hw4

February 11, 2025

1 Homework 4

In this homework, we will delve deeper into panorama stitching. We'll also explore epipolar geometry and stereo matching.

1.1 Submission Instructions

1. Make sure that the first line in any code cell with a function is the `def` line. The HW conversion script will fail to detect the function unless the function definition is on the first line of the code cell. Also double-check that each function is alone in its cell (there should be no function calls or imports anywhere in that cell).

2. Run the conversion script to get your .py file. As a sanity check, the autograder expects the following functions in your .py file:

   - **`harris_corners`**
   - **`simple_descriptor`**
   - **`match_descriptors`**
   - **`fit_affine_matrix`**
   - **`ransac`**
   - **`linear_blend`**
   - **`stitch_multiple_images`**

The code you write in Part 2 will not be autograded, we'll grade it as part of the PDF.

3. Make sure the Python file you upload to Gradescope is named **`hw4.py`**. This is required for the autograder to run correctly.

4. We added a section to the conversion script that generates an **HTML file that you can Ctrl+P and save as a PDF**. This will ensure images don't get cut off in your submission.

Notes on Running This Notebook:

Make sure to run each part from its begining to ensure that you compute all of the dependencies of your current question and don't crossover variables with the same name from other questions. So long as you run each part from its beginning, you can run the parts in any order. When assembling your PDF, we recommend running all cells in order from the top of the notebook to prevent any of these discontinuity errors.

1.2 Setup

[77]:
```
# import os

# if not os.path.exists("CS131_release"):
#     # Clone the repository if it doesn't already exist
#     !git clone https://github.com/StanfordVL/CS131_release.git

# %cd CS131_release/winter_2025/hw4_release/
```

[78]:
```
# Install the necessary dependencies
# (restart your runtime session if prompted to, and then re-run this cell)
!pip install -r requirements.txt
```

```
Requirement already satisfied: numpy in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from -r
requirements.txt (line 1)) (2.2.2)
Requirement already satisfied: scikit-image in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from -r
requirements.txt (line 2)) (0.25.1)
Requirement already satisfied: scipy in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from -r
requirements.txt (line 3)) (1.15.1)
Requirement already satisfied: matplotlib in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from -r
requirements.txt (line 4)) (3.10.0)
Requirement already satisfied: opencv-python in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from -r
requirements.txt (line 5)) (4.11.0.86)
Requirement already satisfied: networkx>=3.0 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (3.4.2)
Requirement already satisfied: pillow>=10.1 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (11.1.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (2025.1.10)
Requirement already satisfied: packaging>=21 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (24.2)
Requirement already satisfied: lazy-loader>=0.4 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from scikit-
image->-r requirements.txt (line 2)) (0.4)
Requirement already satisfied: contourpy>=1.0.1 in
```

```
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (4.55.8)
Requirement already satisfied: kiwisolver>=1.3.1 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (1.4.8)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from
matplotlib->-r requirements.txt (line 4)) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in
/Users/hazel/miniconda3/envs/cs131/lib/python3.10/site-packages (from python-
dateutil>=2.7->matplotlib->-r requirements.txt (line 4)) (1.17.0)
```

[79]:
```
from __future__ import print_function

import numpy as np
from skimage import filters
from skimage.feature import corner_peaks
from skimage.io import imread
from scipy.spatial.distance import cdist
from scipy.ndimage import convolve
import matplotlib.pyplot as plt
import cv2 as cv

from utils import pad, unpad, get_output_space, warp_image, plot_matches,␣
 ↪describe_keypoints

%matplotlib inline
plt.rcParams['figure.figsize'] = (12.0, 9.0)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

2 Part 1: Panorama Stitching

Panorama stitching is an early success of computer vision. Matthew Brown and David G. Lowe published a famous panoramic image stitching paper in 2007. Since then, automatic panorama stitching technology has been widely adopted in many applications such as Google Street View, panorama photos on smartphones, and stitching software such as Photosynth and AutoStitch.

We will detect and match keypoints from multiple images to build a single panoramic

image. This will involve several tasks: 1. Compare two sets of descriptors coming from two different images and find matching keypoints *(we did this in Homework 2)*. 2. Given matching keypoints, use least-squares to find the affine transformation matrix that maps points in one image to another. 3. Use RANSAC to give a more robust estimate of the affine transformation matrix. Given the transformation matrix, use it to transform the second image and overlay it on the first image, forming a panorama. 4. Blend panorama images together to remove blurry regions of overlapping images. 5. Stich multiple panorama images together.

2.1 1. Describing and Matching Keypoints (0 points)

In Homework 2, recall that we used Harris corner detection to detect keypoints in two images. Given these two sets of keypoints, we then determined which pairs of keypoints come from the same 3D points projected onto the two images. We did so by first converting the region around each keypoint into a descriptor. Then, we found good matches in the two sets of descriptors based on Euclidean distance.

Add your code from Homework 2 for the three functions below. You can also feel free to paste in the reference solutions posted on Ed!

[80]:
```
def harris_corners(img, window_size=3, k=0.04):
    """
    Compute Harris corner response map. Follow the math equation
    R=Det(M)-k(Trace(M)^2).

    Hint:
        You may use the function scipy.ndimage.filters.convolve,
        which is already imported above. If you use convolve(), remember to
        specify zero-padding to match our equations, for example:

            out_image = convolve(in_image, kernel, mode='constant', cval=0)

        You can also use for nested loops compute M and the subsequent Harris
        corner response for each output pixel, intead of using convolve().
        Your implementation of conv_fast or conv_nested in HW1 may be a
        useful reference!

    Args:
        img: Grayscale image of shape (H, W)
        window_size: size of the window function
        k: sensitivity parameter

    Returns:
        response: Harris response image of shape (H, W)
    """

    H, W = img.shape
    window = np.ones((window_size, window_size))
```

```
    response = np.zeros((H, W))

    # 1. Compute x and y derivatives (I_x, I_y) of an image
    dx = filters.sobel_v(img)
    dy = filters.sobel_h(img)

    ### YOUR CODE HERE
    # 2. Compute products of derivatives (I_x^2, I_y^2, I_xy) at each pixel
    Ix2 = dx ** 2
    Iy2 = dy ** 2
    Ixy = dy * dx

    # 3. Compute matrix M at each pixel
    Sx2 = convolve(Ix2, window, mode='constant', cval=0)
    Sy2 = convolve(Iy2, window, mode='constant', cval=0)
    Sxy = convolve(Ixy, window, mode='constant', cval=0)

    # 4. Compute corner response R=Det(M) - k*(Trace(M)^2) at each pixel
    det = (Sx2 * Sy2) - (Sxy ** 2)
    trace = Sx2 + Sy2
    response = det - k * (trace ** 2)
    ### END YOUR CODE

    return response

def simple_descriptor(patch):
    """
    Describe the patch by normalizing the image values into a standard
    normal distribution (having mean of 0 and standard deviation of 1)
    and then flattening into a 1D array.

    The normalization will make the descriptor more robust to change
    in lighting condition.

    Hint:
        In this case of normalization, if a denominator is zero, divide by 1␣
 ↪instead.

    Args:
        patch: grayscale image patch of shape (H, W)

    Returns:
        feature: 1D array of shape (H * W)
    """

    feature = []
```

```
    ### YOUR CODE HERE
    if patch.std() == 0:
        feature = (patch - patch.mean()).flatten()
    else:
        feature = ((patch-patch.mean())/patch.std()).flatten()
    ### END YOUR CODE

    return feature

def match_descriptors(desc1, desc2, threshold=0.5):
    """
    Match the feature descriptors by finding distances between them. A match is␣
 ↪formed
    when the distance to the closest vector is much smaller than the distance␣
 ↪to the
    second-closest, that is, the ratio of the distances should be STRICTLY␣
 ↪SMALLER
    than the threshold (NOT equal to). Return the matches as pairs of vector␣
 ↪indices.

    Hint:
        The Numpy functions np.sort, np.argmin, np.asarray might be useful

        The Scipy function cdist calculates Euclidean distance between all
        pairs of inputs
    Args:
        desc1: an array of shape (M, P) holding descriptors of size P about M␣
 ↪keypoints
        desc2: an array of shape (N, P) holding descriptors of size P about N␣
 ↪keypoints

    Returns:
        matches: an array of shape (Q, 2) where each row holds the indices of␣
 ↪one pair
        of matching descriptors
    """

    matches = []

    M = desc1.shape[0]
    dists = cdist(desc1, desc2)

    ### YOUR CODE HERE
    matches = []
    for i in range(M):
```

```
        sortedDI = np.sort(dists[i]) # Get the distances between desc1[i] and␣
↪descriptors in desc[2]
        if sortedDI[0] / sortedDI[1] < threshold:
            # append the index of the vector in desc1 and the index of the␣
↪vector in desc2
            matches.append((i, np.argmin(dists[i])))
    matches = np.asarray(matches)
    ### END YOUR CODE

    return matches
```

Run the following cell to detect and match the keypoints in two images.

[81]:
```
img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

np.random.seed(131)

# Detect keypoints in two images
keypoints1 = corner_peaks(harris_corners(img1, window_size=3),
                          threshold_rel=0.05,
                          exclude_border=8)
keypoints2 = corner_peaks(harris_corners(img2, window_size=3),
                          threshold_rel=0.05,
                          exclude_border=8)

# Extract features from the corners
desc1 = describe_keypoints(img1, keypoints1,
                           desc_func=simple_descriptor,
                           patch_size=5)
desc2 = describe_keypoints(img2, keypoints2,
                           desc_func=simple_descriptor,
                           patch_size=5)

# Match descriptors in image1 to those in image2
matches = match_descriptors(desc1, desc2, 0.7)
```

2.2 2. Transformation Estimation (20 points)

Now, we will use these matched keypoints to find a **transformation matrix** that maps points in the second image to the corresponding coordinates in the first image. In other words, if the point $p_1 = [y_1, x_1]$ in image 1 matches with $p_2 = [y_2, x_2]$ in image 2, we need to find an affine transformation matrix H such that

$$\tilde{p_2} H = \tilde{p_1},$$

where $\tilde{p_1}$ and $\tilde{p_2}$ are homogenous coordinates of p_1 and p_2.

Note that it may be impossible to find the transformation H that maps every point in image 2 exactly to the corresponding point in image 1. However, **we can estimate the transformation matrix with the least squares method.** Given N matched keypoint pairs, let X_1 and X_2 be $N \times 3$ matrices whose rows are homogenous coordinates of corresponding keypoints in image 1 and image 2 respectively. Then, we can estimate H by solving the least squares problem,

$$X_2 H = X_1$$

Implement **`fit_affine_matrix`** below.

Hint: read the documentation for ***`np.linalg.lstsq`***

[82]:
```
def fit_affine_matrix(p1, p2, to_pad=True):
    """
    Fit affine matrix such that p2 * H = p1. First, pad the descriptor vectors
    with a 1 using pad() to convert to homogeneous coordinates, then return
    the least squares fit affine matrix in homogeneous coordinates.

    Hint:
        You can use np.linalg.lstsq function to solve the problem.

        Explicitly specify np.linalg.lstsq's new default parameter rcond=None
        to suppress deprecation warnings, and match the autograder.

    Args:
        p1: an array of shape (M, P) holding descriptors of size P about M␣
↪keypoints
        p2: an array of shape (M, P) holding descriptors of size P about M␣
↪keypoints

    Return:
        H: a matrix of shape (P+1, P+1) that transforms p2 to p1 in homogeneous
        coordinates
    """

    assert (p1.shape[0] == p2.shape[0]),\
        'Different number of points in p1 and p2'

    if to_pad:
        p1 = pad(p1)
        p2 = pad(p2)

    ### YOUR CODE HERE
    H = np.linalg.lstsq(p2, p1, rcond=None)[0]
    ### END YOUR CODE

    # Sometimes numerical issues cause least-squares to produce the last
```

```
    # column which is not exactly [0, 0, 1]
    H[:,2] = np.array([0, 0, 1])
    return H
```

[83]:
```
# Sanity check for fit_affine_matrix

# Test inputs
a = np.array([[0.5, 0.1], [0.4, 0.2], [0.8, 0.2]])
b = np.array([[0.3, -0.2], [-0.4, -0.9], [0.1, 0.1]])

H = fit_affine_matrix(b, a)

# Target output
sol = np.array(
    [[1.25, 2.5, 0.0],
     [-5.75, -4.5, 0.0],
     [0.25, -1.0, 1.0]]
)

error = np.sum((H - sol) ** 2)

if error < 1e-20:
    print('Implementation correct!')
else:
    print('There is something wrong.')
```

```
Implementation correct!
```

After checking that your **fit_affine_matrix** function is running correctly, run the following code to apply it to images.

Images will be warped and image 2 will be mapped to image 1.

[84]:
```
p1 = keypoints1[matches[:,0]]
p2 = keypoints2[matches[:,1]]

# Find affine transformation matrix H that maps p2 to p1
H = fit_affine_matrix(p1, p2)

output_shape, offset = get_output_space(img1, [img2], [H])
print("Output shape:", output_shape)
print("Offset:", offset)

# Warp images into output space
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0
```

```
img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Plot warped images
plt.subplot(1,2,1)
plt.imshow(img1_warped)
plt.title('Image 1 Warped')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(img2_warped)
plt.title('Image 2 Warped')
plt.axis('off')

plt.show()
```

```
Output shape: [493 631]
Offset: [-37.17230306   0.        ]
```

Next, the two warped images are merged to get a panorama.

Your panorama may not look good at this point, but we will later use other techniques to get a better result!

[85]:
```
# Merge the two images
merged = img1_warped + img2_warped

# Track the overlap by adding the masks together
overlap = (img1_mask * 1.0 +  # Multiply by 1.0 for bool -> float conversion
           img2_mask)

# Normalize through division by `overlap` - but ensure the minimum is 1
```

```
normalized = merged / np.maximum(overlap, 1)

plt.imshow(normalized)
plt.axis('off')
plt.title('Fit-Affine Panorama')
plt.show()

plt.imshow(imread('solution_fit_affine_panorama.png'))
plt.axis('off')
plt.title('Fit-Affine Panorama Solution')
plt.show()
```

2.3 3. RANSAC (20 points)

Rather than directly feeding all our keypoint matches into **`fit_affine_matrix`**, we can use **RANSAC** ("RANdom SAmple Consensus") to select only "inliers" to use for computing the transformation matrix.

Use Euclidean distance as a measure of inliers vs. outliers.

The steps of RANSAC are: 1. Select random set of matches 2. Compute affine transformation matrix - You can call your **`fit_affine_matrix`** function for this. **Make sure to explicitly pass in `to_pad=False`.** 3. Find inliers using the given threshold 4. Repeat and keep the largest set of inliers (use >, i.e. break ties by whichever set is seen first) 5. Re-compute least-squares estimate on all of the inliers

Implement **`ransac`** below.

[86]:
```
def ransac(keypoints1, keypoints2, matches, n_iters=200, threshold=20):
    """
    Use RANSAC to find a robust affine transformation:
```

```
        1. Select random set of matches
        2. Compute affine transformation matrix
        3. Compute inliers via Euclidean distance
        4. Keep the largest set of inliers (use >, i.e. break ties by whichever␣
↪set is seen first)
        5. Re-compute least-squares estimate on all of the inliers

    Update max_inliers as a boolean array where True represents the keypoint
    at this index is an inlier, while False represents that it is not an inlier.

    Hint:
        You can use fit_affine_matrix to compute the affine transformation␣
↪matrix.
        Make sure to pass in to_pad=False, since we pad the matches for you␣
↪here.

        You can compute elementwise boolean operations between two numpy arrays,
        and use boolean arrays to select array elements by index:
        https://numpy.org/doc/stable/reference/arrays.indexing.
↪html#boolean-array-indexing

    Args:
        keypoints1: M1 x 2 matrix, each row is a point
        keypoints2: M2 x 2 matrix, each row is a point
        matches: N x 2 matrix, each row represents a match
            [index of keypoint1, index of keypoint 2]
        n_iters: the number of iterations RANSAC will run
        threshold: the threshold to find inliers

    Returns:
        H: a robust estimation of affine transformation from keypoints2 to
        keypoints 1
    """

    # Copy matches array, to avoid overwriting it
    orig_matches = matches.copy()
    matches = matches.copy()

    N = matches.shape[0]
    n_samples = int(N * 0.2)

    matched1 = pad(keypoints1[matches[:,0]])
    matched2 = pad(keypoints2[matches[:,1]])

    max_inliers = np.zeros(N, dtype=bool)
    n_inliers = 0
```

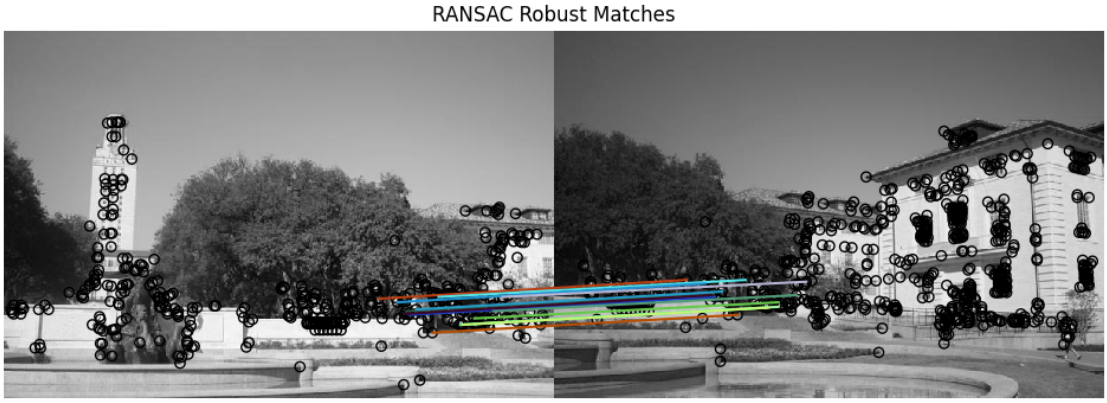
```
    # RANSAC iteration start

    # Note: while there're many ways to do random sampling, we use
    # `np.random.shuffle()` followed by slicing out the first `n_samples`
    # matches here in order to align with the autograder.
    # Sample with this code:
    for i in range(n_iters):
        # 1. Select random set of matches
        np.random.shuffle(matches)
        samples = matches[:n_samples]
        sample1 = pad(keypoints1[samples[:,0]])
        sample2 = pad(keypoints2[samples[:,1]])

    ### YOUR CODE HERE

        # 2. Compute affine transformation matrix, map sample2 to sample1
        H = fit_affine_matrix(sample1, sample2, to_pad=False)

        # 3. Compute inliers via Euclidean distance
        transformed_points = matched2 @ H
        distances = np.sqrt(np.sum((matched1 - transformed_points) ** 2,␣
↪axis=1))

        current_inliers = distances < threshold
        current_n_inliers = np.sum(current_inliers)

        # 4. Keep the largest set of inliers
        if current_n_inliers > n_inliers:
            n_inliers = current_n_inliers
            max_inliers = current_inliers


    # 5. Re-compute least-squares estimate on all of the inliers
    H = fit_affine_matrix(matched1[max_inliers], matched2[max_inliers],␣
↪to_pad=False)
    ### END YOUR CODE

    return H, orig_matches[max_inliers]
```

Now, run through the following cells to get a panorama. You'll be able to see the difference from the result we got before without RANSAC.

[87]:
```
# Set seed to compare output against solution image
np.random.seed(131)

H, robust_matches = ransac(keypoints1, keypoints2, matches, threshold=1)
print("Robust matches shape = ", robust_matches.shape)
```

```
print("H = \n", H)

# Visualize robust matches
fig, ax = plt.subplots(1, 1, figsize=(12, 9))
plot_matches(ax, img1, img2, keypoints1, keypoints2, robust_matches)
plt.axis('off')
plt.title('RANSAC Robust Matches')
plt.show()

plt.imshow(imread('solution_ransac.png'))
plt.axis('off')
plt.title('RANSAC Robust Matches Solution')
plt.show()
```

```
Robust matches shape =  (14, 2)
H =
 [[ 1.01796212e+00  2.42214470e-02  0.00000000e+00]
 [-2.92511861e-02  1.03559397e+00  0.00000000e+00]
 [ 2.03684578e+01  2.58675281e+02  1.00000000e+00]]
```



We can now use the tranformation matrix H computed using the robust matches to warp our images and create a better-looking panorama.

[88]:
```
output_shape, offset = get_output_space(img1, [img2], [H])

# Warp images into output space
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0

img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Plot warped images
plt.subplot(1,2,1)
plt.imshow(img1_warped)
plt.title('Image 1 warped')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(img2_warped)
plt.title('Image 2 warped')
plt.axis('off')

plt.show()
```

[89]:
```
# Merge the two images
merged = img1_warped + img2_warped

# Track the overlap by adding the masks together
overlap = (img1_mask * 1.0 +  # Multiply by 1.0 for bool -> float conversion
           img2_mask)
```

```python
# Normalize through division by `overlap` - but ensure the minimum is 1
normalized = merged / np.maximum(overlap, 1)
plt.imshow(normalized)
plt.axis('off')
plt.title('RANSAC Robust Panorama')
plt.show()

plt.imshow(imread('solution_ransac_panorama.png'))
plt.axis('off')
plt.title('RANSAC Robust Panorama Solution')
plt.show()
```

2.4 4. Better Image Merging (10 points)

You will notice the blurry region and unpleasant lines in the middle of the final panoramic image. Using a very simple technique called linear blending, we can smooth out a lot of these artifacts.

Currently, all the pixels in the overlapping region are weighted equally. However, since the pixels at the left and right ends of the overlap are very well complemented by the pixels in the other image, they can be made to contribute less to the final panorama.

Linear blending can be done with the following steps:

1. Define left and right margins for blending to occur between
2. Define a weight matrix for image 1 such that:
   - From the left of the output space to the left margin the weight is 1
   - From the left margin to the right margin, the weight linearly decrements from 1 to 0
3. Define a weight matrix for image 2 such that:
   - From the right of the output space to the right margin the weight is 1
   - From the left margin to the right margin, the weight linearly increments from 0 to 1
4. Apply the weight matrices to their corresponding images
5. Combine the images

In **linear_blend** below, implement the linear blending scheme to make the panorama look more natural.

[90]:
```
def linear_blend(img1_warped, img2_warped):
    """
    Linearly blend img1_warped and img2_warped by following the steps:

    1. Define left and right margins (already done for you)
    2. Define a weight matrices for img1_warped and img2_warped
        np.linspace and np.tile functions will be useful
    3. Apply the weight matrices to their corresponding images
    4. Combine the images

    Args:
        img1_warped: Refernce image warped into output space
        img2_warped: Transformed image warped into output space

    Returns:
        merged: Merged image in output space
    """

    out_H, out_W = img1_warped.shape # Height and width of output space
    img1_mask = (img1_warped != 0)  # Mask == 1 inside the image
    img2_mask = (img2_warped != 0)  # Mask == 1 inside the image

    # Find column of middle row where warped image 1 ends
```

```
    # This is where to end weight mask for warped image 1
    right_margin = out_W - np.argmax(np.fliplr(img1_mask)[out_H//2, :].
↪reshape(1, out_W), 1)[0]

    # Find column of middle row where warped image 2 starts
    # This is where to start weight mask for warped image 2
    left_margin = np.argmax(img2_mask[out_H//2, :].reshape(1, out_W), 1)[0]

    ### YOUR CODE HERE
    weight1 = np.zeros((out_H, out_W))
    weight2 = np.zeros((out_H, out_W))

    # For image 1:
    # Weight is 1 from left of output space to left margin
    weight1[:, :left_margin] = 1
    # Weight linearly decrements from 1 to 0 between left margin and right␣
↪margin
    ramp1 = np.linspace(1, 0, right_margin - left_margin)
    weight1[:, left_margin:right_margin] = ramp1

    # For image 2:
    # Weight linearly increments from 0 to 1 between left margin and right␣
↪margin
    ramp2 = np.linspace(0, 1, right_margin - left_margin)
    weight2[:, left_margin:right_margin] = ramp2
    # Weight is 1 from right margin to right of output space
    weight2[:, right_margin:] = 1

    # Apply masks
    weight1[~img1_mask] = 0
    weight2[~img2_mask] = 0

    # Blend images
    merged = img1_warped * weight1 + img2_warped * weight2
    ### END YOUR CODE

    return merged
```

Now let's see how linear blending improves our result.

[91]:
```
img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# Set seed to compare output against solution
np.random.seed(131)

# Detect keypoints in both images
```

```
ec1_keypoints1 = corner_peaks(harris_corners(img1, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)
ec1_keypoints2 = corner_peaks(harris_corners(img2, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)

print("EC1 keypoints1 shape = ", ec1_keypoints1.shape)
print("EC1 keypoints2 shape = ", ec1_keypoints2.shape)

# Extract features from the corners
ec1_desc1 = describe_keypoints(img1, ec1_keypoints1,
                           desc_func=simple_descriptor,
                           patch_size=16)
ec1_desc2 = describe_keypoints(img2, ec1_keypoints2,
                           desc_func=simple_descriptor,
                           patch_size=16)

print("EC1 desc1 shape = ", ec1_desc1.shape)
print("EC1 desc2 shape = ", ec1_desc2.shape)

# Match descriptors in image1 to those in image2
ec1_matches = match_descriptors(ec1_desc1, ec1_desc2, 0.7)

H, robust_matches = ransac(ec1_keypoints1, ec1_keypoints2, ec1_matches,␣
 ↪threshold=1)
print("Robust matches shape = ", robust_matches.shape)
print("H = \n", H)

output_shape, offset = get_output_space(img1, [img2], [H])
print("Output shape:", output_shape)
print("Offset:", offset)

# Warp images into output space
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0

img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Merge the warped images using linear blending scheme
merged = linear_blend(img1_warped, img2_warped)

plt.imshow(merged)
plt.axis('off')
```

```
plt.title('Linear Blend')
plt.show()

plt.imshow(imread('solution_linear_blend.png'))
plt.axis('off')
plt.title('Linear Blend Solution')
plt.show()
```

```
EC1 keypoints1 shape =  (396, 2)
EC1 keypoints2 shape =  (627, 2)
EC1 desc1 shape =  (396, 256)
EC1 desc2 shape =  (627, 256)
Robust matches shape =  (43, 2)
H =
 [[ 1.04208721e+00  6.75533965e-02  0.00000000e+00]
 [-2.84886715e-02  1.04672492e+00  0.00000000e+00]
 [ 1.30064490e+01  2.43335682e+02  1.00000000e+00]]
Output shape: [445 915]
Offset: [-4.51408398  0.        ]
```

Linear Blend Solution

2.5 5. Stitching Multiple Images (10 points)

Implement **`stitch_multiple_images`** below to stitch together an ordered chain of images.

Given a sequence of m images $(I_1, I_2, ..., I_m)$, take every neighboring pair of images and compute the transformation matrix which converts points from the coordinate frame of I_{i+1} to the frame of I_i.

Then, select a reference image I_{ref}, which is the first or left-most image in the chain. We want our final panorama image to be in the coordinate frame of I_{ref}.

You do **not** need to use linear blending for this problem: it's not included in the solution so the autograder does not expect it.

Hints: - You may want to review the Linear Algebra recitation slides on how to combine the effects of multiple transformation matrices. - The inverse of transformation matrix has the reverse effect. Use `numpy.linalg.inv` whenever you want to compute matrix inverse. - Take a look at the code cells in Part 3 (RANSAC) for an example of how to merge warped images and track the overlap using masks.

[92]:
```
def stitch_multiple_images(imgs, desc_func=simple_descriptor, patch_size=5):
    """
    Stitch an ordered chain of images together.

    Args:
        imgs: List of length m containing the ordered chain of m images
        desc_func: Function that takes in an image patch and outputs
            a 1D feature vector describing the patch
        patch_size: Size of square patch at each keypoint
```

```
    Returns:
        panorama: Final panorma image in coordinate frame of reference image
    """

    # Detect keypoints in each image
    keypoints = []  # keypoints[i] corresponds to imgs[i]
    for img in imgs:
        kypnts = corner_peaks(harris_corners(img, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)
        keypoints.append(kypnts)

    # Describe keypoints
    descriptors = []  # descriptors[i] corresponds to keypoints[i]
    for i, kypnts in enumerate(keypoints):
        desc = describe_keypoints(imgs[i], kypnts,
                                  desc_func=desc_func,
                                  patch_size=patch_size)
        descriptors.append(desc)

    # Match keypoints in neighboring images
    matches = []  # matches[i] corresponds to matches between
                  # descriptors[i] and descriptors[i+1]
    for i in range(len(imgs)-1):
        mtchs = match_descriptors(descriptors[i], descriptors[i+1], 0.7)
        matches.append(mtchs)

    ### YOUR CODE HERE
    pairwise_affines = []
    for i in range(len(imgs)-1):
        H, _ = ransac(keypoints[i], keypoints[i+1], matches[i], threshold=1)
        pairwise_affines.append(H)

    # Compute cumulative affine transformations to reference frame (first image)
    affines = [np.eye(3)]  # Identity transform for first image
    for i, H in enumerate(pairwise_affines):
        # Get previous transformation matrix
        prev_H = affines[i]
        # New transformation is current pairwise transform @ previous transform
        curr_H = H @ prev_H
        affines.append(curr_H)

    # Compute output space for panorama
    output_space = np.zeros((0, 0))
    offset = np.zeros(2)
    for i, img in enumerate(imgs):
        curr_space, curr_offset = get_output_space(img, imgs, affines)
```

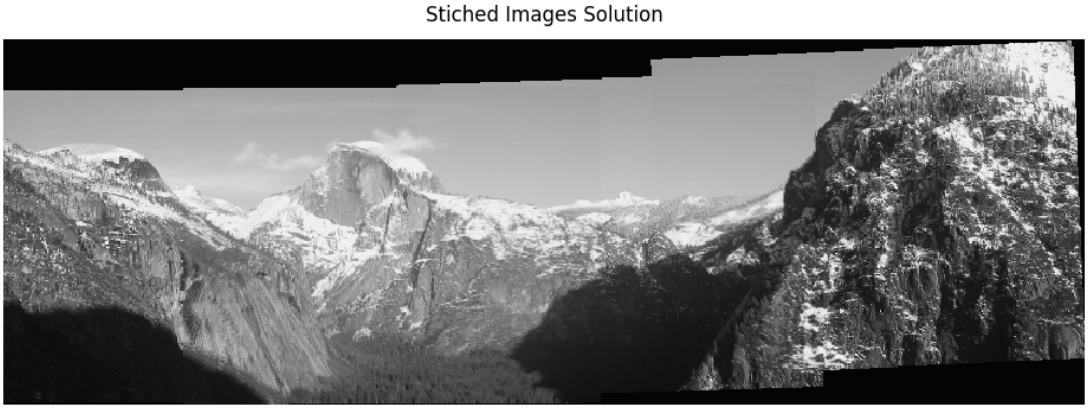
```
        if curr_space.shape[0] > output_space.shape[0]:
            output_space = curr_space
            offset = curr_offset

    # Warp images into output space
    warped_imgs = []
    masks = []
    for i, img in enumerate(imgs):
        img_warped = warp_image(img, affines[i], output_space, offset)
        img_mask = (img_warped != -1)
        img_warped[~img_mask] = 0
        warped_imgs.append(img_warped)
        masks.append(img_mask)

    # Blend warped images
    panorama = warped_imgs[0]
    for i in range(1, len(imgs)):
        panorama = linear_blend(panorama, warped_imgs[i])

    ### END YOUR CODE

    return panorama
```

We can now visualize the final panorama!

[93]:
```
# Set seed to compare output against solution
np.random.seed(131)

# Load images to be stitched
ec2_img1 = imread('yosemite1.jpg', as_gray=True)
ec2_img2 = imread('yosemite2.jpg', as_gray=True)
ec2_img3 = imread('yosemite3.jpg', as_gray=True)
ec2_img4 = imread('yosemite4.jpg', as_gray=True)

imgs = [ec2_img1, ec2_img2, ec2_img3, ec2_img4]

# Stitch images together
panorama = stitch_multiple_images(imgs, desc_func=simple_descriptor,␣
 ↪patch_size=5)
```

[94]:
```
# Visualize final panorama image
plt.imshow(panorama)
plt.axis('off')
plt.title('Stiched Images')
plt.show()
```

[95]:
```
plt.imshow(imread('solution_stitched_images.png'))
plt.axis('off')
plt.title('Stiched Images Solution')
plt.show()
```



3 Part 2: Epipolar Geometry and Stereo Matching

3.1 1. Drawing Epipolar Lines (10 points)

Recall our discussion of epipolar geometry from Lecture 9.

The **epipolar constraint** states that if we observe a single point x in one image, we must find the corresponding x' in the other image along the **epipolar line** corresponding to x. We represented the epipolar constraint as an equation

$$x'^T \mathbf{F} x = 0$$

where $\mathbf{F}$ is the **fundamental matrix**. We discussed estimating $\mathbf{F}$ using the normalized 8-point algorithm.

In this problem, we'll use the OpenCV library to estimate this fundamental matrix and draw epipolar lines on two images that capture the same scene from different cameras. First, run the following cell to load and display the two images we'll be working with.

[96]:
```
img1 = cv.imread('dvd_left.png', cv.IMREAD_GRAYSCALE)
img2 = cv.imread('dvd_right.png', cv.IMREAD_GRAYSCALE)

plt.subplot(121)
plt.imshow(img1)
plt.axis('off')
plt.subplot(122)
plt.imshow(img2)
plt.axis('off')
plt.show()
```

Similar to Homework 2, we'll use SIFT to extract keypoints and descriptors from the two images.

We'll then use a FLANN-based nearest neighbor search to filter good matches.

[97]:
```
sift = cv.SIFT_create()

# Find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)

pts1 = []
pts2 = []

for i,(m,n) in enumerate(matches):
```

```
    if m.distance < 0.8*n.distance:
        pts2.append(kp2[m.trainIdx].pt)
        pts1.append(kp1[m.queryIdx].pt)

pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
```

Now, fill in code in the cell below. 1. Estimate the fundamental matrix using a call to **cv.findFundamentalMat** with the relevant parameters. - For the `method` parameter, you can play around with different values and compare the results you get. - In our solution code, we used `cv.FM_LMEDS`, which applies the LMedS (Least Median of Squares) algorithm.

2. Using the `mask` output as a boolean mask, select only the inlier points (those where `mask.ravel()` is 1) in both images.
3. Compute the epipolar lines for both images using **cv.computeCorrespondEpilines**.

- You'll need to reshape the points array you pass in from to be shape `(N, 1, 2)`.

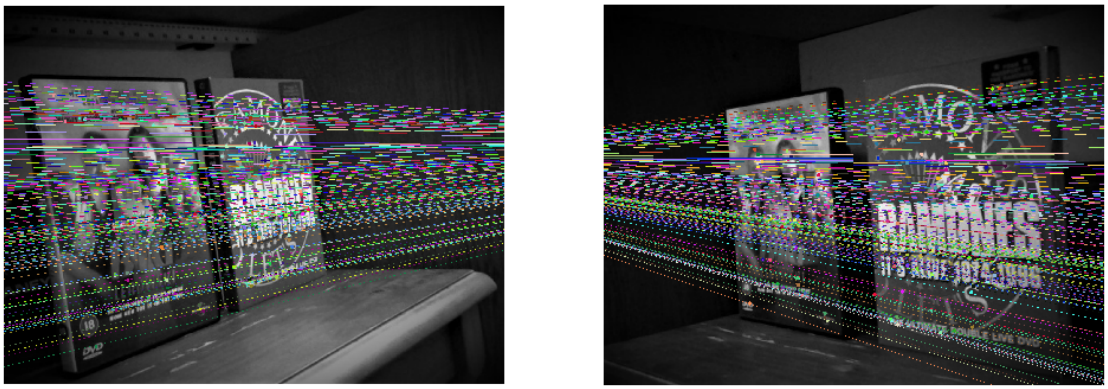
[98]:
```
# Estimate the fundamental matrix
F, mask = cv.findFundamentalMat(pts1, pts2, cv.FM_LMEDS) # YOUR CODE HERE

# Select only the inlier points
mask = mask.ravel().astype(bool)
pts1 = pts1[mask] # YOUR CODE HERE
pts2 = pts2[mask] # YOUR CODE HERE

# Compute the epipolar lines in image 1 corresponding to points in image 2
pts1_reshaped = pts1.reshape(-1, 1, 2)
pts2_reshaped = pts2.reshape(-1, 1, 2)
lines1 = cv.computeCorrespondEpilines(pts2_reshaped, 2, F) # YOUR CODE HERE

# Compute the epipolar lines in image 2 corresponding to points in image 1
lines2 = cv.computeCorrespondEpilines(pts1_reshaped, 1, F) # YOUR CODE HERE
```

Finally, we can visualize the epipolar lines on both the left and right images.

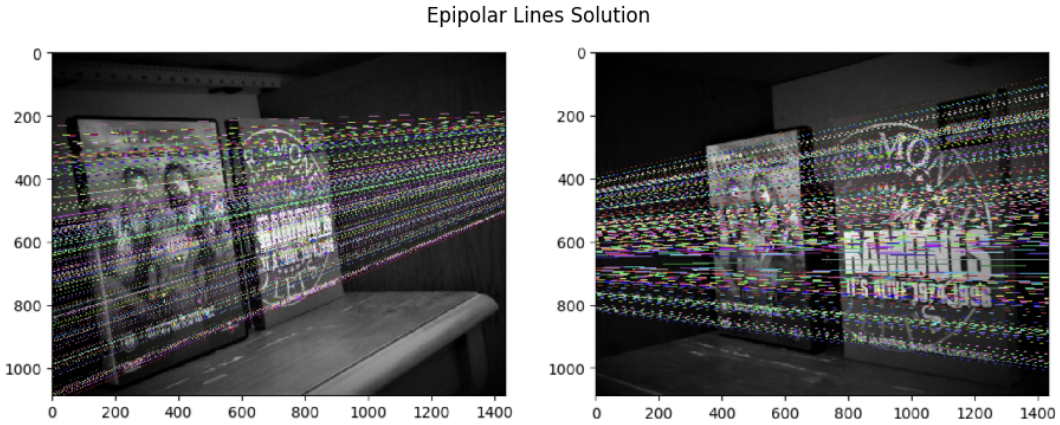
Note: It's normal to get different outputs here each time you re-run the above cells. If your output isn't similar to the solution, try re-running this part of the notebook.

[99]:
```
# Helper function to draw the epipolar lines
def drawlines(img1, img2, lines, pts1, pts2):
  r, c = img1.shape
  img1 = cv.cvtColor(img1, cv.COLOR_GRAY2BGR)
  img2 = cv.cvtColor(img2, cv.COLOR_GRAY2BGR)

  for r, pt1, pt2 in zip(lines, pts1, pts2):
      color = tuple(np.random.randint(0,255,3).tolist())
      x0, y0 = map(int, [0, -r[2]/r[1] ])
      x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
```

```
        img1 = cv.line(img1, (x0,y0), (x1,y1), color, 1)
        img1 = cv.circle(img1, tuple(pt1), 5, color, -1)
        img2 = cv.circle(img2, tuple(pt2), 5, color, -1)
    return img1, img2

# Draw epipolar lines corresponding to points in right image on left image
lines1 = lines1.reshape(-1,3)
img5, img6 = drawlines(img1, img2, lines1, pts1, pts2)

# Draw epipolar lines corresponding to points in left image on right image
lines2 = lines2.reshape(-1,3)
img3, img4 = drawlines(img2, img1, lines2, pts2, pts1)

plt.subplot(121)
plt.imshow(img5)
plt.axis('off')
plt.subplot(122)
plt.imshow(img3)
plt.axis('off')
plt.show()
```



[100]:
```
plt.imshow(imread('solution_epipolar_lines.png'))
plt.axis('off')
plt.title('Epipolar Lines Solution')
plt.show()
```



3.2 2. Computing Disparity Map (5 points)

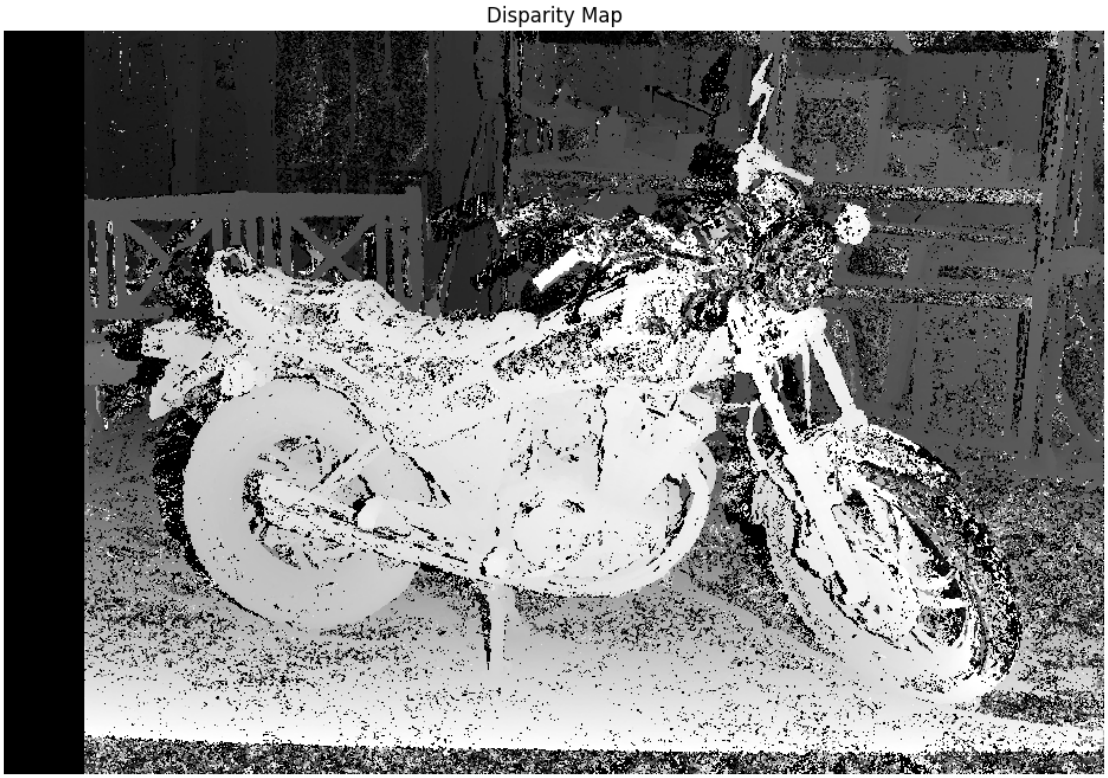
Recall the basic stereo matching algorithm we discussed in Lecture 9: for each pixel x in the first image, we can find the corresponding epipolar scanline in the second image, pick the best matching pixel x' on that scaline, and then compute the **disparity** $= x - x'$.

In this section, we'll work with these two images of a motorcycle.

[101]:
```
img1 = cv.imread('motorcycle_left.png', cv.IMREAD_GRAYSCALE)
img2 = cv.imread('motorcycle_right.png', cv.IMREAD_GRAYSCALE)

plt.subplot(121)
plt.imshow(img1)
plt.axis('off')
plt.subplot(122)
plt.imshow(img2)
plt.axis('off')
plt.show()
```

Given these two images, we can compute a disparity map using OpenCV's stereo semi-global block matching **StereoSGBM**.

[102]:
```
stereo = cv.StereoSGBM_create(minDisparity=16, numDisparities=200, blockSize=7)
disparity = stereo.compute(img1, img2).astype(np.float32) / 16.0

plt.imshow(disparity)
plt.axis('off')
plt.title('Disparity Map')
plt.show()
```



Question: Does the disparity map above look reasonable to you? Why do closer objects appear lighter and farther objects appear darker? What additional steps would we need to take to be able to convert this disparity map to a dense depth map?

Your Answer Here: The disparity map does look reasonable because the motorcycle in the foreground appear bright (high disparity) and the background appears darker (low disparity). Closer objects have a larger apparent shift between the left and right images (larger x - x'). This is because of the parallax effect - nearby objects appear to move more when viewed from different positions. So higher disparity (brighter pixels) correctly corresponds to closer objects. To convert this disparity map to actual depth values, we would need, camera calibration parameters (Focal length and Baseline). Then we can calculate Depth = (f * B) / disparity.

3.3 3. Rerendering from Different Camera Viewpoint (Extra Credit)

We can use the disparity map computed above to rerender the motorcycle image from a new camera viewpoint.

To do this, we have some basic function templates below.

1. In **`disparity_to_depth`**, convert from disparity to depth, using the relation

$$\text{depth} = \frac{B * f}{\text{disparity}}$$

   Estimates of B (the baseline) and f (the focal length) are given to you.

2. In **`backproject_to_3D`**, backproject the 2D image points to 3D using the full camera intrinsics K.

3. In **`transform_3D_points`**, apply the given rotation and translation matrices to the 3D points.

4. In **`project_to_new_image`**, reproject the translated 3D points onto our new image plane (our new viewpoint).

You can add any other functions that you want to improve the appearance of the rerendered image!

[ ]:
```python
### EXTRA CREDIT FUNCTIONS (feel free to add any functions to this)

def disparity_to_depth(disparity, B, f):
    """
    Convert disparity map to depth map.

    Args:
        disparity: disparity output from OpenCV's stereo matching algorithm
        B: baseline
        f: focal length

    Returns:
        depth: depth map

    Hint:
        When applying the formula (B * f) / disparity, add a small
        value (like 1e-6) to the denominator to avoid division by zero.
    """

    ### YOUR CODE HERE
    epsilon = 1e-6
    depth = (B * f) / (disparity + epsilon)

    # Set depth to a large value where disparity is very small/invalid
    depth[disparity < epsilon] = 1e9
    ### END YOUR CODE

    return depth
```

```python
def backproject_to_3D(disparity, K, B):
    """
    Convert disparity map to 3D points using camera intrinsics.

    Args:
        disparity: disparity output from OpenCV's stereo matching algorithm
        K: full camera intrinsics matrix
        B: baseline

    Returns:
        points_3D: 3D points in the original camera frame
    """

    ### YOUR CODE HERE
    height, width = disparity.shape

    # Create pixel coordinate grid
    x, y = np.meshgrid(np.arange(width), np.arange(height))

    # Compute depth from disparity
    f = K[0, 0]  # focal length from intrinsics
    depth = disparity_to_depth(disparity, B, f)

    # Get normalized image coordinates
    x_norm = (x - K[0, 2]) / K[0, 0]
    y_norm = (y - K[1, 2]) / K[1, 1]

    # Compute 3D coordinates
    X = x_norm * depth
    Y = y_norm * depth
    Z = depth

    # Stack coordinates into 3D points array
    points_3D = np.stack([X, Y, Z], axis=-1)
    ### END YOUR CODE

    return points_3D

def transform_3D_points(points_3D, R_new, T_new):
    """
    Apply rigid transformation to 3D points.

    Args:
        points_3D: 3D points in the original camera frame
        R_new: 3x3 rotation matrix
        T_new: 3x1 translation vector
```

```
    Returns:
        points_3D_transformed: 3D points in the new camera frame
    """

    ### YOUR CODE HERE
    # Reshape points for matrix multiplication
    points_shape = points_3D.shape
    points = points_3D.reshape(-1, 3)

    # Apply rotation and translation
    points_rotated = np.dot(points, R_new.T)
    points_transformed = points_rotated + T_new.reshape(1, 3)

    # Restore original shape
    points_3D_transformed = points_transformed.reshape(points_shape)
    ### END YOUR CODE

    return points_3D_transformed

def project_to_new_image(points_3D_transformed, K_new, image):
    """
    Project transformed 3D points back to the 2D image plane.

    Args:
        points_3D_transformed: 3D points in the new camera frame
        K_new: full camera intrinsics matrix for the new camera
        image: original image

    Returns:
        output_image: original image in the new camera frame
    """

    ### YOUR CODE HERE
    height, width = image.shape[:2]
    output_image = np.zeros_like(image)

    # Reshape points for projection
    points = points_3D_transformed.reshape(-1, 3)

    # Project 3D points to 2D
    points_2D = np.dot(points, K_new.T)
    points_2D = points_2D[:, :2] / points_2D[:, 2:3]
    points_2D = points_2D.reshape(height, width, 2)

    # Round to nearest pixel coordinates
    pixels_x = np.round(points_2D[:, :, 0]).astype(np.int32)
```

```
    pixels_y = np.round(points_2D[:, :, 1]).astype(np.int32)

    # Create valid mask for points that project within image bounds
    valid_mask = (pixels_x >= 0) & (pixels_x < width) & \
                 (pixels_y >= 0) & (pixels_y < height)

    # Map pixels from original image to new view
    y_coords, x_coords = np.meshgrid(np.arange(height), np.arange(width),␣
 ↪indexing='ij')
    output_image[pixels_y[valid_mask], pixels_x[valid_mask]] = \
        image[y_coords[valid_mask], x_coords[valid_mask]]

    # # Optional: Fill holes using morphological operations
    # kernel = np.ones((3,3), np.uint8)
    # output_image = cv.morphologyEx(output_image, cv.MORPH_CLOSE, kernel)
    ### END YOUR CODE

    return output_image


def post_process_image(image):
    """
    Apply post-processing to improve the quality of the rendered image
    """
    blurred = cv.medianBlur(image, 3)
    mask = np.all(blurred == 0, axis=2).astype(np.uint8) * 255
    processed = cv.inpaint(blurred, mask, 3, cv.INPAINT_TELEA)
    return processed
```

The following cell will test out the functions above (feel free to modify it depending on your specific implementation).

[104]:
```
image = cv.imread('motorcycle_left.png')
image_height, image_width, _ = image.shape

def estimate_intrinsics(image_width, image_height, fov=960):
    """Estimate camera intrinsics with an assumed field of view in degrees."""
    f = image_width / (2 * np.tan(np.radians(fov / 2)))
    K = np.array([[f, 0, image_width / 2],
                  [0, f, image_height / 2],
                  [0, 0, 1]])
    return f, K

f, K = estimate_intrinsics(image_width, image_height)
B = 0.1 # an estimate
K_new = K
```

```
# New camera pose (you can set these to whatever you like)
R_new = np.eye(3)  # No rotation
T_new = np.array([0.02, 0, 0])  # Move 2cm right

# Compute 3D points and transform to new view
points_3D = backproject_to_3D(disparity, K, B)
points_3D_transformed = transform_3D_points(points_3D, R_new, T_new)
output_image = project_to_new_image(points_3D_transformed, K_new, image)
output_image = post_process_image(output_image)

cv.imwrite('motorcycle_novel_view.png', output_image)
```

[104]: True

[105]:
```
output_image_rgb = cv.cvtColor(output_image, cv.COLOR_BGR2RGB)
plt.imshow(output_image_rgb)
plt.axis('off')
plt.title('Rerendered Image')
plt.show()
```