panorama

May 3, 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/assignment2/'
FOLDERNAME = 'cse455/assignments/assignment2/'
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
```

```
Mounted at /content/drive
/content/drive/My Drive/cse455/assignments/assignment2
```

1 Panorama (100 points + 15 points)

This assignment covers the Harris corner detector, RANSAC and the HOG descriptor for panorama stitching.

Grading: Each part will be graded based on the implementation (marked by autograder) and inline question answers. The total score shown at each part is in a format (implementation score + inline question score). For example, if you see (20 points + 5 points), it means 20 points will go to your implementation and 5 points will go to your inline question answers.

[2]:
```
from __future__ import print_function

# Setup
import numpy as np
from skimage import filters
from skimage.feature import corner_peaks
from skimage.io import imread
import matplotlib.pyplot as plt
```

```
from time import time

%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 extenrnal modules
%load_ext autoreload
%autoreload 2
```

1.1 Introduction: 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.

In this assignment, we will detect and match keypoints from multiple images to build a single panoramic image. This will involve several tasks: 1. Use Harris corner detector to find keypoints. 2. Build a descriptor to describe each point in an image. Compare two sets of descriptors coming from two different images and find matching keypoints. 3. Given a list of matching keypoints, use least-squares method to find the affine transformation matrix that maps points in one image to another. 4. Use RANSAC to give a more robust estimate of affine transformation matrix. Given the transformation matrix, use it to transform the second image and overlay it on the first image, forming a panorama. 5. Implement a different descriptor (HOG descriptor) and get another stitching result.

There's a lot of material to get hands-on with so we recommend starting early!

1.2 Notes on Running This Notebook

Make sure to run each Part from it's 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. For example, don't run parts 4 and 5 and then return to run only the last cell of part 3; your panorama won't be using the right transformed images! So long as you run each Part from it's 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.

1.3 Part 1 Harris Corner Detector (20 points + 5 points)

In this section, you are going to implement Harris corner detector for keypoint localization. Review the lecture slides on Harris corner detector to understand how it works. The Harris detection algorithm can be divide into the following steps: 1. Compute x and y derivatives (I_x, I_y) of an image 2. Compute products of derivatives (I_x^2, I_y^2, I_{xy}) at each pixel 3. Compute matrix M at each

pixel, where

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

4. Compute corner response $R = Det(M) - k(Trace(M)^2)$ at each pixel 5. Output corner response map $R(x, y)$

Step 1 is already done for you in the function **harris_corners** in **panorama.py**. We used the Sobel Operator, which computes smoothed gradients at each pixel in the x and y direction. See skimage documentation for sobel_v and sobel_h for more information on the sobel kernels and operators.

For step 3, we've created a uniform window function w for you in the starter code. You can assume that the window size will be odd.

Complete the function implementation of **harris_corners** and run the code below.

1.3.1 Hint: There are two ways to solve this problem

Vectorized: If you want to be really efficient, you can use the function `scipy.ndimage.filters.convolve`, which is already imported in `panorama.py`, and compute the response map R at every pixel all at once. If you're clever with your convolutions and determinant and trace calculations, you can compute the windowed gradients in M ($\sum_{x,y} w(x,y) \cdot I_x^2$, and $\sum_{x,y} w(x,y) \cdot I_y^2$, and $\sum_{x,y} w(x,y) \cdot I_{xy}$), and then compute the response map without any for loops!

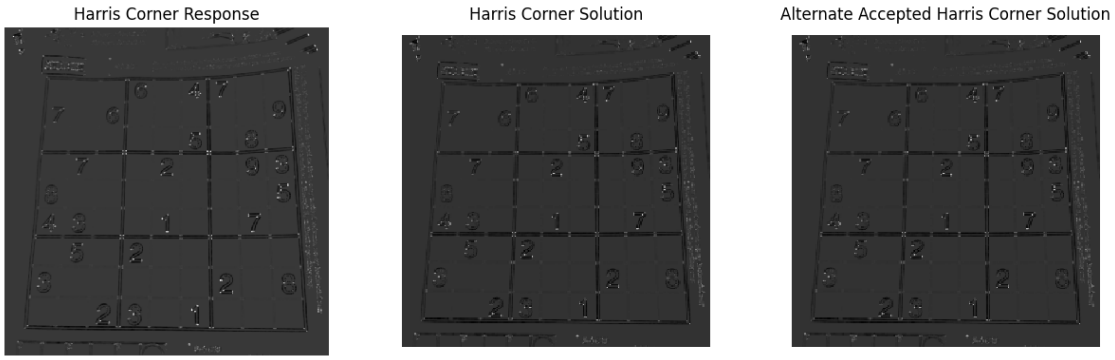
Iterative: The more intuitive solution is to iterate through each pixel of the image, compute M based on the surrounding neighborhood of pixel gradients in I_x, I_y, and I_{xy}, and then compute the response map pixel $R(x, y)$. You may find your implementations of conv_nested and conv_fast from HW 1 to be useful references!

Note that you'll want to explictly specify zero-padding to match the Harris response map definition, but we'll accept the default behavior of `scipy.ndimage.filters.convolve` as well. If you use zero-padding, both the vectorized and for-loop implementations will get you to the same answer!

The 'Alternate Accepted Harris Corner Solution' image presents the result of `scipy.ndimage.filters.convolve`'s default reflection padding, while the 'Harris Corner Solution' image presents the zero-padding solution. Similarly, 'Alternate Accepted Detected Corners Solution' image presents the result of `scipy.ndimage.filters.convolve`'s default reflection padding, while the 'Detected Corners Solution' image presents the zero-padding solution. **Both are accepted solutions!**

[3]:
```
from panorama import harris_corners

img = imread('sudoku.png', as_gray=True)

# Compute Harris corner response
response = harris_corners(img)

# Display corner response
plt.subplot(1,3,1)
plt.imshow(response)
```

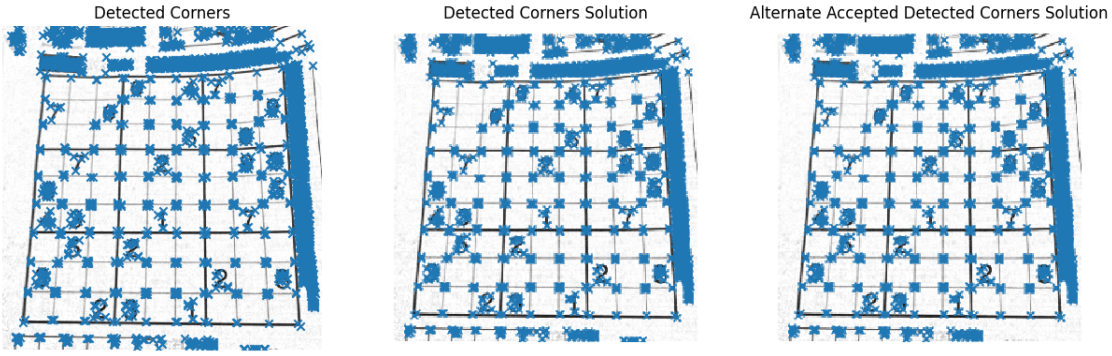
```
plt.axis('off')
plt.title('Harris Corner Response')

plt.subplot(1,3,2)
plt.imshow(imread('solution_harris.png', as_gray=True))
plt.axis('off')
plt.title('Harris Corner Solution')

plt.subplot(1,3,3)
plt.imshow(imread('solution_alternate_harris.png', as_gray=True))
plt.axis('off')
plt.title('Alternate Accepted Harris Corner Solution')

plt.show()
```



Once you implement the Harris detector correctly, you will be able to see small bright blobs around the corners of the sudoku grids and letters in the output corner response image. The function `corner_peaks` from `skimage.feature` performs non-maximum suppression to take local maxima of the response map and localize keypoints.

[4]:
```
# Perform non-maximum suppression in response map
# and output corner coordinates
corners = corner_peaks(response, threshold_rel=0.01)

# Display detected corners
plt.subplot(1,3,1)
plt.imshow(img)
plt.scatter(corners[:,1], corners[:,0], marker='x')
plt.axis('off')
plt.title('Detected Corners')

plt.subplot(1,3,2)
plt.imshow(imread('solution_detected_corners.png'))
plt.axis('off')
```

```
plt.title('Detected Corners Solution')

plt.subplot(1,3,3)
plt.imshow(imread('solution_alternate_detected_corners.png'))
plt.axis('off')
plt.title('Alternate Accepted Detected Corners Solution')

plt.show()
```



Question 1.1 (5 points) Explain what does the corner response R measure and what is the significance of it in the Harris Corner Detector algorithm.

Your Answer: Write your solution in this markdown cell.

Given that the Harris Corner Detector tries to find, well, corners, it should make sense the corner response R measures the probability that the pixel is in a corner. Notice that the expression for R takes in both eigenvalues in practically equal amounts, so if one is far larger than the other, this means that there is only a major change in intensity in one direction (and therefore a small posistive or negative R), so the detector found an edge, not a corner.

1.4 Part 2 Describing and Matching Keypoints (20 points + 5 points)

We are now able to localize keypoints in two images by running the Harris corner detector independently on them. Next question is, how do we determine which pair of keypoints come from corresponding locations in those two images? In order to *match* the detected keypoints, we must come up with a way to *describe* the keypoints based on their local appearance. Generally, each region around detected keypoint locations is converted into a fixed-size vectors called *descriptors*.

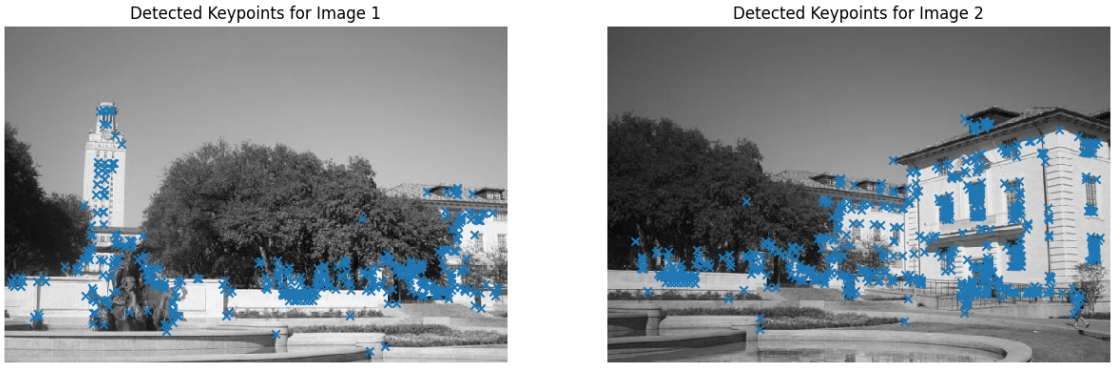
1.4.1 Part 2.1 Creating Descriptors (10 points)

In this section, you are going to implement the **`simple_descriptor`** function, where each keypoint is described by the normalized intensity of a small patch around it.

[5]:
```
from panorama import harris_corners
```

```
img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# 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)

print("Keypoints 1 shape = ", keypoints1.shape)
print("Keypoints 2 shape = ", keypoints2.shape)

# Display detected keypoints
plt.subplot(1,2,1)
plt.imshow(img1)
plt.scatter(keypoints1[:,1], keypoints1[:,0], marker='x')
plt.axis('off')
plt.title('Detected Keypoints for Image 1')

plt.subplot(1,2,2)
plt.imshow(img2)
plt.scatter(keypoints2[:,1], keypoints2[:,0], marker='x')
plt.axis('off')
plt.title('Detected Keypoints for Image 2')
plt.show()
```

```
Keypoints 1 shape =  (396, 2)
Keypoints 2 shape =  (627, 2)
```



1.4.2 Part 2.2 Matching Descriptors (10 points)

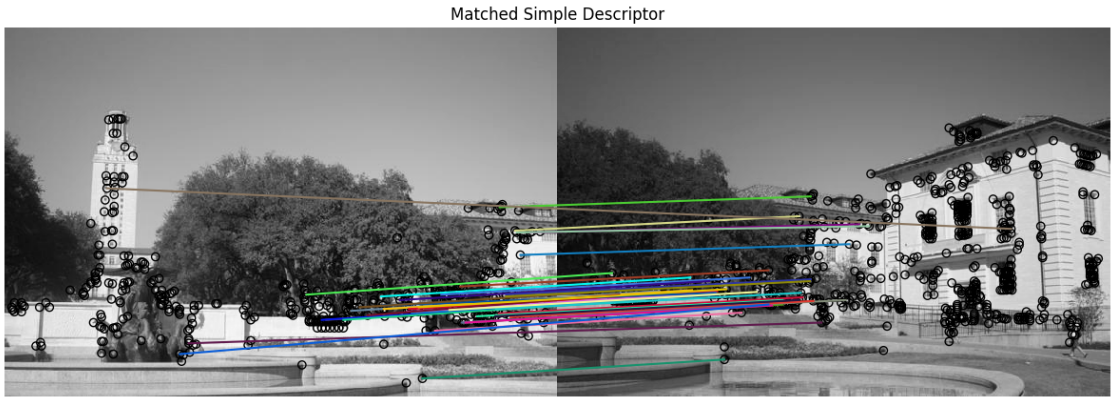
Next, implement the **match_descriptors** function to find good matches in two sets of descriptors. First, calculate Euclidean distance between all pairs of descriptors from image 1 and image 2. Then use this to determine if there is a good match: for each descriptor in image 1, if the distance to the closest descriptor in image 2 is significantly (by a given factor) smaller than the distance to the second-closest, we call it a match. The output of the function is an array where each row holds the indices of one pair of matching descriptors.

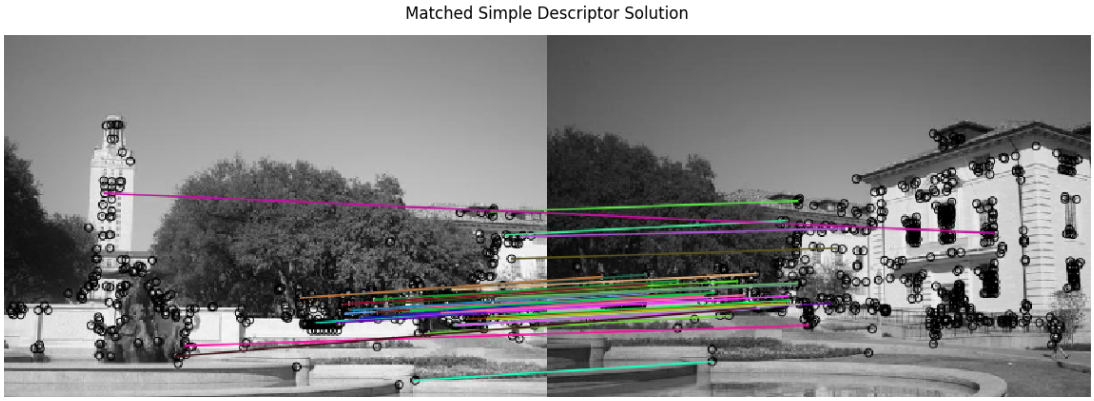
Checking your answer: you should see an identical matching of keypoints as the solution, but the precise colors of each line will change with every run of keypoint matching so colors do not need to match.

Optional ungraded food for thought: Think about why this method of keypoint matching is not commutative.

[6]:
```
from panorama import simple_descriptor, match_descriptors, describe_keypoints
from utils import plot_matches

# Set seed to compare output against solution
np.random.seed(455)

patch_size = 5

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

print("Desc1 shape = ", desc1.shape)
print("Desc2 shape = ", desc2.shape)

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

# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
ax.axis('off')
plt.title('Matched Simple Descriptor')
plot_matches(ax, img1, img2, keypoints1, keypoints2, matches)
plt.show()

plt.imshow(imread('solution_simple_descriptor.png'))
plt.axis('off')
plt.title('Matched Simple Descriptor Solution')
plt.show()
```

```
Desc1 shape =  (396, 25)
Desc2 shape =  (627, 25)
```





Question 2.1 (5 points) How does matching descriptors help in identifying corresponding keypoints in different images?

Your Answer: Write your solution in this markdown cell.

It should be quite obvious, considering that given images that either are transformed in some way, or capture slightly different pieces of the same general objects, having descriptors being matched allows a systematic process to occur when it comes to matching the most distinct and unique points of an image specfically because they are so distinct, which are clearly helpful to identify areas because of the aftermentioned reasons, along wih the fact that they usaually are maintained after transforming the image. How else could one mathmatically match important pieces in different pieces in a mathematical way without using math?

1.5 Part 3 Transformation Estimation (20 points + 0 points)

We now have a list of matched keypoints across the two images. We will use this 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 least squares. 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`** in `panorama.py`

-Hint: read the documentation about np.linalg.lstsq

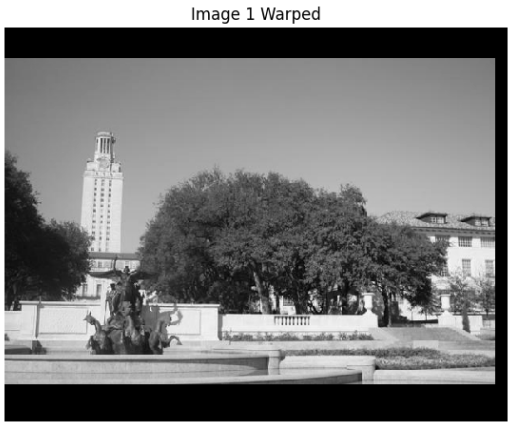
[7]:
```
from panorama import fit_affine_matrix

# 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.')
```

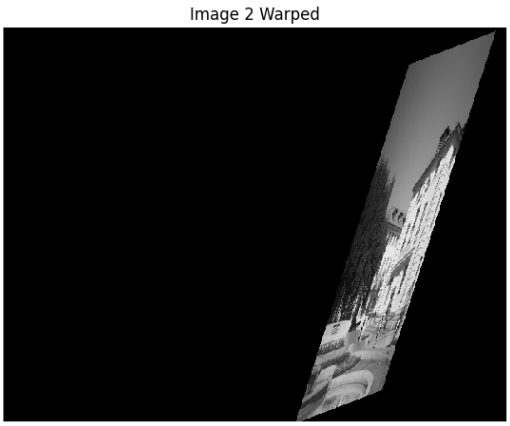
```
Implementation correct!
```

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.

[8]:
```
from utils import get_output_space, warp_image

# Extract matched keypoints
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 sapce
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.

[9]:
```
# 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()
```

Fit-Affine Panorama

1.6 Part 4 RANSAC (20 points + 5 points)

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

The steps of RANSAC are: 1. Select random set of matches 2. Compute affine transformation matrix 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

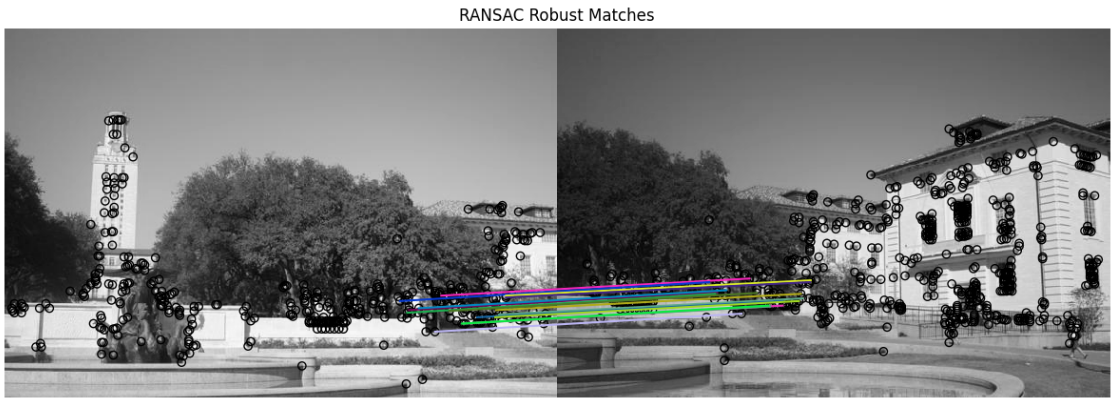
In this case, use Euclidean distance between matched points as a measure of inliers vs outliers.

Implement **`ransac`** in `panorama.py`, run through the following code to get a panorama. You can see the difference from the result we get without RANSAC.

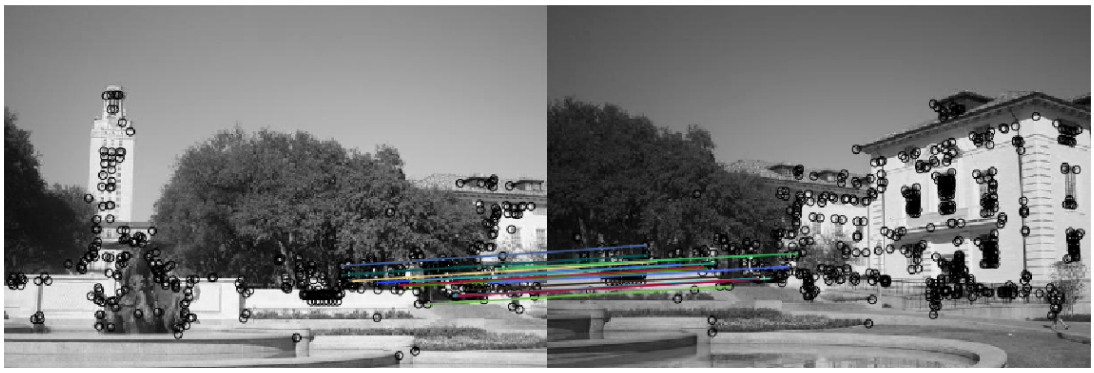
[10]:
```
from panorama import ransac

# Set seed to compare output against solution image
np.random.seed(455)
```

```
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=(15, 12))
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 =  (16, 2)
H =
 [[ 1.02466297e+00  2.32906256e-02  0.00000000e+00]
 [-3.06310846e-02  1.04662432e+00  0.00000000e+00]
 [ 1.88464439e+01  2.56565361e+02  1.00000000e+00]]
```

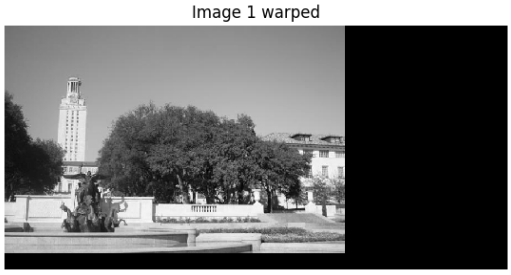


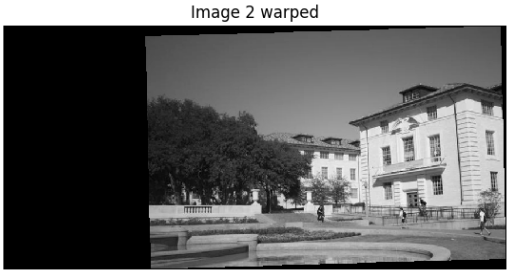
RANSAC Robust Matches Solution



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

[11]:
```
output_shape, offset = get_output_space(img1, [img2], [H])

# Warp images into output sapce
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()
```





[12]:
```
# 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('RANSAC Robust Panorama')
plt.show()

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

Question 4.1 (5 points) Discuss the purpose of RANSAC in improving the estimation of the affine transformation matrix.

Your Answer: Write your solution in this markdown cell.

The simple reason why RANSAC helps with estimating the affine transformation matrix is because of the main purpose of RANSAC itself, which is to reduce the impacts of outliers in data, and so obviously here, when we try to 'fit' the affine matrix to our data, we use RANSAC to reduce the impact of noise to RANSAC, which is what is was designed for.

1.7 Part 5 Histogram of Oriented Gradients (HOG) (20 points + 0 points)

In the above code, you are using the `simple_descriptor`, and in this section, you are going to implement a simplified version of HOG descriptor. HOG stands for Histogram of Oriented Gradients. In HOG descriptor, the distribution (histograms) of the directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of a gradient is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions. The steps of HOG are: 1. Compute the gradient image in x and y directions * Use the sobel filter provided by skimage.filters 2. Compute gradient histograms * Divide image into cells (this part is done for you – see Scikit documentation for `view_as_blocks` for more info) * Calculate histogram of gradients for each celll 3. Flatten 3D matrix of histograms into feature vector 4. Normalize flattened histogram feature vector by L2 norm

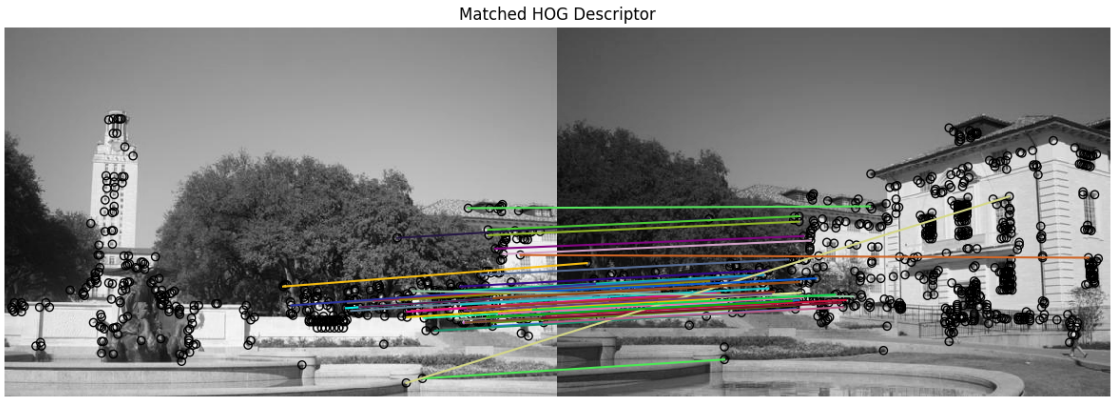
Implement **`hog_descriptor`** in `panorama.py` and run through the following code to get a panorama image.

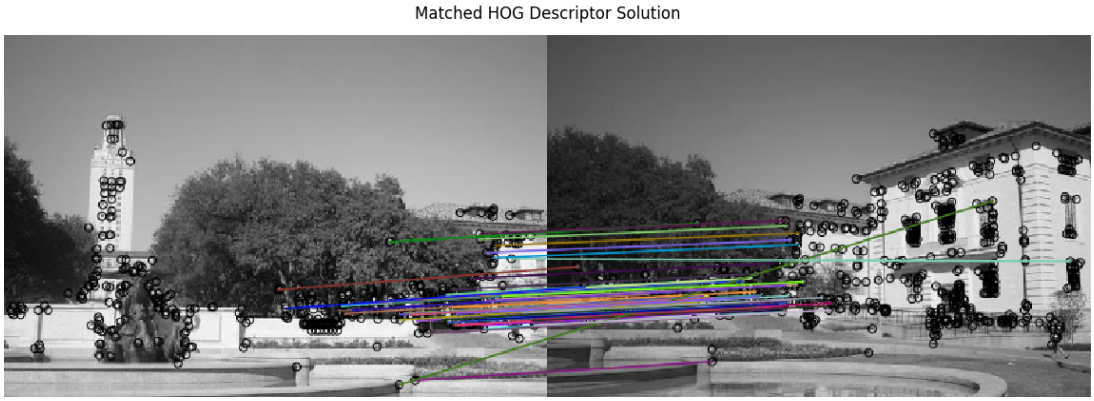
[13]:
```
from panorama import hog_descriptor

img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# Detect keypoints in both images
hog_keypoints1 = corner_peaks(harris_corners(img1, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)
hog_keypoints2 = corner_peaks(harris_corners(img2, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)

print("HoG keypoints1 shape = ", hog_keypoints1.shape)
print("HoG keypoints2 shape = ", hog_keypoints2.shape)
```

```
HoG keypoints1 shape =  (396, 2)
HoG keypoints2 shape =  (627, 2)
```

[14]:
```
# Set seed to compare output against solution
np.random.seed(455)

# Extract features from the corners
hog_desc1 = describe_keypoints(img1, hog_keypoints1,
                               desc_func=hog_descriptor,
                               patch_size=16)
hog_desc2 = describe_keypoints(img2, hog_keypoints2,
                               desc_func=hog_descriptor,
                               patch_size=16)

print("HoG desc1 shape = ", hog_desc1.shape)
print("HoG desc2 shape = ", hog_desc2.shape)

# Match descriptors in image1 to those in image2
hog_matches = match_descriptors(hog_desc1, hog_desc2, 0.7)

# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
ax.axis('off')
plt.title('Matched HOG Descriptor')
plot_matches(ax, img1, img2, hog_keypoints1, hog_keypoints2, hog_matches)
plt.show()

plt.imshow(imread('solution_hog.png'))
plt.axis('off')
plt.title('Matched HOG Descriptor Solution')
plt.show()
```

```
HoG desc1 shape =  (396, 36)
HoG desc2 shape =  (627, 36)
```



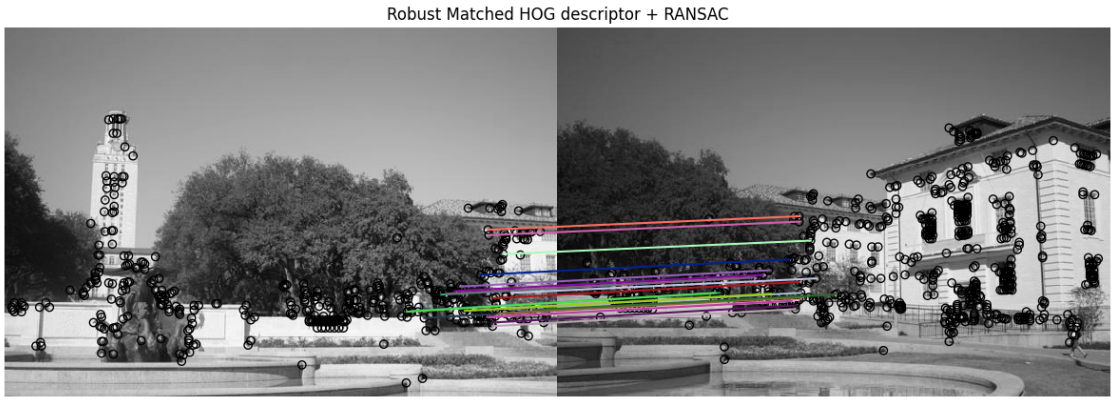


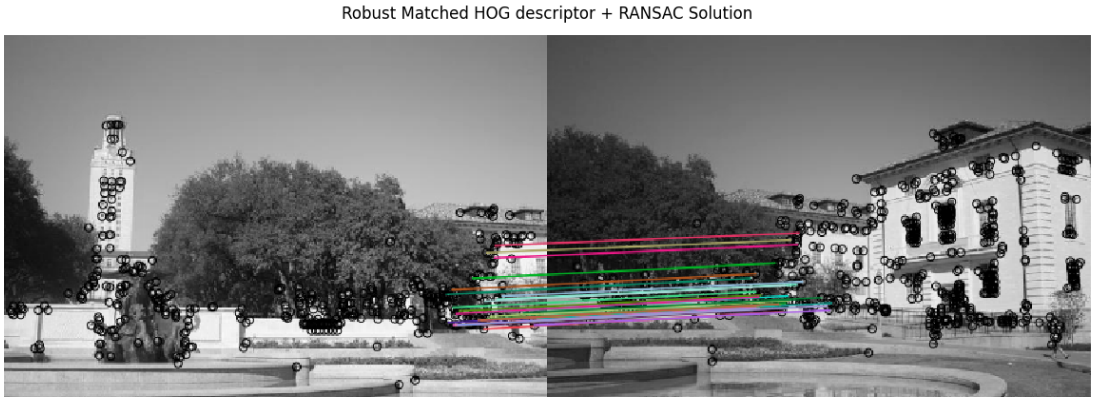
Once we've described our keypoints with the HOG descriptor and have found matches between these keypoints, we can use RANSAC to select robust matches for computing the tranasformtion matrix.

[15]:
```
from panorama import ransac

# Set seed to compare output against solution image
np.random.seed(455)

H, robust_matches = ransac(hog_keypoints1, hog_keypoints2, hog_matches,␣
↪threshold=1)
print("Robust matches shape = ", robust_matches.shape)
print("H = \n", H)

# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
```

```
plot_matches(ax, img1, img2, hog_keypoints1, hog_keypoints2, robust_matches)
plt.axis('off')
plt.title('Robust Matched HOG descriptor + RANSAC')
plt.show()

plt.imshow(imread('solution_hog_ransac.png'))
plt.axis('off')
plt.title('Robust Matched HOG descriptor + RANSAC Solution')
plt.show()
```

```
Robust matches shape =  (22, 2)
H =
 [[ 1.03725954e+00  8.10534695e-02  0.00000000e+00]
 [-2.77254677e-02  1.05281841e+00  0.00000000e+00]
 [ 1.43383896e+01  2.38151847e+02  1.00000000e+00]]
```





Now we use the computed transformation matrix H to warp our images and produce our panorama.

[16]:
```
output_shape, offset = get_output_space(img1, [img2], [H])

# Warp images into output sapce
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()
```

[17]:
```
# 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('HOG Descriptor + RANSAC Robust Panorama')
```

```
plt.show()

plt.imshow(imread('solution_hog_panorama.png'))
plt.axis('off')
plt.title('HOG Descriptor + RANSAC Robust Panorama Solution')
plt.show()
```

1.8 Extra Credit: Better Image Merging

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 from the panorama.

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** in **panorama.py** implement the linear blending scheme to make the panorama look more natural. This extra credit can be worth up to 1% of your final grade.

[18]:
```
from panorama import linear_blend

img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# Set seed to compare output against solution
np.random.seed(455)

# 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=hog_descriptor,
                           patch_size=16)
ec1_desc2 = describe_keypoints(img2, ec1_keypoints2,
                           desc_func=hog_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 sapce
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, 36)
EC1 desc2 shape =  (627, 36)
Robust matches shape =  (22, 2)
H =
 [[ 1.03725954e+00  8.10534695e-02  0.00000000e+00]
 [-2.77254677e-02  1.05281841e+00  0.00000000e+00]
 [ 1.43383896e+01  2.38151847e+02  1.00000000e+00]]
Output shape: [443 919]
Offset: [-2.71277307  0.        ]
```

Linear Blend

Linear Blend Solution

1.9 Extra Credit: Stitching Multiple Images

Implement **`stitch_multiple_images`** in `panorama.py` to stitch together an ordered chain of images. This extra credit can be worth up to 1% of your final grade.

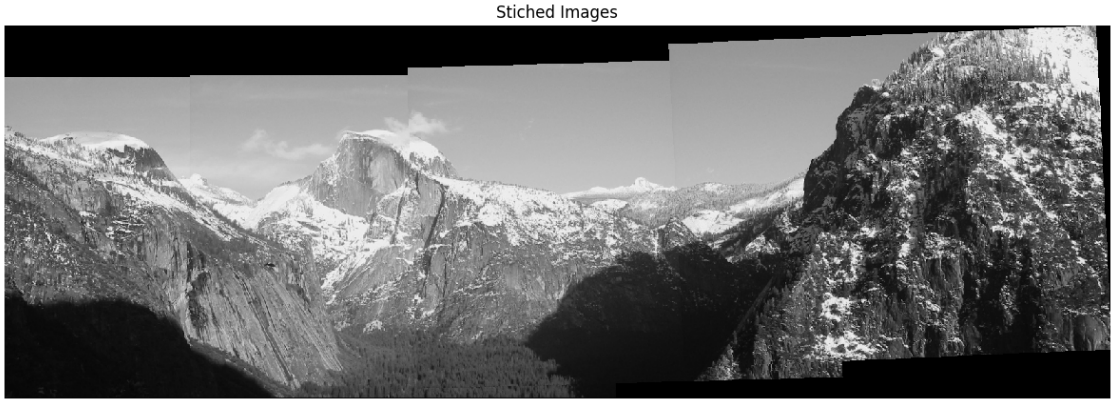
Given a sequence of m images $(I_1, I_2, ..., I_m)$, take every neighboring pair of images and compute the transformation matrix which converts points from the coordinate frame of I_{i+1} to the frame of I_i. Then, select a reference image I_{ref}, which is the first or left-most image in the chain. 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.

-Hint: - If you are confused, you may want to review the Linear Algebra slides on how to combine the effects of multiple transformation matrices. - The inverse of transformation matrix has the reverse effect. Please use `numpy.linalg.inv` function whenever you want to compute matrix inverse.

[19]:
```
from panorama import stitch_multiple_images

# Set seed to compare output against solution
np.random.seed(455)

# 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)
```

[20]:
```
# Visualize final panorama image
plt.imshow(panorama)
plt.axis('off')
plt.title('Stiched Images')
plt.show()
```



[21]:
```
plt.imshow(imread('solution_stitched_images.png'))
plt.axis('off')
plt.title('Stiched Images Solution')
```

```
plt.show()
```

seam_carving

May 3, 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/assignment2/'
FOLDERNAME = 'cse455/assignments/assignment2/'
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/cse455/assignments/assignment2
```

1 Seam Carving (100 points + 10 points)

This assignment covers seam carving for the purpose of content-aware resizing.

The material presented here is inspired from: - paper on seam carving: http://graphics.cs.cmu.edu/courses/15-463/2007_fall/hw/proj2/imret.pdf - tutorial: http://cs.brown.edu/courses/cs129/results/proj3/taox - tutorial: http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/f07/proj2/www/wwedler/

Don't hesitate to check these links if you have any doubt on the seam carving process.

The whole seam carving process was covered in lecture, please refer to the slides for more details to the different concepts introduced here.

Grading: Each part will be graded based on the implementation (marked by autograder) and inline question answers. The total score shown at each part is in a format (implementation score + inline question score). For example, if you see (20 points + 5 points), it means 20 points will go to your implementation and 5 points will go to your inline question answers.

[2]:
```
from __future__ import print_function

# Setup
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from skimage import color

from time import time
from IPython.display import HTML

%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'

fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)

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

```
<Figure size 1850x1050 with 0 Axes>
```

1.1 Image Reducing using Seam Carving

Seam carving is an algorithm for content-aware image resizing.
To understand all the concepts in this homework, make sure to read the slides from lecture again.

[3]:
```
from skimage import io, util

# Load image
img = io.imread('imgs/broadway_tower.jpg')
img = util.img_as_float(img)

plt.title('Original Image')
plt.imshow(img)
plt.show()
```

1.1.1 Energy function (5 points + 5 points)

We will now implemented the `energy_function` to compute the energy of the image.
The energy at each pixel is the sum of: - absolute value of the gradient in the x direction - absolute value of the gradient in the y direction

The function should take less than 0.1 seconds to compute.

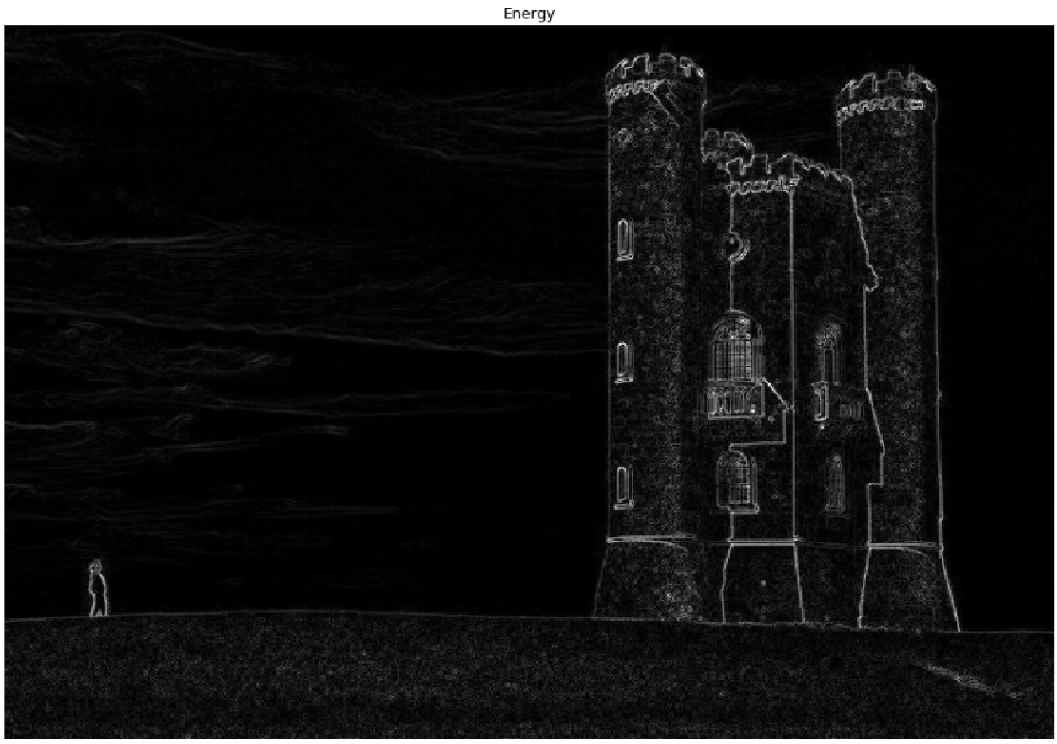
[4]:
```
from seam_carving import energy_function

test_img = np.array([[1.0, 2.0, 1.5],
                     [3.0, 1.0, 2.0],
                     [4.0, 0.5, 3.0]])
test_img = np.stack([test_img] * 3, axis=2)
assert test_img.shape == (3, 3, 3)


# Compute energy function
test_energy = energy_function(test_img)

solution_energy = np.array([[3.0, 1.25,  1.0],
                            [3.5, 1.25, 1.75],
                            [4.5,  1.0,  3.5]])
```

```
print("Image (channel 0):")
print(test_img[:, :, 0])

print("Energy:")
print(test_energy)
print("Solution energy:")
print(solution_energy)

assert np.allclose(test_energy, solution_energy)
```

```
Image (channel 0):
[[1.  2.  1.5]
 [3.  1.  2. ]
 [4.  0.5 3. ]]
Energy:
[[3.   1.25 1.  ]
 [3.5  1.25 1.75]
 [4.5  1.   3.5 ]]
Solution energy:
[[3.   1.25 1.  ]
 [3.5  1.25 1.75]
 [4.5  1.   3.5 ]]
```

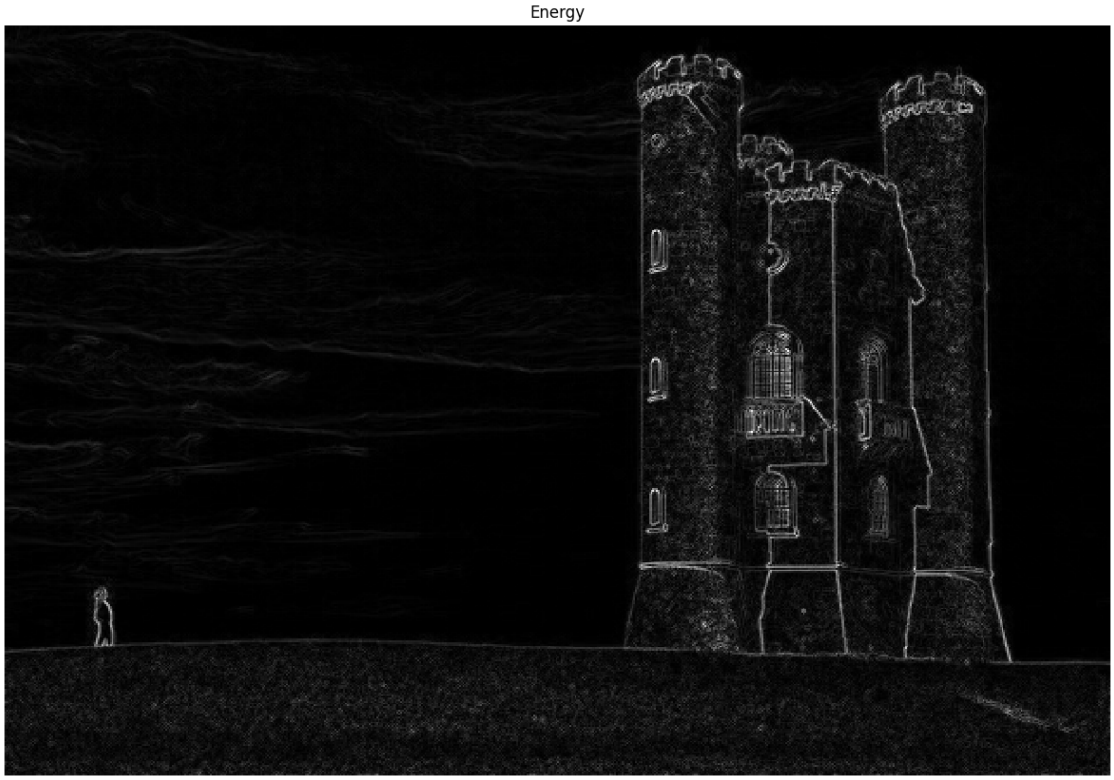
Now, we can visualize your energy function. The expected output is:

[5]:
```
# Expected visualization of energy function
img_soln = io.imread('imgs/energy_soln.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```



[6]:
```
# Compute energy function
start = time()
energy = energy_function(img)
end = time()

print("Computing energy function: %f seconds." % (end - start))

plt.title('Energy')
plt.axis('off')
plt.imshow(energy)
plt.show()
```

```
Computing energy function: 0.006674 seconds.
```



Question 1 (5 points) How is the energy of a pixel determined in the context of seam carving?

Your Answer: Write your solution in this markdown cell.

The energy of a pixel is determined in exactly the way that we just did in the function above: we calculate the gradients in the x and y direction for each pixel, take their absolute values (as we energy should be represented by drastic change in either the posistive or negative direction), and add them together. We have spent so much time explaining that keypoints and descriptors are places such as edges, points, cornerns, and blobs, place where the gradient is generally large, so this should not be surprising.

1.1.2 Compute cost (10 points + 5 points)

Now implement the function `compute_cost`. Starting from the energy map, we'll go from the first row of the image to the bottom and compute the minimal cost at each pixel.

We'll use dynamic programming to compute the cost line by line starting from the first row.

The function should take less than 0.1 seconds to complete.

[7]:
```
from seam_carving import compute_cost

# Let's first test with a small example
```

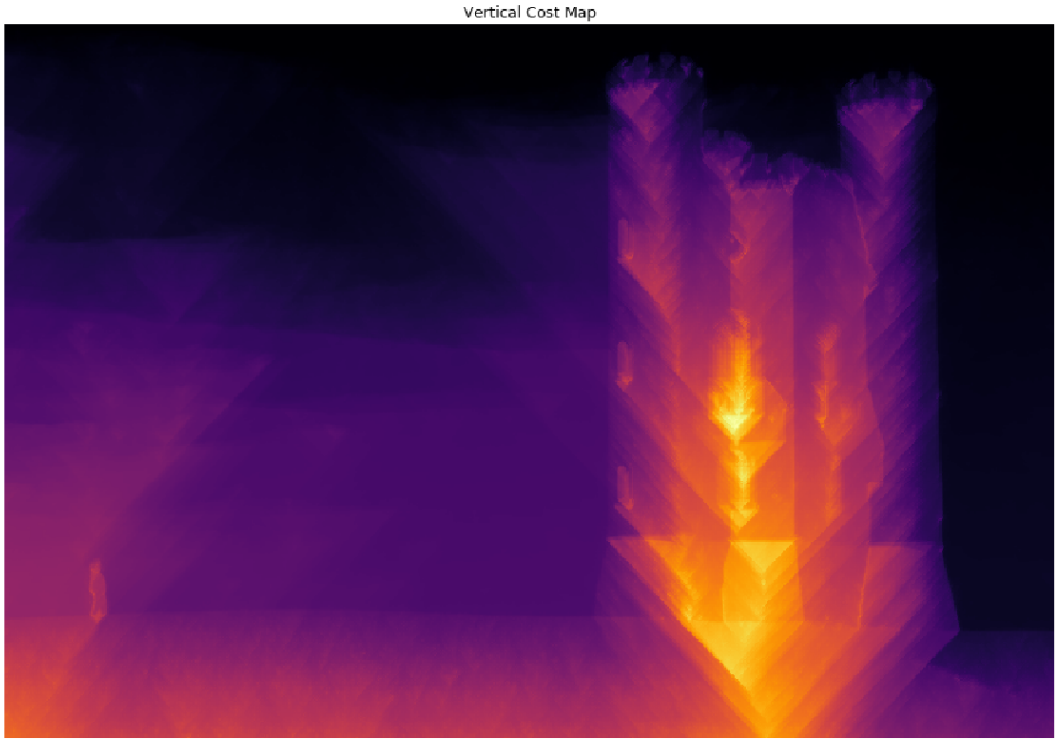
```
test_energy = np.array([[1.0, 2.0, 1.5],
                        [3.0, 1.0, 2.0],
                        [4.0, 0.5, 3.0]])

solution_cost = np.array([[1.0, 2.0, 1.5],
                          [4.0, 2.0, 3.5],
                          [6.0, 2.5, 5.0]])

solution_paths = np.array([[ 0,  0,  0],
                           [ 0, -1,  0],
                           [ 1,  0, -1]])

# Vertical Cost Map
vcost, vpaths = compute_cost(_, test_energy, axis=1)  # don't need the first␣
 ↪argument for compute_cost

print("Energy:")
print(test_energy)

print("Cost:")
print(vcost)
print("Solution cost:")
print(solution_cost)

print("Paths:")
print(vpaths)
print("Solution paths:")
print(solution_paths)
```

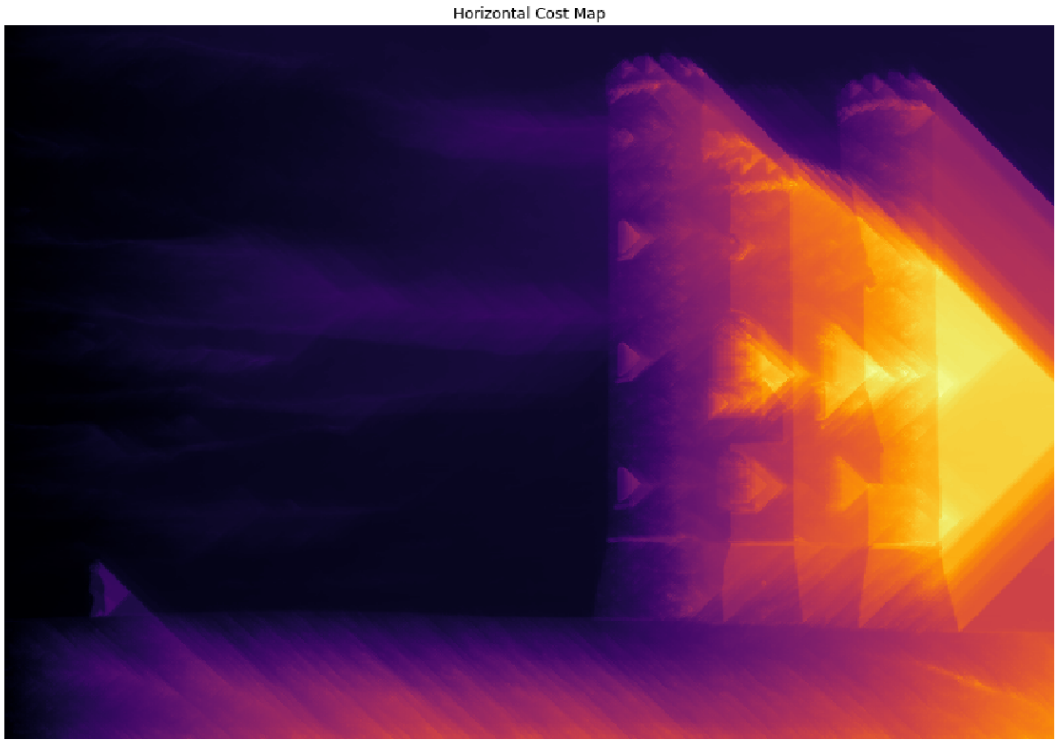
```
Energy:
[[1.  2.  1.5]
 [3.  1.  2. ]
 [4.  0.5 3. ]]
Cost:
[[1.  2.  1.5]
 [4.  2.  3.5]
 [6.  2.5 5. ]]
Solution cost:
[[1.  2.  1.5]
 [4.  2.  3.5]
 [6.  2.5 5. ]]
Paths:
[[ 0  0  0]
 [ 0 -1  0]
 [ 1  0 -1]]
Solution paths:
[[ 0  0  0]
```

```
 [ 0 -1  0]
 [ 1  0 -1]]
```

Now, we can visualize a map of the vertical and horizontal costs. The expected outputs are:

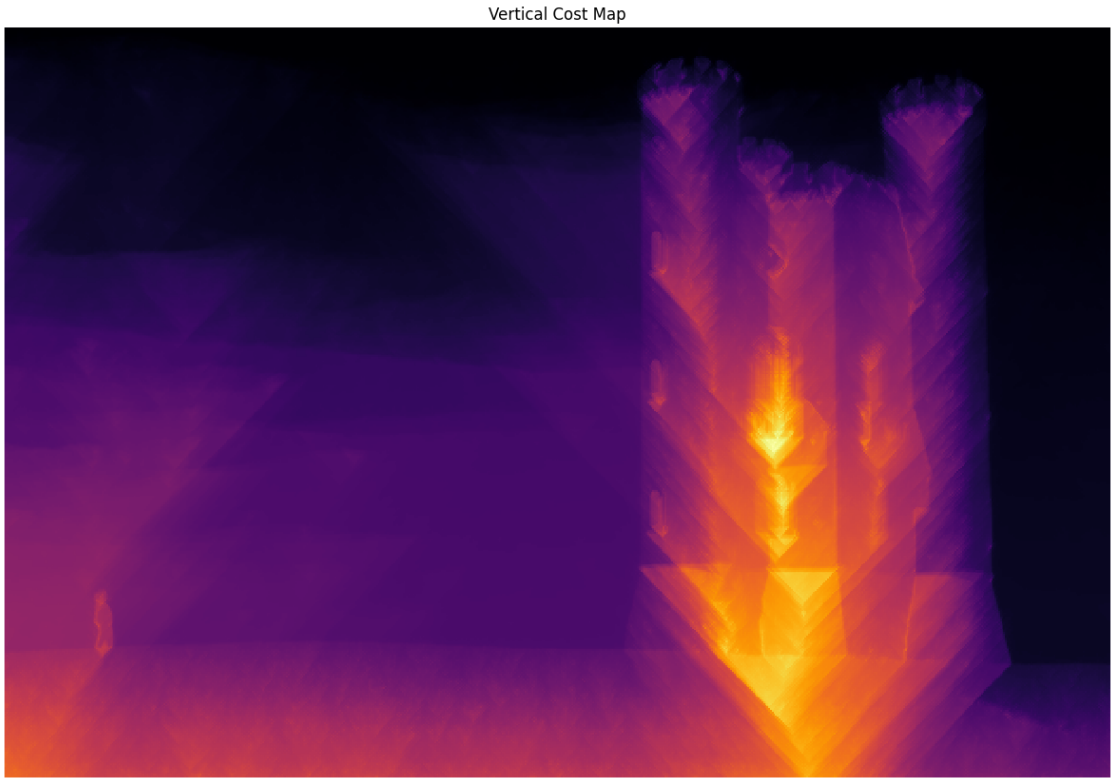
[8]:
```
# Expected visualization of vertical cost
img_soln = io.imread('imgs/vertical_cost_soln.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()

# Expected visualization of horizontal cost
img_soln = io.imread('imgs/horizontal_cost_soln.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```





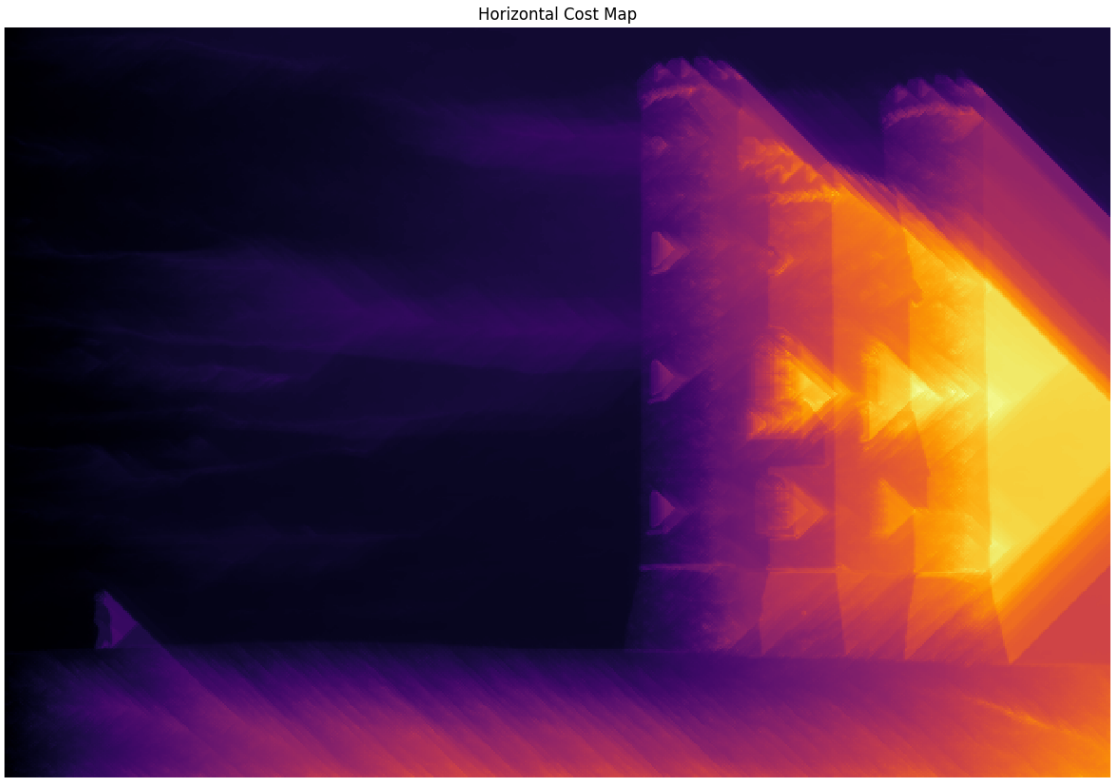
[9]:
```
# Vertical Cost Map
start = time()
vcost, _ = compute_cost(_, energy, axis=1)  # don't need the first argument for␣
 ↪compute_cost
end = time()

print("Computing vertical cost map: %f seconds." % (end - start))

plt.title('Vertical Cost Map')
plt.axis('off')
plt.imshow(vcost, cmap='inferno')
plt.show()
```

```
Computing vertical cost map: 0.075907 seconds.
```



[10]:
```
# Horizontal Cost Map
start = time()
hcost, _ = compute_cost(_, energy, axis=0)
end = time()

print("Computing horizontal cost map: %f seconds." % (end - start))

plt.title('Horizontal Cost Map')
plt.axis('off')
plt.imshow(hcost, cmap='inferno')
plt.show()
```

```
Computing horizontal cost map: 0.060632 seconds.
```



Question 2 (5 points) Explain the purpose of computing a cost map in seam carving.

Your Answer: Write your solution in this markdown cell.

The whole point of the cost map is that it is a concise but comphrensive way to understand the least and most important pieces of an image, which therefore helps us figure out the best seams to remove in either direction, and greatly speed up the intial and later computations.

1.2 Finding optimal seams

Using the cost maps we found above, we can determine the seam with the lowest energy in the image.
We can then remove this optimal seam, and repeat the process until we obtain a desired width.

1.2.1 Backtrack seam (5 points + 0 points)

Implement function `backtrack_seam`.

[11]:
```
from seam_carving import backtrack_seam

# Let's first test with a small example
cost = np.array([[1.0, 2.0, 1.5],
                 [4.0, 2.0, 3.5],
                 [6.0, 2.5, 5.0]])
```

```
paths = np.array([[ 0,  0,  0],
                  [ 0, -1,  0],
                  [ 1,  0, -1]])


# Vertical Backtracking

end = np.argmin(cost[-1])
seam_energy = cost[-1, end]
seam = backtrack_seam(vpaths, end)

print('Seam Energy:', seam_energy)
print('Seam:', seam)

assert seam_energy == 2.5
assert np.allclose(seam, [0, 1, 1])
```

```
Seam Energy: 2.5
Seam: [0 1 1]
```

Now, we can visualize the optimal vertical seam. The expected output is:

[12]:
```
# Expected visualization of optimal vertical seam
img_soln = io.imread('imgs/optimal_vertical_seam_soln.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```

[13]:
```
vcost, vpaths = compute_cost(img, energy)

# Vertical Backtracking
start = time()
end = np.argmin(vcost[-1])
seam_energy = vcost[-1, end]
seam_ = backtrack_seam(vpaths, end)
end = time()

print("Backtracking optimal seam: %f seconds." % (end - start))
print('Seam Energy:', seam_energy)

# Visualize seam
vseam = np.copy(img)
for row in range(vseam.shape[0]):
    vseam[row, seam_[row], :] = np.array([1.0, 0, 0])

plt.title('Vertical Seam')
plt.axis('off')
plt.imshow(vseam)
plt.show()
```

```
Backtracking optimal seam: 0.004603 seconds.
Seam Energy: 2.4323654901960783
```

In the image above, the optimal vertical seam (minimal cost) goes through the portion of sky without any cloud, which yields the lowest energy.

1.2.2 Reduce (25 points + 0 points)

We can now use the function `backtrack` and `remove_seam` iteratively to reduce the size of the image through **seam carving**.

Each reduce can take around 10 seconds to compute, depending on your implementation. If it's too long, try to vectorize your code in `compute_cost` to only use one loop.

[14]:
```
from seam_carving import reduce

# Let's first test with a small example
test_img = np.arange(9, dtype=np.float64).reshape((3, 3))
test_img = np.stack([test_img, test_img, test_img], axis=2)
assert test_img.shape == (3, 3, 3)

cost = np.array([[1.0, 2.0, 1.5],
```

```
                [4.0, 2.0, 3.5],
                [6.0, 2.5, 5.0]])

paths = np.array([[ 0,  0,  0],
                  [ 0, -1,  0],
                  [ 1,  0, -1]])

out_remove_seam = np.asarray([[[1., 1., 1.],[2., 2., 2.]],\
                              [[3., 3., 3.],[5., 5., 5.]],\
                              [[6., 6., 6.],[8., 8., 8.]]])

# Reduce image width
W_new = 2

# We force the cost and paths to our values
out = reduce(test_img, W_new, cfunc=lambda x, y: (cost, paths), bfunc=lambda␣
 ↪x,y: seam, rfunc=lambda x,y: out_remove_seam)

print("Original image (channel 0):")
print(test_img[:, :, 0])
print("Reduced image (channel 0): we see that seam [0, 4, 7] is removed")
print(out[:, :, 0])

assert np.allclose(out[:, :, 0], np.array([[1, 2], [3, 5], [6, 8]]))
```

```
Original image (channel 0):
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Reduced image (channel 0): we see that seam [0, 4, 7] is removed
[[1. 2.]
 [3. 5.]
 [6. 8.]]
```

Now, we can visualize resizing the width from 600 to 400. The expected output is:

[15]:
```
# Expected visualization of resizing the width from 600 to 400.
img_soln = io.imread('imgs/resize_width_400.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```


Original
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


Resized
0
50
100
150
200
250
300
350
400
0
50
100
150
200
250
300
350
400

[16]:
```
# Reduce image width
H, W, _ = img.shape
W_new = 400

start = time()
out = reduce(img, W_new)
end = time()

print("Reducing width from %d to %d: %f seconds." % (W, W_new, end - start))

plt.subplot(2, 1, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(2, 1, 2)
plt.title('Resized')
plt.imshow(out)

plt.show()
```

```
Reducing width from 640 to 400: 19.385664 seconds.
```


Original
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

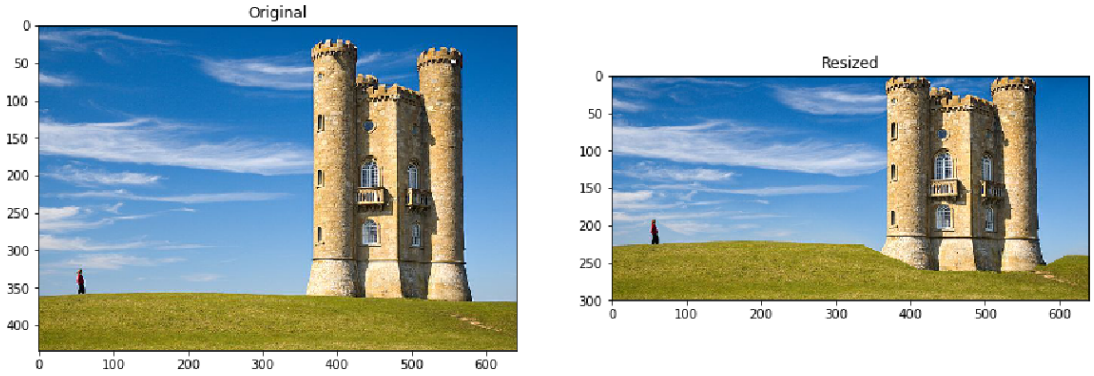

Resized
0
50
100
150
200
250
300
350
400
0
50
100
150
200
250
300
350

We observe that resizing from width 640 to width 400 conserves almost all the important part of the image (the person and the castle), where a standard resizing would have compressed everything.

All the vertical seams removed avoid the person and the castle.

Now, we can visualize resizing the height from 434 to 300. The expected output is:

[17]:
```
# Expected visualization of resizing the height from 434 to 300.
img_soln = io.imread('imgs/resize_height_300.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```



[18]:
```
# Reduce image height
H, W, _ = img.shape
H_new = 300

start = time()
out = reduce(img, H_new, axis=0)
end = time()

print("Reducing height from %d to %d: %f seconds." % (H, H_new, end - start))

plt.subplot(1, 2, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(1, 2, 2)
plt.title('Resized')
plt.imshow(out)

plt.show()
```

```
Reducing height from 434 to 300: 5.131153 seconds.
```

For reducing the height, we observe that the result does not look as nice.

The issue here is that the castle is on all the height of the image, so most horizontal seams will go through it.
Interestingly, we observe that most of the grass is not removed. This is because the grass has small variation between neighboring pixels (in a kind of noisy pattern) that make it high energy.
The seams removed go through the sky on the left, go under the castle to remove some grass and then back up in the low energy blue sky.

1.3 Image Enlarging

1.3.1 Enlarge naive (10 points + 0 points)

We now want to tackle the reverse problem of enlarging an image.
One naive way to approach the problem would be to duplicate the optimal seam iteratively until we reach the desired size.

[43]:
```python
from seam_carving import enlarge_naive

# Let's first test with a small example
test_img = np.arange(9, dtype=np.float64).reshape((3, 3))
test_img = np.stack([test_img, test_img, test_img], axis=2)
assert test_img.shape == (3, 3, 3)

cost = np.array([[1.0, 2.0, 1.5],
                 [4.0, 2.0, 3.5],
                 [6.0, 2.5, 5.0]])

paths = np.array([[ 0,  0,  0],
                  [ 0, -1,  0],
                  [ 1,  0, -1]])
```

```
out_duplicate_seam = np.asarray( [[[0., 0., 0.],[0., 0., 0.],[1., 1., 1.],[2.,␣
 ↪2., 2.]],\
                                  [[3., 3., 3.],[4., 4., 4.],[4., 4., 4.],[5.,␣
 ↪5., 5.]],\
                                  [[6., 6., 6.],[7., 7., 7.],[7., 7., 7.],[8.,␣
 ↪8., 8.]]])

# Increase image width
W_new = 4

# We force the cost and paths to our values
out = enlarge_naive(test_img, W_new, cfunc=lambda x, y: (cost, paths),␣
 ↪bfunc=lambda x,y: seam , dfunc=lambda x,y:out_duplicate_seam)

print("Original image (channel 0):")
print(test_img[:, :, 0])
print("Enlarged image (channel 0): we see that seam [0, 4, 7] is duplicated")
print(out[:, :, 0])

assert np.allclose(out[:, :, 0], np.array([[0, 0, 1, 2], [3, 4, 4, 5], [6, 7,␣
 ↪7, 8]]))
```

```
Original image (channel 0):
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Enlarged image (channel 0): we see that seam [0, 4, 7] is duplicated
[[0. 0. 1. 2.]
 [3. 4. 4. 5.]
 [6. 7. 7. 8.]]
```

Now, we can visualize naively enlarging the width from 640 to 800. The expected output is:

[44]:
```
# Expected visualization of naively enlarging the width from 640 to 800.
img_soln = io.imread('imgs/enlarge_naive_width_800.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```

[45]:
```
W_new = 800

# This is a naive implementation of image enlarging
# which iteratively computes energy function, finds optimal seam
# and duplicates it.
# This process will create a stretching artifact by choosing the same seam
start = time()
enlarged = enlarge_naive(img, W_new)
end = time()

# Can take around 20 seconds
print("Enlarging(naive) width from %d to %d: %f seconds." \
      % (W, W_new, end - start))

plt.imshow(enlarged)
plt.show()
```

```
Enlarging(naive) width from 640 to 800: 7.426157 seconds.
```

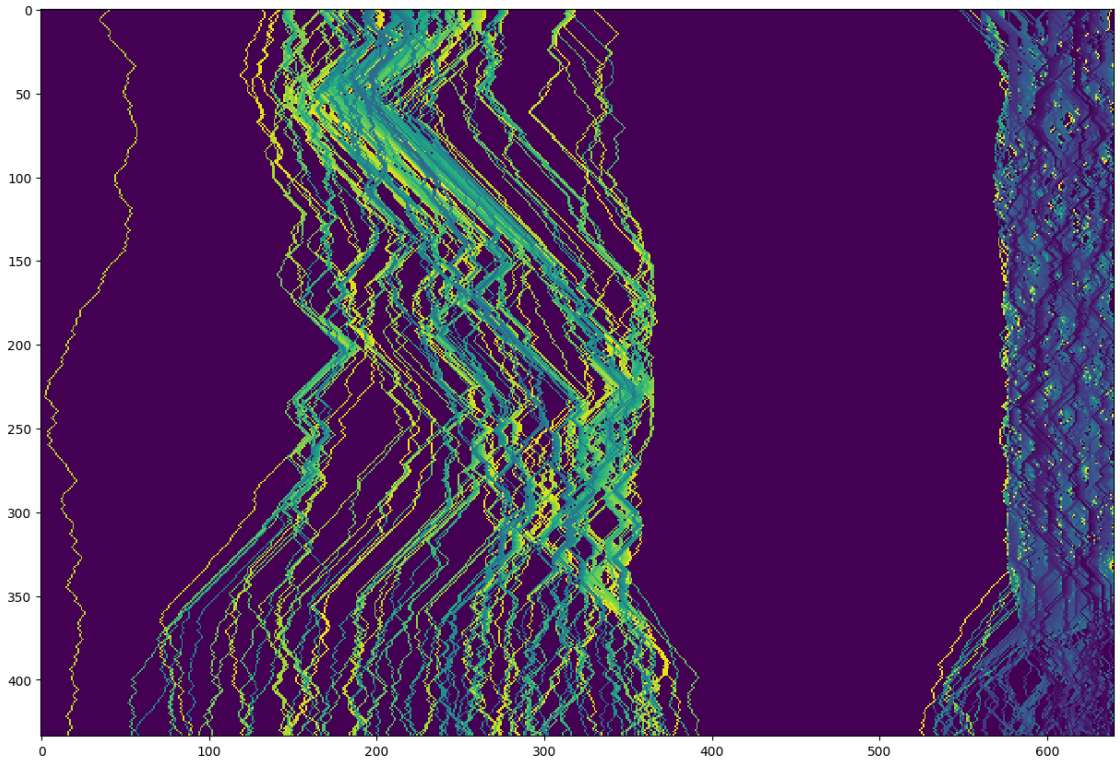
The issue with `enlarge_naive` is that the same seam will be selected again and again, so this low energy seam will be the only to be duplicated.

Another way to get k different seams is to apply the process we used in function `reduce`, and keeping track of the seams we delete progressively. The function `find_seams(image, k)` will find the top k seams for removal iteratively.

The inner workings of the function are a bit tricky so we've implemented it for you, but you should go into the code and understand how it works.
This should also help you for the implementation of `enlarge`.

[22]:
```
from seam_carving import find_seams

# Alternatively, find k seams for removal and duplicate them.
start = time()
seams = find_seams(img, W_new - W)
end = time()

# Can take around 10 seconds
print("Finding %d seams: %f seconds." % (W_new - W, end - start))

plt.imshow(seams, cmap='viridis')
plt.show()
```

```
Finding 160 seams: 6.096428 seconds.
```



1.3.2 Enlarge (25 points + 0 points)

We can see that all the seams found are different, and they avoid the castle and the person.

One issue we can mention is that we cannot enlarge more than we can reduce. Because of our process, the maximum enlargement is the width of the image `W` because we first need to find `W` different seams in the image.

One effect we can see on this image is that the blue sky at the right of the castle can only be enlarged x2. The concentration of seams in this area is very strong.
We can also note that the seams at the right of the castle have a blue color, which means they have low value and were removed in priority in the seam selection process.

[23]:
```
from seam_carving import enlarge

# Let's first test with a small example
test_img = np.array([[0.0, 1.0, 3.0],
                     [0.0, 1.0, 3.0],
                     [0.0, 1.0, 3.0]])
#test_img = np.arange(9, dtype=np.float64).reshape((3, 3))
test_img = np.stack([test_img, test_img, test_img], axis=2)
assert test_img.shape == (3, 3, 3)

# Increase image width
```

```
W_new = 5

out_naive = enlarge_naive(test_img, W_new)
out = enlarge(test_img, W_new)

print("Original image (channel 0):")
print(test_img[:, :, 0])
print("Enlarged naive image (channel 0): first seam is duplicated twice.")
print(out_naive[:, :, 0])
print("Enlarged image (channel 0): first and second seam are each duplicated␣
 ↪once.")
print(out[:, :, 0])

assert np.allclose(out[:, :, 0], np.array([[0, 0, 1, 1, 3], [0, 0, 1, 1, 3],␣
 ↪[0, 0, 1, 1, 3]]))
```

```
Original image (channel 0):
[[0. 1. 3.]
 [0. 1. 3.]
 [0. 1. 3.]]
Enlarged naive image (channel 0): first seam is duplicated twice.
[[0. 0. 0. 1. 3.]
 [0. 0. 0. 1. 3.]
 [0. 0. 0. 1. 3.]]
Enlarged image (channel 0): first and second seam are each duplicated once.
[[0. 0. 1. 1. 3.]
 [0. 0. 1. 1. 3.]
 [0. 0. 1. 1. 3.]]
```

Now, we can visualize enlarging the width from 640 to 800. The expected output is:

[24]:
```
# Expected visualization of enlarging the width from 640 to 800.
img_soln = io.imread('imgs/enlarge_width_800.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```

[25]:
```
W_new = 800

start = time()
out = enlarge(img, W_new)
end = time()

# Can take around 20 seconds
print("Enlarging width from %d to %d: %f seconds." \
```

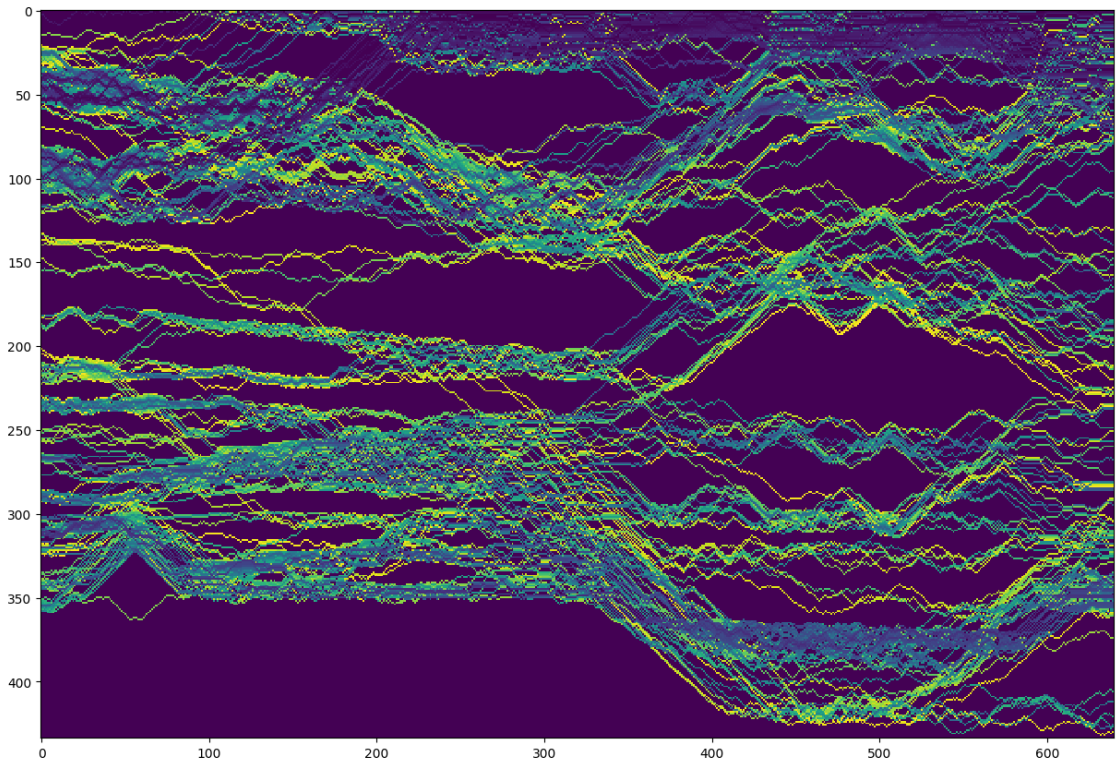
```
      % (W, W_new, end - start))

plt.subplot(2, 1, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(2, 1, 2)
plt.title('Resized')
plt.imshow(out)

plt.show()
```

```
Enlarging width from 640 to 800: 10.596401 seconds.
```

[26]:
```
# Map of the seams for horizontal seams.
start = time()
seams = find_seams(img, W_new - W, axis=0)
end = time()

# Can take around 15 seconds
print("Finding %d seams: %f seconds." % (W_new - W, end - start))
```

```
plt.imshow(seams, cmap='viridis')
plt.show()
```

```
Finding 160 seams: 6.511919 seconds.
```



Now, we can visualize enlarging the height from 434 to 600. The expected output is:

[27]:
```
# Expected visualization of enlarging the height from 434 to 600.
img_soln = io.imread('imgs/enlarge_height_600.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```

[28]:
```
H_new = 600

start = time()
out = enlarge(img, H_new, axis=0)
end = time()

# Can take around 20 seconds
print("Enlarging height from %d to %d: %f seconds." \
      % (H, H_new, end - start))

plt.subplot(1, 2, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(1, 2, 2)
plt.title('Resized')
plt.imshow(out)

plt.show()
```

```
Enlarging height from 434 to 600: 13.668958 seconds.
```

As you can see in the example above, the sky above the castle has doubled in size, the grass below has doubled in size but we still can't reach a height of 600 by just doubling the sky and grass.
So, the algorithm then needs to enlarge the castle itself, while trying to avoid enlarging the windows for instance.

1.4 Other experiments on the image

Feel free to experiment more on this image, try different sizes to enlarge or reduce, or check what seams are chosen…

Reducing by a 2x factor often leads to weird patterns.
Enlarging by more than 2x is impossible since we only duplicate seams. One solution is to enlarge in mutliple steps (enlarge x1.4, enlarge again x1.4…)

[29]:
```
# Reduce image width
H, W, _ = img.shape
W_new = 200

start = time()
out = reduce(img, W_new)
end = time()

print("Reducing width from %d to %d: %f seconds." % (W, W_new, end - start))

plt.subplot(2, 1, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(2, 1, 2)
plt.title('Resized')
plt.imshow(out)
```

```
plt.show()
```

```
Reducing width from 640 to 200: 13.358882 seconds.
```


Original
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


Resized
0
50
100
150
200
250
300
350
400
0
50
100
150

1.5 Extra Credit: Faster `reduce`

Implement a faster version of `reduce` called `reduce_fast` in the file `seam_carving.py`.

We will have a leaderboard on gradescope with the performance of students.

The autograder tests will check that the outputs match, and run the `reduce_fast` function on a set of images with varying shapes (say between 200 and 800).

This extra credit can be worth up to 1% of your final grade.

[30]:
```
from seam_carving import reduce_fast

# Reduce image width
H, W, _ = img.shape
W_new = 400

start = time()
out = reduce(img, W_new)
end = time()

print("Normal reduce width from %d to %d: %f seconds." % (W, W_new, end -␣
 ↪start))

start = time()
out_fast = reduce_fast(img, W_new)
end = time()

print("Faster reduce width from %d to %d: %f seconds." % (W, W_new, end -␣
 ↪start))

assert np.allclose(out, out_fast), "Outputs don't match."


plt.subplot(3, 1, 1)
plt.title('Original')
plt.imshow(img)

plt.subplot(3, 1, 2)
plt.title('Resized')
plt.imshow(out)

plt.subplot(3, 1, 3)
plt.title('Faster resized')
plt.imshow(out)

plt.show()
```

```
Normal reduce width from 640 to 400: 9.927539 seconds.
Faster reduce width from 640 to 400: 9.610538 seconds.
```


Original
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


Resized
0
50
100
150
200
250
300
350
400
0
100
200
300


Faster resized
0
50
100
150
200
250
300
350
400
0
100
200
300

1.6 Reducing and enlarging on another image

Also check these outputs with another image.

[31]:
```
# Load image
img2 = io.imread('imgs/wave.jpg')
img2 = util.img_as_float(img2)

plt.title('Original Image')
plt.imshow(img2)
plt.show()
```

[32]:
```
out = reduce(img2, 300)
plt.imshow(out)
plt.show()
```

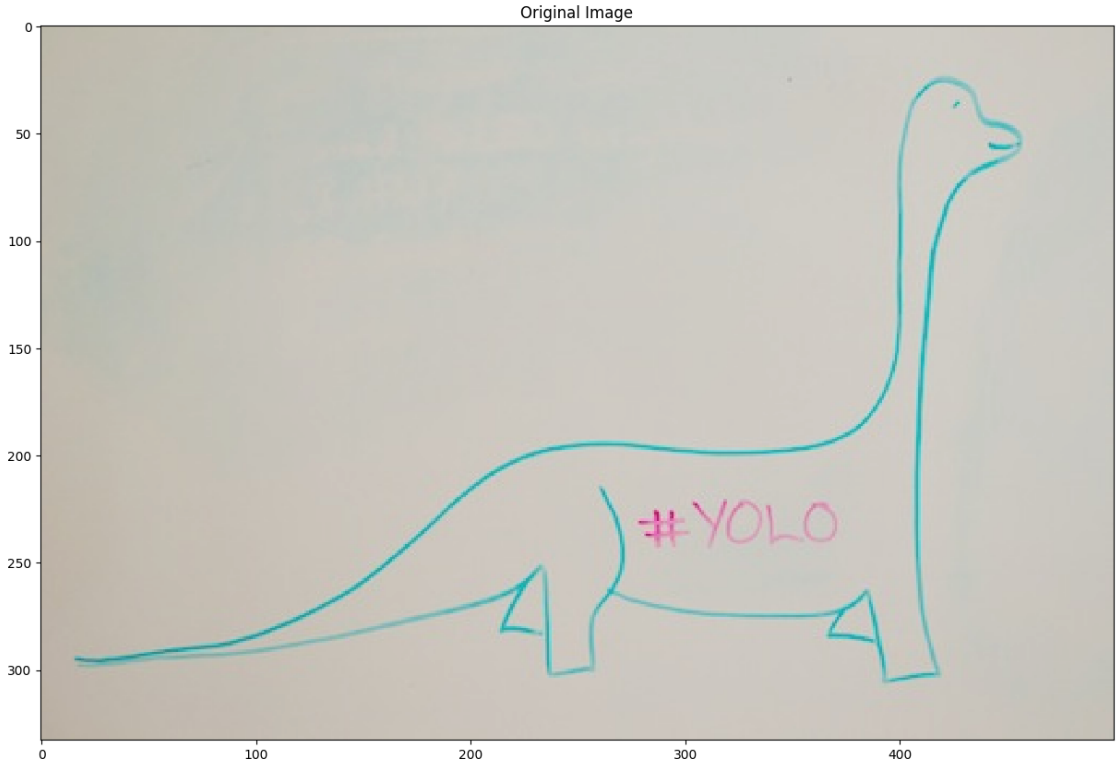
[33]:
```
out = enlarge(img2, 800)
plt.imshow(out)
plt.show()
```

1.7 Forward Energy (20 points + 0 points)

Forward energy is a solution to some artifacts that appear when images have curves for instance.

Implement the function `compute_forward_cost`. This function will replace the `compute_cost` we have been using until now.

[34]:
```
# Load image
img_yolo = io.imread('imgs/yolo.jpg')
img_yolo = util.img_as_float(img_yolo)

plt.title('Original Image')
plt.imshow(img_yolo)
plt.show()
```



[50]:
```
from seam_carving import compute_forward_cost

# Let's first test with a small example
img_test = np.array([[1.0, 1.0, 2.0],
                     [0.5, 0.0, 0.0],
                     [1.0, 0.5, 2.0]])
img_test = np.stack([img_test]*3, axis=2)
assert img_test.shape == (3, 3, 3)

energy = energy_function(img_test)

solution_cost = np.array([[0.5, 2.5, 3.0],
                          [1.0, 2.0, 3.0],
                          [2.0, 4.0, 6.0]])

solution_paths = np.array([[ 0,  0,  0],
                           [ 0, -1,  0],
                           [ 0, -1, -1]])

# Vertical Cost Map
vcost, vpaths = compute_forward_cost(img_test, energy)  # don't need the first␣
 ↪argument for compute_cost
```

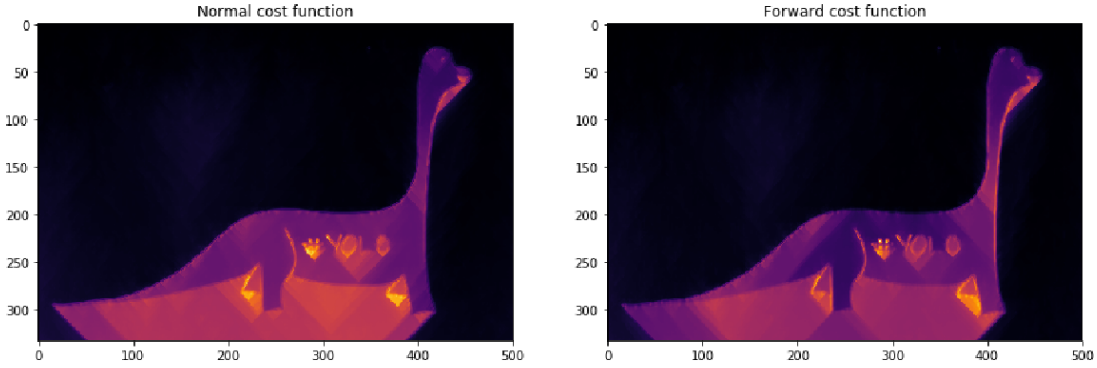
```
print("Image:")
print(color.rgb2gray(img_test))

print("Energy:")
print(energy)

print("Cost:")
print(vcost)
print("Solution cost:")
print(solution_cost)

print("Paths:")
print(vpaths)
print("Solution paths:")
print(solution_paths)

print("Difference between solution_cost and vcost:")
print(solution_cost - vcost)


assert np.allclose(solution_cost, vcost)
assert np.allclose(solution_paths, vpaths)
```

```
Image:
[[1.  1.  2. ]
 [0.5 0.  0. ]
 [1.  0.5 2. ]]
Energy:
[[0.5 1.5 3. ]
 [0.5 0.5 0. ]
 [1.  1.  3.5]]
Cost:
[[0.5 2.5 3. ]
 [1.  2.  3. ]
 [2.  4.  6. ]]
Solution cost:
[[0.5 2.5 3. ]
 [1.  2.  3. ]
 [2.  4.  6. ]]
Paths:
[[ 0  0  0]
 [ 0 -1  0]
 [ 0 -1 -1]]
Solution paths:
[[ 0  0  0]
 [ 0 -1  0]
```

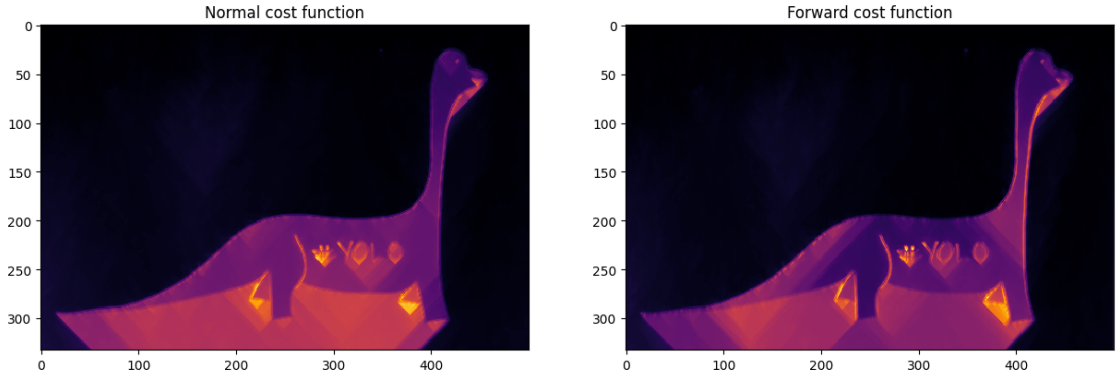
```
 [ 0 -1 -1]]
Difference between solution_cost and vcost:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Now, we can visualize the differences between the normal and forward cost functions. The expected output is:

[47]:
```
# Expected visualization of the differences between the normal and forward cost␣
↪functions.
img_soln = io.imread('imgs/normal_vs_forward_cost.png')
plt.axis('off')
plt.imshow(img_soln)
plt.show()
```

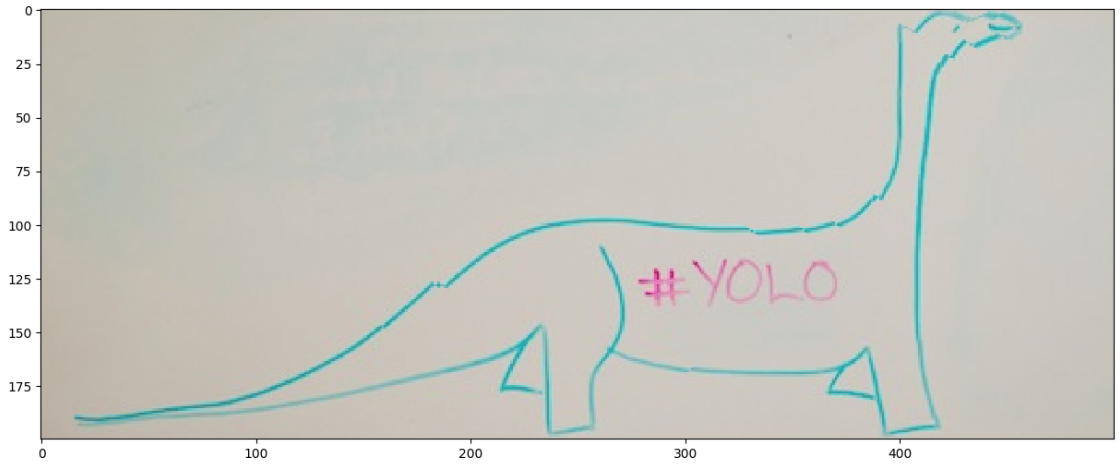


[53]:
```
from seam_carving import compute_forward_cost

energy = energy_function(img_yolo)

out, _ = compute_cost(img_yolo, energy)
plt.subplot(1, 2, 1)
plt.imshow(out, cmap='inferno')
plt.title("Normal cost function")

out, _ = compute_forward_cost(img_yolo, energy)
plt.subplot(1, 2, 2)
plt.imshow(out, cmap='inferno')
plt.title("Forward cost function")

plt.show()
```



We observe that the forward energy insists more on the curved edges of the image.

[38]:
```
from seam_carving import reduce
out = reduce(img_yolo, 200, axis=0)
plt.imshow(out)
plt.show()
```



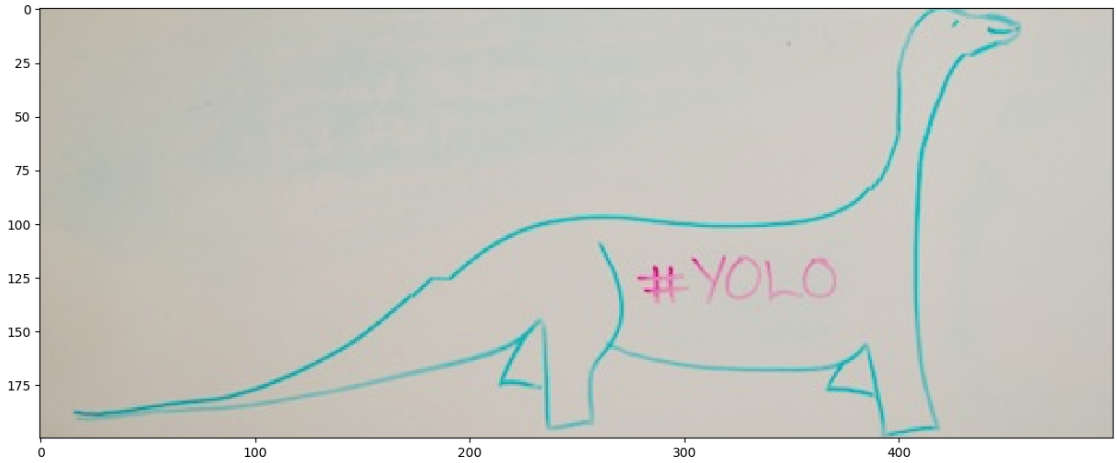
The issue with our standard `reduce` function is that it removes vertical seams without any concern for the energy introduced in the image.

In the case of the dinosaure above, the continuity of the shape is broken. The head is totally wrong for instance, and the back of the dinosaure lacks continuity.

Forward energy will solve this issue by explicitely putting high energy on a seam that breaks this continuity and introduces energy.

[39]:
```
# This step can take a very long time depending on your implementation.
out = reduce(img_yolo, 200, axis=0, cfunc=compute_forward_cost)
plt.imshow(out)
```

```
plt.show()
```

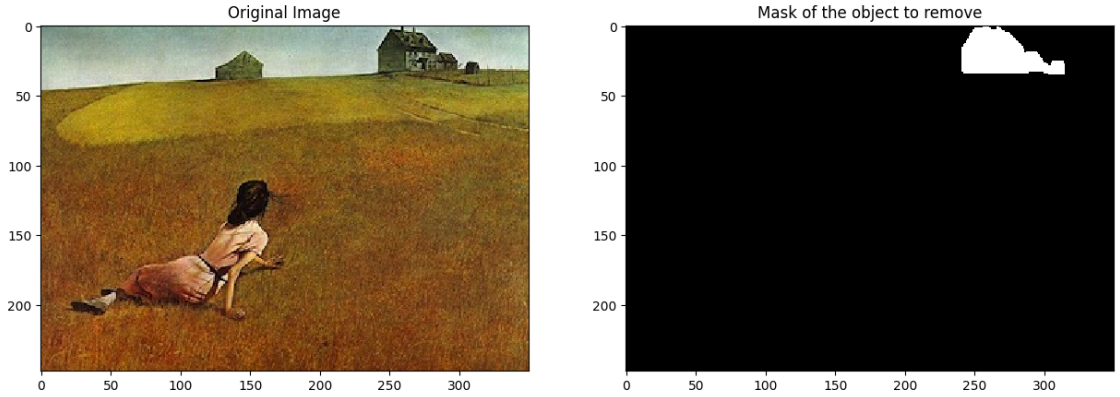


1.8 Extra Credit: Object Removal

Object removal uses a binary mask of the object to be removed.

Using the `reduce` and `enlarge` functions you wrote before, complete the function `remove_object` to output an image of the same shape but without the object to remove.

[40]:
```
# Load image
image = io.imread('imgs/wyeth.jpg')
image = util.img_as_float(image)

mask = io.imread('imgs/wyeth_mask.jpg', as_gray=True)
mask = util.img_as_bool(mask)

plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(image)

plt.subplot(1, 2, 2)
plt.title('Mask of the object to remove')
plt.imshow(mask)

plt.show()
```



[41]:
```
from seam_carving import remove_object

# Use your function to remove the object
out = remove_object(image, mask)

plt.subplot(2, 2, 1)
plt.title('Original Image')
plt.imshow(image)

plt.subplot(2, 2, 2)
plt.title('Mask of the object to remove')
plt.imshow(mask)

plt.subplot(2, 2, 3)
plt.title('Image with object removed')
plt.imshow(out)

plt.show()
```


Original Image
0
50
100
150
200
0
50
100
150
200
250
300


Mask of the object to remove
0
50
100
150
200
0
50
100
150
200
250
300


Image with object removed
0
50
100
150
200
0
50
100
150
200
250
300