, alpha=0.4)
    ax.axline(*points[indices[2:]], zorder=0.1, c=color, alpha=0.4)
ax.scatter(v[:, 0], v[:, 1], c=colors)

pass
```



1.3.2 3.2 Computing Optical Centers

Next, implement **`optical_center_from_vanishing_points()`** to compute the 2D optical center from our vanishing points. Then, run the cell below to compute a set of optical center coordinates from our vanishing points.

[10]:
```
from cameras import optical_center_from_vanishing_points

optical_center = optical_center_from_vanishing_points(v[0], v[1], v[2],)

# TODO
assert np.allclose(np.mean(optical_center), 583.4127277436276)
assert np.allclose(np.mean(optical_center ** 2), 343524.39942528843)
print("Looks correct!")
```

```
# Display image
fig, ax = plt.subplots(figsize=(8, 10))
ax.imshow(img)

# Display optical center
ax.scatter(*optical_center, color="yellow")
ax.annotate(
    "Optical center",
    optical_center + np.array([20, 5]),
    color="white",
    backgroundcolor=(0, 0, 0, 0.5),
    zorder=0.1,
)

pass
```

```
Looks correct!
```

Above, as you can see, it shows a cyberpunk screenshot that I took in game instead of the cube with it's optical center placed. This is wrong, I am not fully sure why, and I literally couldn't get the pdf to compile without this hack. If you want to verify that I get the right image I believe you

could just download the Cameras.ipynb notebook and look there.

1.3.3 3.3 Computing Focal Lengths

Consider two vanishing points corresponding to orthogonal directions, and the constraint from above:

$$(K^{-1}v_0)^\top(K^{-1}v_1) = 0, \text{for each } i \neq j$$

Derive an expression for computing the focal length when the optical center is known, then implement **`focal_length_from_two_vanishing_points()`**.

When we assume square pixels and no skew, recall that the intrinsic matrix K is:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Hint: Optional, but this problem maybe be simpler if you factorize K as:

$$K = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When working with homogeneous coordinates, note that the lefthand matrix is a simple translation.

[11]:
```
from cameras import focal_length_from_two_vanishing_points

# If your implementation is correct, these should all be ~the same
f = focal_length_from_two_vanishing_points(v[0], v[1], optical_center)
print(f"Focal length from v0, v1: {f}")
f = focal_length_from_two_vanishing_points(v[1], v[2], optical_center)
print(f"Focal length from v1, v2: {f}")
f = focal_length_from_two_vanishing_points(v[0], v[2], optical_center)
print(f"Focal length from v0, v2: {f}")
```

```
Focal length from v0, v1: 1056.9925197084738
Focal length from v1, v2: 1056.9925197084735
Focal length from v0, v2: 1056.992519708474
```

1.3.4 3.4 Comparison to EXIF data

To validate our focal length computation, one smoke test we can run is compare it to parameters supplied by the camera manufacturer.

In JPEG images, these parameters and other metadata are sometimes stored using tags that are written when the photo is taken. Run the cell below to read & print some of this using the Python Imaging Library!

[12]:
```
from PIL.ExifTags import TAGS

# Grab EXIF data
exif = {TAGS[key]: value for key, value in img._getexif().items()}

# Print subset of keys
print(f"EXIF data for {img.filename}\n=====")
for key in (
    "DateTimeOriginal",
    "FocalLength",
    "GPSInfo",
    "Make",
    "Model",
):
    print(key.ljust(25), exif[key])
```

```
EXIF data for C:\Users\Rohan
Mukherjee\Documents\Homework\CSE455\assignment3_colab\images\pressure_cooker.jpg
=====
DateTimeOriginal          2020:11:06 01:02:20
FocalLength               4.3
GPSInfo                   {1: 'N', 2: (37.0, 25.0, 29.903), 3: 'W', 4: (122.0,
9.0, 34.294), 5: b'\x00', 6: 0.0}
Make                      samsung
Model                     SM-G970U
```

From above, we see that the focal length of our camera system is **4.3mm**.

Focal lengths are typically in millimeters, but all of the coordinates we've worked with thus far have been in pixel-space. Thus, we first need to convert our focal length from pixels to millimeters.

Try to visualize this conversion, then implement **physical_focal_length_from_calibration()**.

[13]:
```
from cameras import physical_focal_length_from_calibration

# Length across sensor diagonal for SM-G970U (Galaxy S10e)
# > https://en.wikipedia.org/wiki/Samsung_CMOS
sensor_diagonal_mm = 7.06

# Length across image diagonal
image_diagonal_pixels = np.sqrt(img.width ** 2 + img.height ** 2)

f_mm = physical_focal_length_from_calibration(
    f, sensor_diagonal_mm, image_diagonal_pixels,
)
print(f"Computed focal length:".ljust(30), f_mm)

error = np.abs(f_mm - 4.3) / 4.3
print("Calibration vs spec error:".ljust(30), f"{error * 100:.2f}%")
```

```
assert 0.06 < error < 0.07
```

```
Computed focal length:          4.592225962548816
Calibration vs spec error:      6.80%
```

1.3.5 3.5 Analysis (5 points)

If everything went smoothly, your computed focal length should only deviate from the manufacturer spec by ~6.8%.

Aside from manufacturing tolerances, name two or more other possible causes for this error, then discuss the limitations of this calibration method.

You answer here: Write your answer in this markdown cell. I have discovered that even using this method doesn't give consistent values. The cube has many faces, so when I tried 2 different orientations, they gave different values. Originally I used the top going out, front face going left, and front face going up, vs top going out, front face going up, and top face going left. So there is clearly some variance in the method. On top of this, there is also a problem with depth perception. Another thing is that we are completely assuming that the cube's lines are parallel in the real world, and that the faces are perpendicular. While true in theory, this could be ever so slightly wrong in practice, which would lead to a 7 percent error when everything compounds. Lastly, it says on there that the camera used was a samsung G970U. Presumably, this is not a pinhole camera, and might do more complicated things. So although we assume a pinhole model there could be some skew, which would also give some variance in the focal length.

1.4 4 Extra Credit

You can choose to attempt both, either, or neither! Each will count for up to 0.5% of your grade (1% total).

a) Projection

Generate a set of geometric shapes: cylinders, cubes, spheres, etc. Then, use your calibrated intrinsics to render them into the scene with correct perspective.

These can be simple wireframe representations (eg `plt.plot()`); no need for fancy graphics.

b) Extrinsinc Calibration (*Hard, possibly requires a lot of Google*)

Given that our box is 340mm (L) x 310mm (W) x 320mm (H), compute a 3D transformation (position, orientation) between the center of the box and the camera. In your submission, describe your approach and verify it by using both your calibrated extrinsics and intrinsics to overlay the image with a wireframe version of the box.