Contents

1 Basic Test Results

```
ex4/
ex4/README.md
ex4/my_panorama.py
ex4/sol4.py
ex4/sol4_utils.py
ex4/videos/
ex4/videos/books.mp4
ex4 presubmission script

    Disclaimer
    ----------
    The purpose of this script is to make sure that your code is compliant
    with the exercise API and some of the requirements
    The script does not test the quality of your results.
    Don't assume that passing this script will guarantee that you will get
    a high grade in the exercise

=== Check Submission ===

README file:

tal.porezky
sol4.py
sol4_utils.py
my_panorama.py
videos/books.mp4

=== Bonus submitted? ===
no
=== IMPORTANT NOTICE ===
 If you do not see at the end "Presubmission Completed Successfully", then it means the script failed.
 This might be the result of your script running too slow and causing a timeout error.

=== Load Student Library ===

Loading...

=== Section 3.1 ===

Harris corner detector...
    Passed!
Checking structure...
    Passed!
Sample descriptor
Trying to build Gaussian pyramid...
    Passed!
Sample descriptor at the third level of the Gaussian pyramid...
Checking the descriptor type and structure...
    Passed!
Find features.
    Passed!

=== Section 3.2 ===

Match Features
    Passed!
    Passed!

=== Section 3.3 ===
```

```

Compute and apply homography
    Passed!
display matches
    Passed!

=== Section 3.4 ===

Accumulate homographies
    Passed!

=== Section 4.1 ===

Warp grayscale image
    Passed!
Compute bounding box
    Passed!


=== Presubmission Completed Successfully ===


    Please go over the output and verify that there were no failures / warnings.
    Remember that this script tested only some basic technical aspects of your implementation.
    It is your responsibility to make sure your results are actually correct and not only
    technically valid.
```

2 ex4/README.md

```
tal.porezky
sol4.py
sol4_utils.py
my_panorama.py
videos/books.mp4
```

3 ex4/my panorama.py

```python
import os
import sol4
import time


def main():
    experiments = ['books.mp4']

    for experiment in experiments:
        exp_no_ext = experiment.split('.')[0]
        os.system('mkdir dump')
        os.system('mkdir dump\%s' % exp_no_ext)
        os.system('ffmpeg -i videos\%s dump\%s\%s%%03d.jpg' % (experiment,
                                                               exp_no_ext,
                                                               exp_no_ext))

        s = time.time()
        panorama_generator = sol4.PanoramicVideoGenerator('dump/%s/' %
                                                          exp_no_ext,
                                                          exp_no_ext, 2100)
        panorama_generator.align_images(translation_only=True)
        panorama_generator.generate_panoramic_images(9)
        print(' time for %s: %.1f' % (exp_no_ext, time.time() - s))

        panorama_generator.save_panoramas_to_video()

        panorama_generator.show_panorama(0)


if __name__ == '__main__':
    main()
```

4 ex4/sol4.py

```
# ----------- Imports -----------
import numpy as np
import os
import matplotlib.pyplot as plt

from scipy.ndimage.morphology import generate_binary_structure
from scipy.ndimage.filters import maximum_filter
from scipy.ndimage import label, center_of_mass, map_coordinates
from scipy.signal import convolve2d
import shutil
from imageio import imwrite, imread

import sol4_utils

# ----------- Constants -----------
MIN_IMG_SIZE_IN_EACH_AXIS = 16
EXPEND_FACTOR = 2


# ----------- Helper functions -----------

def non_maximum_suppression(image):
    """
    Finds local maximas of an image.
    :param image: A 2D array representing an image.
    :return: A boolean array with the same shape as the input image, where True indicates local maximum.
    """
    # Find local maximas.
    neighborhood = generate_binary_structure(2, 2)
    local_max = maximum_filter(image, footprint=neighborhood) == image
    local_max[image < (image.max() * 0.1)] = False

    # Erode areas to single points.
    lbs, num = label(local_max)
    centers = center_of_mass(local_max, lbs, np.arange(num) + 1)
    centers = np.stack(centers).round().astype(np.int)
    ret = np.zeros_like(image, dtype=np.bool)
    ret[centers[:, 0], centers[:, 1]] = True

    return ret


def spread_out_corners(im, m, n, radius):
    """
    Splits the image im to m by n rectangles and uses harris_corner_detector on each.
    :param im: A 2D array representing an image.
    :param m: Vertical number of rectangles.
    :param n: Horizontal number of rectangles.
    :param radius: Minimal distance of corner points from the boundary of the image.
    :return: An array with shape (N,2), where ret[i,:] are the [x,y] coordinates of the ith corner points.
    """
    corners = [np.empty((0, 2), dtype=np.int)]
    x_bound = np.linspace(0, im.shape[1], n + 1, dtype=np.int)
    y_bound = np.linspace(0, im.shape[0], m + 1, dtype=np.int)
    for i in range(n):
        for j in range(m):
            # Use Harris detector on every sub image.
            sub_im = im[y_bound[j]:y_bound[j + 1], x_bound[i]:x_bound[i + 1]]
            sub_corners = harris_corner_detector(sub_im)
```

```python
            sub_corners += np.array([x_bound[i], y_bound[j]])[np.newaxis, :]
            corners.append(sub_corners)
    corners = np.vstack(corners)
    legit = ((corners[:, 0] > radius) & (
                corners[:, 0] < im.shape[1] - radius) &
             (corners[:, 1] > radius) & (
                         corners[:, 1] < im.shape[0] - radius))
    ret = corners[legit, :]
    return ret


def warp_image(image, homography):
    """
    Warps an RGB image with a given homography.
    :param image: an RGB image.
    :param homography: homograhpy.
    :return: A warped image.
    """
    return np.dstack(
        [warp_channel(image[..., channel], homography) for channel in
         range(3)])


def filter_homographies_with_translation(homographies,
                                         minimum_right_translation):
    """
    Filters rigid transformations encoded as homographies by the amount of translation from left to right.
    :param homographies: homograhpies to filter.
    :param minimum_right_translation: amount of translation below which the transformation is discarded.
    :return: filtered homographies..
    """
    translation_over_thresh = [0]
    last = homographies[0][0, -1]
    for i in range(1, len(homographies)):
        if homographies[i][0, -1] - last > minimum_right_translation:
            translation_over_thresh.append(i)
            last = homographies[i][0, -1]
    return np.array(translation_over_thresh).astype(np.int)


def estimate_rigid_transform(points1, points2, translation_only=False):
    """
    Computes rigid transforming points1 towards points2, using least squares method.
    points1[i,:] corresponds to poins2[i,:]. In every point, the first coordinate is *x*.
    :param points1: array with shape (N,2). Holds coordinates of corresponding points from image 1.
    :param points2: array with shape (N,2). Holds coordinates of corresponding points from image 2.
    :param translation_only: whether to compute translation only. False (default) to compute rotation as well.
    :return: A 3x3 array with the computed homography.
    """
    centroid1 = points1.mean(axis=0)
    centroid2 = points2.mean(axis=0)

    if translation_only:
        rotation = np.eye(2)
        translation = centroid2 - centroid1

    else:
        centered_points1 = points1 - centroid1
        centered_points2 = points2 - centroid2

        sigma = centered_points2.T @ centered_points1
        U, _, Vt = np.linalg.svd(sigma)

        rotation = U @ Vt
        translation = -rotation @ centroid1 + centroid2

    H = np.eye(3)
    H[:2, :2] = rotation
```

```python
    H[:2, 2] = translation
    return H


class PanoramicVideoGenerator:
    """
    Generates panorama from a set of images.
    """

    def __init__(self, data_dir, file_prefix, num_images):
        """
        The naming convention for a sequence of images is file_prefixN.jpg,
        where N is a running number 001, 002, 003...
        :param data_dir: path to input images.
        :param file_prefix: see above.
        :param num_images: number of images to produce the panoramas with.
        """
        self.file_prefix = file_prefix
        self.files = [
            os.path.join(data_dir, '%s%03d.jpg' % (file_prefix, i + 1)) for i
            in range(num_images)]
        self.files = list(filter(os.path.exists, self.files))
        self.panoramas = None
        self.homographies = None
        print('found %d images' % len(self.files))

    def align_images(self, translation_only=False):
        """
        compute homographies between all images to a common coordinate system
        :param translation_only: see estimte_rigid_transform
        """
        # Extract feature point locations and descriptors.
        points_and_descriptors = []
        for file in self.files:
            image = sol4_utils.read_image(file, 1)
            self.h, self.w = image.shape
            pyramid, _ = sol4_utils.build_gaussian_pyramid(image, 3, 7)
            points_and_descriptors.append(find_features(pyramid))

        # Compute homographies between successive pairs of images.
        Hs = []
        for i in range(len(points_and_descriptors) - 1):
            points1, points2 = points_and_descriptors[i][0], \
                               points_and_descriptors[i + 1][0]
            desc1, desc2 = points_and_descriptors[i][1], \
                           points_and_descriptors[i + 1][1]

            # Find matching feature points.
            ind1, ind2 = match_features(desc1, desc2, .7)
            points1, points2 = points1[ind1, :], points2[ind2, :]

            # Compute homography using RANSAC.
            H12, inliers = ransac_homography(points1, points2, 100, 6,
                                             translation_only)

            # Uncomment for debugging: display inliers and outliers among matching points.
            # In the submitted code this function should be commented out!
            # display_matches(self.images[i], self.images[i+1], points1 , points2, inliers)

            Hs.append(H12)

        # Compute composite homographies from the central coordinate system.
        accumulated_homographies = accumulate_homographies(Hs,
                                                           (len(Hs) - 1) // 2)
        self.homographies = np.stack(accumulated_homographies)
        self.frames_for_panoramas = filter_homographies_with_translation(
            self.homographies, minimum_right_translation=5)
        self.homographies = self.homographies[self.frames_for_panoramas]
```

```python

    def generate_panoramic_images(self, number_of_panoramas):
        """
        combine slices from input images to panoramas.
        :param number_of_panoramas: how many different slices to take from each input image
        """
        assert self.homographies is not None

        # compute bounding boxes of all warped input images in the coordinate system of the middle image (as given by the hom
        self.bounding_boxes = np.zeros((self.frames_for_panoramas.size, 2, 2))
        for i in range(self.frames_for_panoramas.size):
            self.bounding_boxes[i] = compute_bounding_box(
                self.homographies[i], self.w, self.h)

        # change our reference coordinate system to the panoramas
        # all panoramas share the same coordinate system
        global_offset = np.min(self.bounding_boxes, axis=(0, 1))
        self.bounding_boxes -= global_offset

        slice_centers = np.linspace(0, self.w, number_of_panoramas + 2,
                                    endpoint=True, dtype=np.int)[1:-1]
        warped_slice_centers = np.zeros(
            (number_of_panoramas, self.frames_for_panoramas.size))
        # every slice is a different panorama, it indicates the slices of the input images from which the panorama
        # will be concatenated
        for i in range(slice_centers.size):
            slice_center_2d = np.array([slice_centers[i], self.h // 2])[None,
                              :]
            # homography warps the slice center to the coordinate system of the middle image
            warped_centers = [apply_homography(slice_center_2d, h) for h in
                              self.homographies]
            # we are actually only interested in the x coordinate of each slice center in the panoramas' coordinate system
            warped_slice_centers[i] = np.array(warped_centers)[:, :,
                                      0].squeeze() - global_offset[0]

        panorama_size = np.max(self.bounding_boxes, axis=(0, 1)).astype(
            np.int) + 1

        # boundary between input images in the panorama
        x_strip_boundary = ((warped_slice_centers[:,
                             :-1] + warped_slice_centers[:, 1:]) / 2)
        x_strip_boundary = np.hstack([np.zeros((number_of_panoramas, 1)),
                                      x_strip_boundary,
                                      np.ones((number_of_panoramas, 1)) *
                                      panorama_size[0]])
        x_strip_boundary = x_strip_boundary.round().astype(np.int)

        self.panoramas = np.zeros(
            (number_of_panoramas, panorama_size[1], panorama_size[0], 3),
            dtype=np.float64)
        for i, frame_index in enumerate(self.frames_for_panoramas):
            # warp every input image once, and populate all panoramas
            image = sol4_utils.read_image(self.files[frame_index], 2)
            warped_image = warp_image(image, self.homographies[i])
            x_offset, y_offset = self.bounding_boxes[i][0].astype(np.int)
            y_bottom = y_offset + warped_image.shape[0]

            for panorama_index in range(number_of_panoramas):
                # take strip of warped image and paste to current panorama
                boundaries = x_strip_boundary[panorama_index, i:i + 2]
                image_strip = warped_image[:,
                              boundaries[0] - x_offset: boundaries[
                                                            1] - x_offset]
                x_end = boundaries[0] + image_strip.shape[1]
                self.panoramas[panorama_index, y_offset:y_bottom,
                boundaries[0]:x_end] = image_strip

        # crop out areas not recorded from enough angles
```

```python
        # assert will fail if there is overlap in field of view between the left most image and the right most image
        # crop_left = int(self.bounding_boxes[0][1, 0])
        # crop_right = int(self.bounding_boxes[-1][0, 0])
        # assert crop_left < crop_right, 'for testing your code with a few images do not crop.'
        # print(crop_left, crop_right)
        # self.panoramas = self.panoramas[:, :, crop_left:crop_right, :]  # todo

    def save_panoramas_to_video(self):
        assert self.panoramas is not None
        out_folder = 'tmp_folder_for_panoramic_frames/%s' % \
                     self.file_prefix  # todo
        try:
            shutil.rmtree(out_folder)
        except:
            print('could not remove folder')
            pass
        os.makedirs(out_folder)
        # save individual panorama images to 'tmp_folder_for_panoramic_frames'
        for i, panorama in enumerate(self.panoramas):
            imwrite('%s/panorama%02d.png' % (out_folder, i + 1), panorama)
            # todo
        if os.path.exists('%s.mp4' % self.file_prefix):
            os.remove('%s.mp4' % self.file_prefix)
        # write output video to current folder
        os.system('ffmpeg -framerate 3 -i %s/panorama%%02d.png %s.mp4' %
                  (out_folder, self.file_prefix))  #todo \ and 9 to 3

    def show_panorama(self, panorama_index, figsize=(20, 20)):
        assert self.panoramas is not None
        plt.figure(figsize=figsize)
        plt.imshow(self.panoramas[panorama_index].clip(0, 1))
        plt.show()


# ----------- 3.1: Feature point detection and descriptor extraction  -----------


def harris_corner_detector(im):
    """
    Detects harris corners.
    Make sure the returned coordinates are x major!!!
    :param im: A 2D array representing an image.
    :return: An array with shape (N,2), where ret[i,:] are the [x,y] coordinates of the ith corner points.
    """
    x_der_vec = np.array([1, 0, -1])[np.newaxis, :]
    y_der_vec = x_der_vec.T
    I_x = convolve2d(im, x_der_vec, mode='same', boundary='symm')
    I_y = convolve2d(im, y_der_vec, mode='same', boundary='symm')
    I_xx = I_x * I_x
    I_yy = I_y * I_y
    I_xy = I_x * I_y
    blur_I_xx = sol4_utils.blur_spatial(I_xx, 3)
    blur_I_yy = sol4_utils.blur_spatial(I_yy, 3)
    blur_I_xy = sol4_utils.blur_spatial(I_xy, 3)
    det = blur_I_xx * blur_I_yy - blur_I_xy * blur_I_xy
    trace = blur_I_xx + blur_I_yy
    R = det - 0.04 * (trace ** 2)
    corners = non_maximum_suppression(R)
    cor_arr = np.where(corners > 0)
    points = np.dstack((cor_arr[1], cor_arr[0]))[0]

    return points


def sample_descriptor(im, pos, desc_rad):
    """
    Samples descriptors at the given corners.
    :param im: A 2D array representing an image.
```

```
    :param pos: An array with shape (N,2), where pos[i,:] are the [x,y] coordinates of the ith corner point.
    :param desc_rad: "Radius" of descriptors to compute.
    :return: A 3D array with shape (N,K,K) containing the ith descriptor at desc[i,:,:].
    """
    desc_size = desc_rad * 2 + 1
    desc_array = np.zeros((len(pos), desc_size, desc_size))
    for i in range(pos.shape[0]):
        p_x = np.tile(np.linspace(pos[i][0] - desc_rad, pos[i][0] +
                                  desc_rad, desc_size), desc_size).reshape(
            desc_size, desc_size)
        p_y = np.repeat(np.linspace(pos[i][1] - desc_rad, pos[i][1] +
                                    desc_rad, desc_size), desc_size).reshape(
            desc_size, desc_size)

        pos_from_map = map_coordinates(im, [p_y, p_x], order=1, prefilter='false')
        desc_avg = np.average(pos_from_map)
        if np.linalg.norm(pos_from_map - desc_avg) == 0:
            desc_array[i, :, :] = pos_from_map
        else:
            normalized_desc = (pos_from_map - desc_avg) / (np.linalg.norm(pos_from_map -
                                                                          desc_avg))
            desc_array[i, :, :] = normalized_desc

    return np.array(desc_array)


def find_features(pyr):
    """
    Detects and extracts feature points from a pyramid.
    :param pyr: Gaussian pyramid of a grayscale image having 3 levels.
    :return: A list containing:
                1) An array with shape (N,2) of [x,y] feature location per row found in the image.
                   These coordinates are provided at the pyramid level pyr[0].
                2) A feature descriptor array with shape (N,K,K)
    """
    pos = spread_out_corners(pyr[0], 7, 7, 12)
    desc_array = sample_descriptor(pyr[2], pos * 0.25, 3)
    return [pos, desc_array]


# ----------- 3.2: Matching descriptors -----------


def match_features(desc1, desc2, min_score):
    """
    Return indices of matching descriptors.
    :param desc1: A feature descriptor array with shape (N1,K,K).
    :param desc2: A feature descriptor array with shape (N2,K,K).
    :param min_score: Minimal match score.
    :return: A list containing:
                1) An array with shape (M,) and dtype int of matching indices in desc1.
                2) An array with shape (M,) and dtype int of matching indices in desc2.
    """
    desc1_flattered = desc1.reshape(desc1.shape[0],
                                    desc1.shape[1] * desc1.shape[2])
    desc2_flattered = desc2.reshape(desc2.shape[0],
                                    desc2.shape[1] * desc2.shape[2]).T
    desc_score = desc1_flattered @ desc2_flattered
    desc_score_above_min_score = desc_score >= min_score
    second_biggest_score_in_row_desc_score = np.sort(desc_score, axis=1)[:,
                                             -2]
    second_biggest_score_in_col_desc_score = np.sort(desc_score, axis=0)[-2,
                                             :]
    big_enough_score_in_row_desc_score = desc_score.T >= \
                                         second_biggest_score_in_row_desc_score
    big_enough_score_in_col_desc_score = desc_score >= \
                                         second_biggest_score_in_col_desc_score
    good_points = np.argwhere(big_enough_score_in_row_desc_score.T *
```

```python
                                big_enough_score_in_col_desc_score *
                                desc_score_above_min_score)
    return [good_points[:, 0], good_points[:, 1]]


# ----------- 3.3: Registering the transformation -----------

def apply_homography(pos1, H12):
    """
    Apply homography to inhomogenous points.
    :param pos1: An array with shape (N,2) of [x,y] point coordinates.
    :param H12: A 3x3 homography matrix.
    :return: An array with the same shape as pos1 with [x,y] point coordinates obtained from transforming pos1 using H12.
    """
    pos1_tilda = np.insert(pos1, 2, [1], axis=1)
    pos2_tilda = H12 @ pos1_tilda.T
    x2_tilda = pos2_tilda[0]
    y2_tilda = pos2_tilda[1]
    z2_tilda = pos2_tilda[2]
    pos2 = np.dstack((x2_tilda / z2_tilda, y2_tilda / z2_tilda))[0]
    return np.array(pos2)


def ransac_homography(points1, points2, num_iter, inlier_tol,
                      translation_only=False):
    """
    Computes homography between two sets of points using RANSAC.
    :param points1: An array with shape (N,2) containing N rows of [x,y] coordinates of matched points in image 1.
    :param points2: An array with shape (N,2) containing N rows of [x,y] coordinates of matched points in image 2.
    :param num_iter: Number of RANSAC iterations to perform.
    :param inlier_tol: inlier tolerance threshold.
    :param translation_only: see estimate rigid transform
    :return: A list containing:
                1) A 3x3 normalized homography matrix.
                2) An Array with shape (S,) where S is the number of inliers,
                    containing the indices in pos1/pos2 of the maximal set of inlier matches found.
    """
    inlier = list()
    max_inlier = 0
    for i in range(num_iter):
        random_idx = np.random.randint(0, high=points1.shape[0], size=2)
        P1_J = points1[random_idx]
        P2_J = points2[random_idx]
        H12 = estimate_rigid_transform(P1_J, P2_J, translation_only)
        P2_J_transformed = apply_homography(points1, H12)
        E = np.power((np.linalg.norm(P2_J_transformed - points2, axis=1)), 2)
        inlier_idx = E < inlier_tol
        inlier_num = np.sum(inlier_idx)
        if inlier_num > max_inlier:
            max_inlier = np.sum(inlier_num)
            inlier = inlier_idx
    best_H12 = estimate_rigid_transform(points1[inlier], points2[inlier],
                                        translation_only)
    best_inlier = np.argwhere(inlier)
    return [best_H12, best_inlier[:, 0]]


def display_matches(im1, im2, points1, points2, inliers):
    """
    Dispalay matching points.
    :param im1: A grayscale image.
    :param im2: A grayscale image.
    :parma pos1: An aray shape (N,2), containing N rows of [x,y] coordinates of matched points in im1.
    :param pos2: An aray shape (N,2), containing N rows of [x,y] coordinates of matched points in im2.
    :param inliers: An array with shape (S,) of inlier matches.
    """
    im = np.hstack((im1, im2))
    plt.figure()
```

```python
    plt.imshow(im, cmap='gray')
    plt.plot([points1[inliers, 0], points2[inliers, 0] + im1.shape[1]],
             [points1[inliers, 1], points2[inliers, 1]], c='y', lw='0.5')
    plt.plot(points1[:, 0], points1[:, 1], '.', c='r', ms='1')
    plt.plot(points2[:, 0] + im1.shape[1], points2[:, 1], '.', c='r', ms='1')
    outliers = points1
    outliers[inliers] = 0
    outliers = np.argwhere(outliers)[:, 0]
    plt.plot([points1[outliers, 0], points2[outliers, 0] + im1.shape[1]],
             [points1[outliers, 1], points2[outliers, 1]], c='b',
             lw='0.08')

    plt.show()


# ----------- 3.4: Transforming to a common coordinate system -----------


def accumulate_homographies(H_succesive, m):
    """
    Convert a list of succesive homographies to a
    list of homographies to a common reference frame.
    :param H_successive: A list of M-1 3x3 homography
      matrices where H_successive[i] is a homography which transforms points
      from coordinate system i to coordinate system i+1.
    :param m: Index of the coordinate system towards which we would like to
      accumulate the given homographies.
    :return: A list of M 3x3 homography matrices,
      where H2m[i] transforms points from coordinate system i to coordinate system m
    """

    new_H = np.zeros((len(H_succesive) + 1, 3,3))

    new_H[m, :, :] = np.eye(3)
    for i in range(m, len(H_succesive)):
        inv = np.linalg.inv(H_succesive[i])
        new_H[i + 1] = np.dot(new_H[i], inv)
        new_H[i + 1] = new_H[i + 1]/new_H[i + 1][2,2]
    for i in range(m):
        new_H[m - i - 1] = np.dot(new_H[m - i], H_succesive[m - i - 1])
        new_H[m - i - 1] = new_H[m - i - 1] / new_H[m - i - 1][2, 2]

    return list(new_H)


def compute_bounding_box(homography, w, h):
    """
    computes bounding box of warped image under homography, without actually warping the image
    :param homography: homography
    :param w: width of the image
    :param h: height of the image
    :return: 2x2 array, where the first row is [x,y] of the top left corner,
     and the second row is the [x,y] of the bottom right corner
    """
    top_left = apply_homography(np.array([[0, 0]]), homography)
    top_right = apply_homography(np.array([[w, 0]]), homography)
    bottom_left = apply_homography(np.array([[0, h]]), homography)
    bottom_right = apply_homography(np.array([[h, h]]), homography)
    max_x = max(top_left[0, 0],
                top_right[0, 0],
                bottom_left[0, 0],
                bottom_right[0, 0])
    min_x = min(top_left[0, 0],
                top_right[0, 0],
                bottom_left[0, 0],
                bottom_right[0, 0])
    max_y = max(top_left[0, 1],
                top_right[0, 1],
```

```
                bottom_left[0, 1],
                bottom_right[0, 1])
    min_y = min(top_left[0, 1],
                top_right[0, 1],
                bottom_left[0, 1],
                bottom_right[0, 1])
    bounding_box = np.array([[min_x, min_y],
                             [max_x, max_y]], dtype=int)
    return bounding_box


# ----------- 4: Stitching -----------


def warp_channel(image, homography):
    """
    Warps a 2D image with a given homography.
    :param image: a 2D image.
    :param homography: homograhpy.
    :return: A 2d warped image.
    """
    [[min_x, min_y], [max_x, max_y]] = compute_bounding_box(homography,
                                                            image.shape[1],
                                                            image.shape[0])
    x = np.arange(min_x, max_x)
    y = np.arange(min_y, max_y)
    x_mesh, y_mesh = np.meshgrid(x, y, indexing='xy')
    back_warp = apply_homography(
        np.vstack((x_mesh.flatten(), y_mesh.flatten())).T,
        np.linalg.inv(homography))
    warp = map_coordinates(image,
                           [back_warp[:, 1], back_warp[:, 0]],
                           order=1,
                           prefilter=False)
    return warp.reshape((max_y - min_y, max_x - min_x))
```

5 ex4/sol4 utils.py

```python
from scipy.signal import convolve2d
import numpy as np
from imageio import imwrite, imread
from skimage.color import rgb2gray
from scipy.ndimage.filters import convolve as _convolve


# ----------- Constants -----------
MIN_IMG_SIZE_IN_EACH_AXIS = 16
EXPEND_FACTOR = 2


def _im_downsample(im, blur_filter):
    """
    downsamples the image according to the blur filter given and takes
    every even pixel in the image.
    :param im: np.array image
    :param blur_filter: np.array of size (1,) of the filter
    :return: smaller sample.
    """
    im = _convolve(im, blur_filter, mode='reflect')
    im = _convolve(im, blur_filter.T, mode='reflect')
    im = im[::2, ::2]
    return im


def _im_expand(im, blur_filter):
    """
    expands the image accodring to the blur filter. Makes the image twice
    as bigger.
    :param im: np.array image
    :param blur_filter: np.array of size (1,) of the filter
    :return: bigger image.
    """
    expended_im = np.zeros(
        (im.shape[0] * EXPEND_FACTOR, im.shape[1] * EXPEND_FACTOR))
    expended_im[1::2, 1::2] = im
    blur_filter = blur_filter * EXPEND_FACTOR
    expended_im = _convolve(_convolve(expended_im, blur_filter),
                            blur_filter.T)
    return expended_im

def gaussian_kernel(kernel_size):
    conv_kernel = np.array([1, 1], dtype=np.float64)[:, None]
    conv_kernel = convolve2d(conv_kernel, conv_kernel.T)
    kernel = np.array([1], dtype=np.float64)[:, None]
    for i in range(kernel_size - 1):
        kernel = convolve2d(kernel, conv_kernel, 'full')
    return kernel / kernel.sum()


def blur_spatial(img, kernel_size):
    kernel = gaussian_kernel(kernel_size)
    blur_img = np.zeros_like(img)
    if len(img.shape) == 2:
        blur_img = convolve2d(img, kernel, 'same', 'symm')
    else:
        for i in range(3):
            blur_img[..., i] = convolve2d(img[..., i], kernel, 'same', 'symm')
```

```
    return blur_img

def read_image(filename, representation):
    """
    function which reads an image ﬁle and converts it into a given
    representation.
    :param filename: the filename of an image on disk (could be grayscale or
    RGB).
    :param representation: representation code, either 1 or 2 defining
    whether the output should be a grayscaleimage (1) or an RGB image (2).
    If the input image is grayscale, we won't call it with representation = 2.
    :return: rgb or grayscale img
    """
    image = imread(filename)
    new_image = image.astype(np.float64)
    new_image /= 255
    if representation == 1:
        new_image = rgb2gray(new_image)
    return new_image

def build_gaussian_pyramid(im, max_levels, filter_size):
    """
    constructs a gaussian pyramid.
    :param im: grayscale image with double values in [0,1].
    :param max_levels: maximal number of levels in the resulting pyramid.
    :param filter_size: the size of the gaussian filter
    :return: pyr, filter_vec. pyr - python array where the elements are
    images with different sizes. filter_vec - the gaussian filter used.
    """
    blur_filter_not_normalized = _get_binomial_coefficients(filter_size)
    blur_filter = blur_filter_not_normalized / np.sum(
      blur_filter_not_normalized)

    pyr = list()
    curr_im = im
    while ((curr_im.shape[0] >= MIN_IMG_SIZE_IN_EACH_AXIS) and
           (curr_im.shape[1] >= MIN_IMG_SIZE_IN_EACH_AXIS) and
           (len(pyr) < max_levels)):
      pyr.append(curr_im)
      curr_im = _im_downsample(curr_im, blur_filter)

    return pyr, blur_filter


def _get_binomial_coefficients(size):
  """
  creates gaussian vector of the given size. size must be odd.
  :param size: odd integer
  :return: gaussian vector of the given size
  """
  binomial_coefficients = np.array([1, 1], dtype=np.float64)
  for _ in range(size - 2):
    binomial_coefficients = np.convolve([1, 1], binomial_coefficients)
  binomial_coefficients = binomial_coefficients[np.newaxis, :]
  return binomial_coefficients
```