Homework 1

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

In [ ]:
```
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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

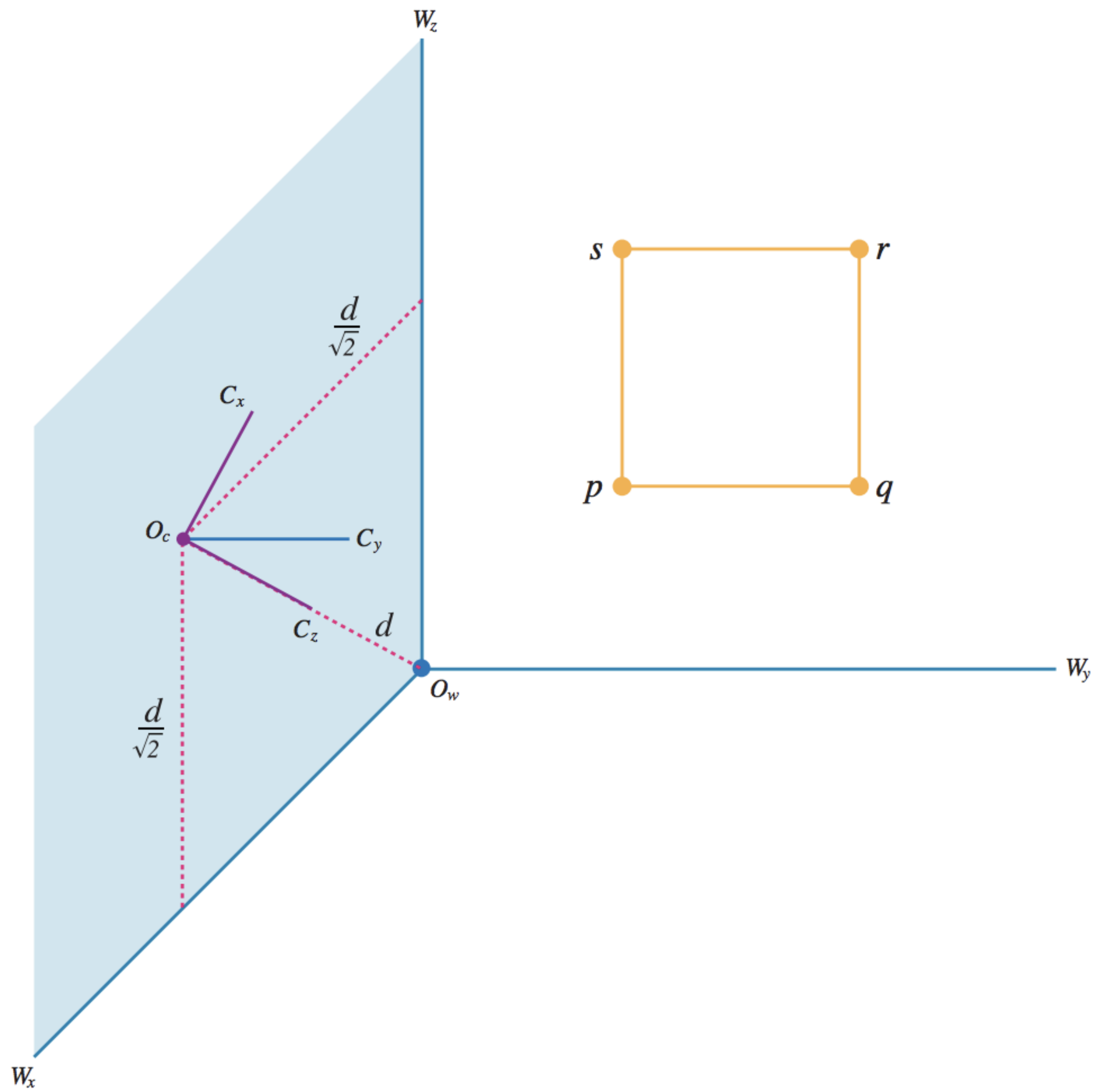
1. Transformations in 3D

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 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 $d = 1$.

In [ ]:
```python
from math import sqrt

d = 1.0

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

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

c_wrt_world = np.array([d/sqrt(2), 0.0, d/sqrt(2)])  # Assign me!
c_wrt_camera = np.array([0.0, 0.0, 0.0])  # Assign me!

### YOUR CODE HERE
pass
### END YOUR CODE
```

In [ ]:
```
# 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(w_wrt_camera) == "6248a1dcfe0c8822ba52527f68f7f98955584277"
), "Double check your w_wrt_camera!"
assert (
    hash_numpy(c_wrt_camera) == "d3399b7262fb56cb9ed053d68db9291c410839c4"
), "Double check your c_wrt_camera!"
assert (
    hash_numpy(c_wrt_world) == "a4c525cd853a072d96cade8b989a9eaf1e13ed3d"
), "Double check your c_wrt_world!"

print("Looks correct!")
```

```
Looks correct!
```

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:

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

(Alternatively you could simply take the rotation matrix from the [Wikipedia (https://en.wikipedia.org/wiki/Rotation_matrix)](https://en.wikipedia.org/wiki/Rotation_matrix) page)

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: The homogenous transformation matrix is constructed by applying a rotation and then a translation. First, the rotation matrix is $\begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \end{bmatrix}$ since $\theta = 45^\circ$ from the scene above. Second, the translation vector is simply $\begin{bmatrix} 0 \\ 0 \\ d \end{bmatrix}$ which we get from 1.1. Using *Hint #1*, we combine the two transformations as a 4×4 matrix:

$$\begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In [ ]:

```
# 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 [0. 0. 0.]
Rotation components looks reasonable!
```

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:

The top and bottom edges (sr and pq) retain their orientation, while the left and right edges (sp and rq) change orientation by rotating about the W_y axis. This is because the top and bottom edges are parallel to W_y so the orientation does not change.

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:

For the bottom edge, the nonzero component is on the W_y axis, so the edge vector is $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$. Applying the transformation from 1.2 returns the same vector $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, so the bottom edge does not rotate.

For the left edge, the nonzero component is on the W_z axis, so the edge vector is $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$. Applying the transformation from 1.2 returns the vector $\begin{bmatrix} \frac{\sqrt{2}}{2} \\ 0 \\ -\frac{\sqrt{2}}{2} \\ 0 \end{bmatrix}$, so the left edge rotated.

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!

1.5 Visualization

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

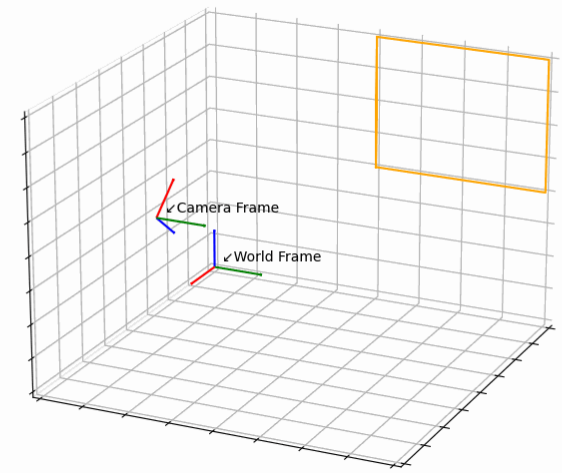
Then, run the cell below to start visualizing our frames and the world square in PyPlot!

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`** !

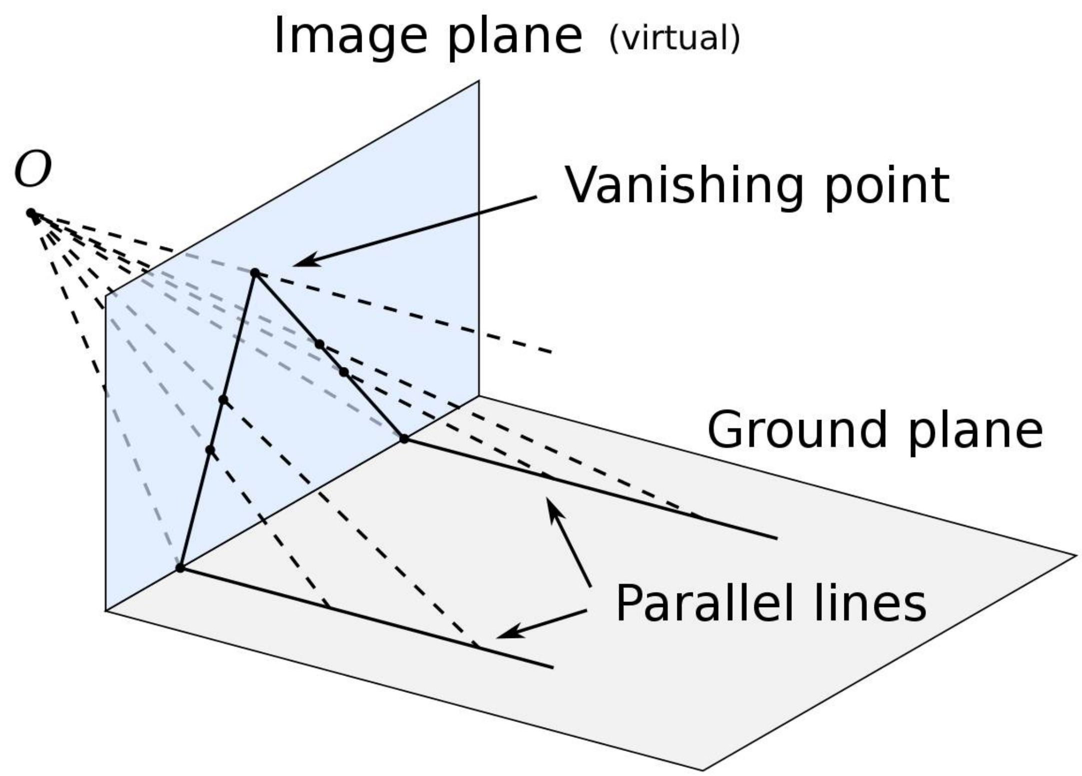
In [ ]:

```python
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()))
```


Camera Frame
World Frame

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**:



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:

$$[\,x \quad y \quad z \quad w\,]^\top \iff [\,x/w \quad y/w \quad z/w \quad 1\,]^\top$$

Your answer here:

We can use the transformation matrix to convert between world coordinates and image coordinates for a line l which is defined homogeneously as $[\,x/w \quad y/w \quad z/w \quad 1/w\,]^\top$ until infinity. We can write:

$$v = \lim_{w\to\infty} [\,K \quad 0\,] \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} [\,x/w \quad y/w \quad z/w \quad 1/w\,]^\top$$

$$v = [\,K \quad 0\,] \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} [\,x/w \quad y/w \quad z/w \quad 0\,]^\top$$

$$v = [\,K \quad 0\,] \begin{bmatrix} Rd \\ 0 \end{bmatrix}$$

$$v = KRd$$

2.2 Calibration from vanishing points (5 points)

Let $d_0, d_1, \ldots$ represent directional vectors for 3D lines in a scene, and $v_0, v_1, \ldots$ 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:

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

Your answer here:

We know that K and R are invertible, and can write $d = R^{-1}K^{-1}v$ and $Rd = K^{-1}v$. From this, we can see that:

$$
\begin{gathered}
d_i^\top d_j = 0 \\
d_i^\top (R^\top R) d_j = 0 \\
(Rd_i)^\top (Rd_j) = 0 \\
(K^{-1}v_i)^\top (K^{-1}v_j) = 0
\end{gathered}
$$

for each $i \neq j$

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?

Your answer here:

- This might simplify camera calibration because we don't need to deal with a rotation matrix.
- We need to solve for three vanishing points
- When the optical center is known, we need to solve for two vanishing points.

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.

In [ ]:

```python
# Load image and annotated points; note that:
# > Our image is a PIL image type; you can convert this to NumPy with `np.asar
ray(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
```

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.

Note that later parts of this homework will fail if you choose the side face instead of the front face for producing the leftmost vanishing point.

In [ ]:

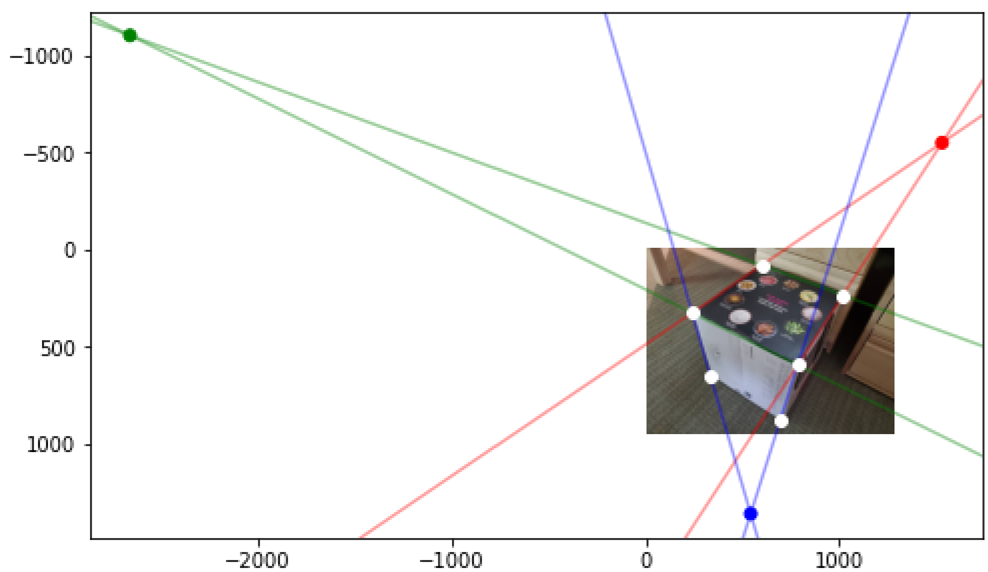
```
from cameras import intersection_from_lines

# Python trivia: the following two assert statements are the same.
# > https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lis
ts
# > 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!")
print(points)
```

```
Looks correct!
[[ 270.  327.]
 [ 356.  647.]
 [ 610.   76.]
 [ 706.  857.]
 [ 780.  585.]
 [1019.  226.]]
```

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

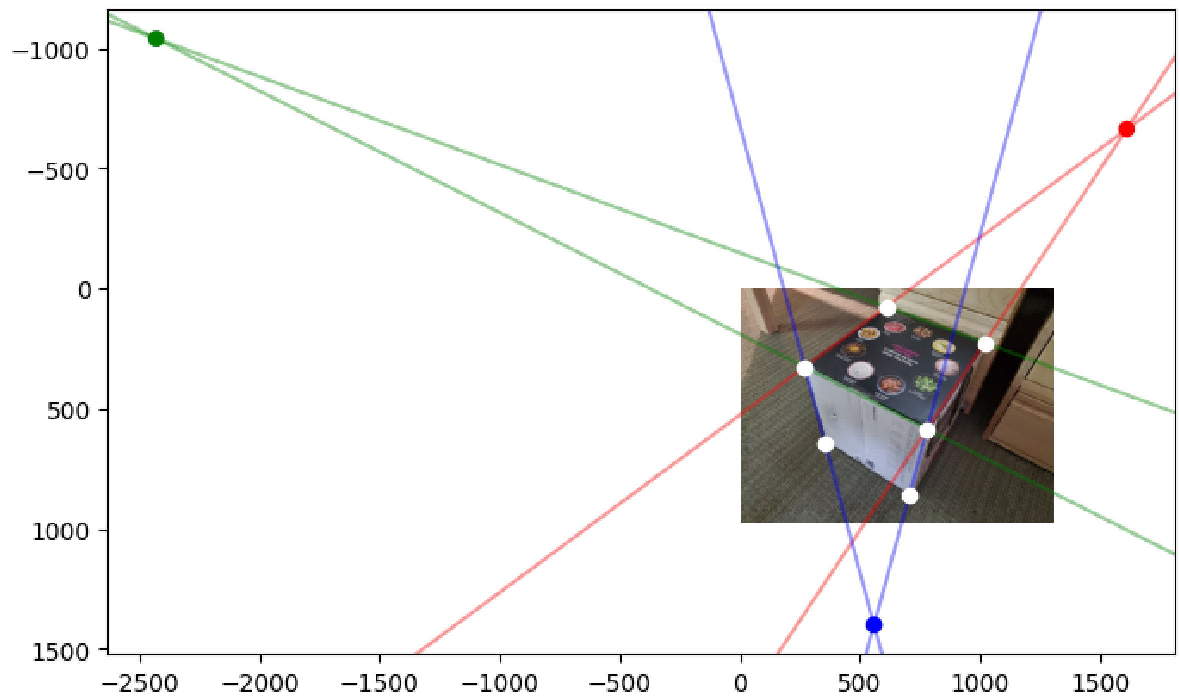
Populate `v0_indices` , `v1_indices` , and `v2_indices` , then run the cell below to compute `v` .

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



In [ ]:

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

# 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
```



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.

Hint: Property 3 from [1] may be useful. (Try connecting to Stanford campus network otherwise the paper link might not work for you.)

> [1] Caprile, B., Torre, V. **Using vanishing points for camera calibration.** *Int J Comput Vision 4, 127–139 (1990)*. [https://doi.org/10.1007/BF00127813 (https://doi.org/10.1007/BF00127813)](https://doi.org/10.1007/BF00127813)

In [ ]:

```python
from cameras import optical_center_from_vanishing_points

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

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!
```

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.

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

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 [EXIF (https://en.wikipedia.org/wiki/Exif)](https://en.wikipedia.org/wiki/Exif) tags that are written when the photo is taken. Run the cell below to read & print some of this using the Python Imaging Library!

In [ ]:

```python
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 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()`** .

In [ ]:

```python
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:        4.592225962548815
Calibration vs spec error:    6.80%
```

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.

Your answer here: One other possible cause for this error could be the image used for calibration. Another possible cause for this error could be distortion such as lens distortion. This calibration method has limitations such as accuracy and error in values such as focal lengths and vanishing points. In many applications, this error can be significant and a limitation this calibration method.

4 Extra Credit

You can choose to attempt both, either, or neither! This will be factored towards your final grade.

a) Projection

Generate a set of geometric shapes: cylinders, cubes, spheres, etc. Then, use your calibrated intrinsics to render them into the scene (i.e. overlaying them onto the image from Q3) 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.