Contents

1 Basic Test Results

```
Archive:  ex4_additional_files.zip
  inflating: current/sol4_add.py
  inflating: current/example_panoramas.py
  inflating: current/presubmit_externals/oxford1.jpg
  inflating: current/presubmit_externals/oxford2.jpg
Archive:  /tmp/bodek.0In70k/impr/ex4/mottig/presubmission/submission
  inflating: current/README
   creating: current/external/
  inflating: current/external/lib1.jpg
  inflating: current/external/lib2.jpg
  inflating: current/external/res_pan.jpg
  inflating: current/my_panorama.py
  inflating: current/sol4.py
  inflating: current/sol4_utils.py
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 ===

login:  mottig

submitted files:
sol4.py
sol4_utils.py
my_panorama.py
external/lib1.jpg
external/lib2.jpg
external/res_pan.jpg

=== Bonus submittes? ===
no

=== 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 4.1 ===

Accumulate homographies
    Passed!

=== Section 4.3 ===

Render grayscale panorama, actual panorama should be RGB
    Passed!

=== Testing runtime ===

    Passed!
Your runtime was: 19, expected runtime should be no more than 30 seconds!

=== All tests have passed ===
=== Pre-submission script done ===


    Please go over the output and verify that there are 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 README

```
mottig
sol4.py
sol4_utils.py
my_panorama.py
external/lib1.jpg
external/lib2.jpg
external/res_pan.jpg
```

3 my panorama.py

```
import matplotlib.pyplot as plt
import numpy as np
import os

import sol4
import sol4_utils

def generate_panorama(data_dir, file_prefix, num_images, figsize=(20,20)):
  # The naming convention for a sequence of images is nameN.jpg, where N is a running number 1,2,..
  files = [os.path.join(data_dir,'%s%d.jpg'%(file_prefix, i+1)) for i in range(num_images)]

  # Read images.
  ims = [sol4_utils.read_image(f,1) for f in files]
  # Extract feature point locations and descriptors.
  def im_to_points(im):
    pyr,_ = sol4_utils.build_gaussian_pyramid(im, 3, 7)
    return sol4.find_features(pyr)
  p_d = [im_to_points(im) for im in ims]

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

    # Find matching feature points.
    ind1, ind2 = sol4.match_features(desc1, desc2, .7)
    points1, points2 = points1[ind1,:], points2[ind2,:]

    # Compute homography using RANSAC.
    H12, inliers = sol4.ransac_homography(points1, points2, 10000, 6)

    # Display inlier and outlier matches.
    # sol4.display_matches(ims[i], ims[i+1], points1 , points2, inliers=inliers)
    Hs.append(H12)

  # Compute composite homographies from the panorama coordinate system.
  Htot = sol4.accumulate_homographies(Hs, (num_images-1)//2)

  # Final panorama is generated using 3 channels of the RGB images
  ims_rgb = [sol4_utils.read_image(f,2) for f in files]

  # Render panorama for each color channel and combine them.
  panorama = [sol4.render_panorama([im[...,i] for im in ims_rgb], Htot) for i in range(3)]
  panorama = np.dstack(panorama)

  #plot the panorama
  plt.figure(figsize=figsize)
  plt.imshow(panorama.clip(0,1))
  plt.imsave(os.path.join('external/', 'lib_pan.jpg'), panorama.clip(0,1))
  plt.show()

def main():
  generate_panorama('external/', 'lib', 2)
if __name__ == '__main__':
  main()
```

4 sol4.py

```python
import sol4_utils
from sol4_utils import np
import sol4_add
from itertools import accumulate
import matplotlib.pyplot as plt
from scipy.ndimage.interpolation import map_coordinates

DER_FILTER = np.array([1, 0, -1], np.float32).reshape(1, 3)
BLUR_SIZE = 3
K = 0.04
N = M = 4  # defaults for spread_out_corners n and m
RADIUS = 3
DEFAULT_DESC_RAD = 3
DEFAULT_MIN_SCORE = 0.5
NUM_OF_POINTS_TO_TRANS = 4
EPSILON = 10 ** -5
OVERLAP = 30


def harris_corner_detector(im: np.ndarray) -> np.ndarray:
    """
    extract harris-corner key feature points
    :param im: the image to extract key points
    :return: An array with shape (N,2) of [x,y] key points locations in im.
    """
    ix = sol4_utils.sp_signal.convolve2d(im, DER_FILTER, 'same', 'wrap')
    iy = sol4_utils.sp_signal.convolve2d(im, DER_FILTER.T, 'same', 'wrap')
    ix_2 = sol4_utils.blur_spatial(ix**2, BLUR_SIZE)
    iy_2 = sol4_utils.blur_spatial(iy**2, BLUR_SIZE)
    ix_iy = sol4_utils.blur_spatial(ix * iy, BLUR_SIZE)
    r = ix_2 * iy_2 - ix_iy**2 - K*(ix_2 + iy_2)**2  # R - the response of the
    max_of_r = sol4_add.non_maximum_suppression(r)
    pos = np.transpose(np.nonzero(max_of_r))  # array of the indices of non-zero pixels in max_of_r
    pos_for_spread = np.transpose(np.array([pos[:, 1], pos[:, 0]]))  # school function works with [yx]
    return pos_for_spread


def sample_descriptor(im: np.ndarray, pos: np.ndarray, desc_rad: int) -> np.ndarray:
    """
    sample a given image with simplified version of the MOPS descriptor
    :param im: grayscale image to sample within.
    :param pos: An array with shape (N,2) of [x,y] positions to sample descriptors in im
    :param desc_rad: "Radius" of descriptors to compute
    :return: A 3D array with shape (K,K,N) containing the ith descriptor at desc[:,:,i]
    """
    k = 1+2*desc_rad
    n = pos.shape[0]
    desc = np.empty((k, k, n), np.float32)
    for i in range(n):
        grid = np.meshgrid(np.arange(pos[i, 1] - desc_rad, pos[i, 1] + desc_rad + 1),
                           np.arange(pos[i, 0] - desc_rad, pos[i, 0] + desc_rad + 1))
        desc_win = map_coordinates(im, grid, order=1, prefilter=False)
        desc_win -= np.mean(desc_win)
        win_std = np.linalg.norm(desc_win)
        if win_std:
            desc_win /= win_std
            desc[:, :, i] = desc_win
    return desc
```

```python

def find_features(pyr: list) -> tuple:
    """
    get simplified version of the MOPS descriptors from the given pyramid and the positions of them
    :param pyr: Gaussian pyramid of a grayscale image having 3 levels
    :return: the positions of the descriptors and array of them [with shape (1+2*desc_rad,1+2*desc_rad,N)]
    """
    pos = sol4_add.spread_out_corners(pyr[0], N, M, RADIUS)
    desc = sample_descriptor(pyr[2], pos/4, DEFAULT_DESC_RAD)
    return pos, desc


def get_binary_mat(idx_of_max: np.ndarray, shape: tuple) -> np.ndarray:
    """
    get binary matrix with once where the indices of descriptors are with max value
    :param idx_of_max: matrix with the indices of max values
    :param shape: the shape of the desire matrix
    :return: a binary matrix with once where the indices of descriptors are with max value
    """
    binary_mat = np.zeros(shape)
    for i in range(idx_of_max.shape[0]):
        binary_mat[i, idx_of_max[i, :]] = 1
    return binary_mat


def match_features(desc1: np.ndarray, desc2: np.ndarray, min_score: float) -> tuple:
    """
    get the indices of matching descriptors between tow arrays of descriptors (desc1 and desc2)
    :param desc1: A feature descriptor array with shape(1+2*desc_rad,1+2*desc_rad,N1).
    :param desc2: A feature descriptor array with shape(1+2*desc_rad,1+2*desc_rad,N2).
    :param min_score: min score between two descriptors required to be regarded as corresponding points.
    :return: 2 arrays with shape (M,) and dtype int, of matching indices in the given descs (each for one)
    """
    # reshape so every column is a single flatten descriptor
    desc1 = desc1.reshape(desc1.shape[0]**2, desc1.shape[2])
    desc2 = desc2.reshape(desc2.shape[0]**2, desc2.shape[2])
    sjk = desc1.T.dot(desc2)
    sjk[sjk < min_score] = 0  # apply nin_score condition

    idx_of_desc1_max = np.argpartition(sjk, -2, 1)[:, -2:]
    idx_of_desc2_max = np.argpartition(sjk, -2, 0)[-2:, :].T

    # get binary matrices represent the indices of max descriptors
    desc1_ind = get_binary_mat(idx_of_desc1_max, (desc1.shape[1], desc2.shape[1]))
    desc2_ind = get_binary_mat(idx_of_desc2_max, (desc2.shape[1], desc1.shape[1]))

    # only if both desc1 and desc2 agree on descriptor, it will remain 1
    match_ind1, match_ind2 = np.where(desc1_ind * desc2_ind.T == 1)
    return match_ind1, match_ind2


def apply_homography(pos1: np.ndarray, H12: np.ndarray) -> np.ndarray:
    """
    apply an homography transformation on a set of 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 in image i+1
    obtained from transforming pos1 using H12.
    """
    pos1 = np.hstack((pos1, np.ones((pos1.shape[0], 1))))  # add homographic element
    trans = pos1.dot(H12.T).astype(np.float32)  # its is more efficient to transpose H12 and not pos1
    trans /= trans[:, 2].reshape(trans.shape[0], 1)
    pos2 = trans[:, [0, 1]]   # normalize back to x,y
    return pos2


def ransac_homography(pos1: np.ndarray, pos2: np.ndarray, num_iters: int, inlier_tol: np.float32) -> tuple:
    """
```

```python
    apply RANSAC homography fitting
    :param pos1: an array with n rows of [x,y] coordinates of matched points of first image.
    :param pos2: an array with n rows of [x,y] coordinates of matched points of second image.
    :param num_iters: number of RANSAC iterations to perform.
    :param inlier_tol: inlier tolerance threshold.
    :return: A 3x3 normalized homography matrix and 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.
    """
    inliers = np.array([])
    for i in range(num_iters):
        rand_idx = np.random.choice(pos1.shape[0], size=NUM_OF_POINTS_TO_TRANS)  # choose 4 points
        pos1_smpl, pos2_smpl = pos1[rand_idx, :], pos2[rand_idx, :]
        h = sol4_add.least_squares_homography(pos1_smpl, pos2_smpl)
        if h is None:
            continue
        pos1_trans = apply_homography(pos1, h)
        e = np.linalg.norm(pos1_trans - pos2, axis=1)**2
        curr_inliers = np.where(e < inlier_tol)[0]  # indices of "good" points of pos2
        if len(curr_inliers) > len(inliers):
            inliers = curr_inliers

    H12 = sol4_add.least_squares_homography(pos1[inliers, :], pos2[inliers, :])
    return H12, inliers


def display_matches(im1, im2, pos1, pos2, inliers) -> None:
    """
    visualize the full set of point matches and the inlier matches detected by RANSAC
    :param im1: first grayscale image
    :param im2: second grayscale images
    :param pos1: an array with n rows of [x,y] coordinates of matched points of first image.
    :param pos2: an array with n rows of [x,y] coordinates of matched points of second image.
    :param inliers: An array with shape (S,) of inlier matches (e.g. see output of ransac homography)
    """
    im = np.hstack((im1, im2))
    pos2[:, 0] += im1.shape[1]
    plt.figure()
    plt.imshow(im, cmap=plt.cm.gray)
    for i in range(len(pos1)):
        color = 'y' if i in inliers else 'b'
        plt.plot([pos1[i, 0], pos2[i, 0]], [pos1[i, 1], pos2[i, 1]], mfc='r', c=color, lw=1, ms=5, marker='.')
    plt.show()


def accumulate_homographies(H_successive: list, m: int) -> list:
    """
    get Hi,m from {Hi,i+1 : i = 0..M-1}.
    :param H_successive: A list of 3x3 homography matrices where H successive[i] is a homography
    that transforms points from coordinate system i to coordinate system i+1.
    :param m: Index of the coordinate system we would like to accumulate the given homographies towards.
    :return: A list of M 3x3 homography matrices, where H2m[i] transforms points from coordinate system i
    to coordinate system m.
    """
    less_than_m = H_successive[:m]  # all matrices for i<m
    less_than_m = list(accumulate(less_than_m[::-1], np.dot))[::-1]  # reverse again to 0-m
    less_than_m.append(np.eye(3))  # add for i=m
    bigger_than_m = H_successive[m:]  # all matrices for i>m
    bigger_than_m = list(map(np.linalg.inv, bigger_than_m))
    bigger_than_m = list(accumulate(bigger_than_m, np.dot))
    H2m = np.array(less_than_m + bigger_than_m)
    H2m = H2m.T / H2m[:, 2, 2]
    return list(H2m.T)


def next_power(d: int):
    """
    :param d: size of dimension of image
    :return: the next power of 2 of this size
```

```python
    """
    res = 1
    while res < d: res <<= 1
    return res


def get_centers_and_corners(ims: list, Hs: list) -> tuple:
    """
    get the corners and centers of each im in ims
    :param ims: list of grayscale images.
    :param Hs: list of 3x3 homography matrices.
    :return: tuple containing a list of x,y centers  and  a list with x_min,x_max,y_min and y_max
    """
    centers = []
    corners = np.empty((len(ims), 4))
    for i in range(len(ims)):
        rows_cor, cols_cor = ims[i].shape[0] - 1, ims[i].shape[1] - 1

        # get center of im[i]:
        curr_center = np.array([cols_cor/2, rows_cor/2]).reshape(1, 2)
        centers.append(apply_homography(curr_center, Hs[i])[0])

        # get corners of im[i]:
        curr_cornrs = np.array([[0, 0], [cols_cor, 0], [0, rows_cor], [cols_cor, rows_cor]]).reshape(4, 2)
        curr_cornrs = apply_homography(curr_cornrs, Hs[i])
        corners[i, 0] = np.min(curr_cornrs[:, 0])  # curr_x_min
        corners[i, 1] = np.max(curr_cornrs[:, 0])  # curr_x_max
        corners[i, 2] = np.min(curr_cornrs[:, 1])  # curr_y_min
        corners[i, 3] = np.max(curr_cornrs[:, 1])  # curr_y_max

    # calc canvas corners
    x_min, x_max = int(np.min(corners[:, 0])), int(np.max(corners[:, 1]))
    y_min, y_max = int(np.min(corners[:, 2])), int(np.max(corners[:, 3]))
    corners = [x_min, x_max, y_min, y_max]
    return centers, corners


def render_panorama(ims: list, Hs: list) -> np.ndarray:
    """
    panorama rendering
    :param ims: list of grayscale images.
    :param Hs: list of 3x3 homography matrices. Hs[i] is a homography that transforms points from the
    coordinate system of ims [i] to the coordinate system of the panorama.
    :return: A grayscale panorama image composed of vertical strips, backwarped using homographies from Hs,
    one from every image in ims.
    """
    if len(ims) == 1:
        return ims[0]

    # get data of the shape of the panorama
    centers, corners = get_centers_and_corners(ims, Hs)  # corners = [x_min, x_max, y_min, y_max]
    x_min, x_max, y_min, y_max = corners[0], corners[1], corners[2], corners[3]
    width, height = x_max - x_min + 1, y_max - y_min + 1

    # calc a fake shape so it could fit the pyramid blending - only power of 2 sizes
    next_power_of_cols = next_power(width)
    next_power_of_rows = next_power(height)
    cols_pad = next_power_of_cols - width
    rows_pad = next_power_of_rows - height

    # create the canvas of the panorama
    x_pano, y_pano = np.meshgrid(np.arange(x_min, x_max + cols_pad + 1),
                                 np.arange(y_min, y_max + rows_pad + 1))
    panorama = np.zeros(x_pano.shape)  # the canvas of the panorama
    pan_rows, pan_cols = panorama.shape

    # create borders of strips
    borders = [int(np.round((centers[i][0] + centers[i+1][0])/2) - x_min) for i in range(len(ims)-1)]
```

```
    borders.insert(0, 0)
    borders.append(x_pano.shape[1])

    # apply panorama
    for i in range(len(ims)):
        left = borders[i] - OVERLAP if i != 0 else borders[i]
        right = borders[i+1] + OVERLAP if i != len(ims)-1 else borders[i+1]
        x_coord, y_coord = x_pano[:, left:right], y_pano[:, left:right]  # indices of the current part
        xi_yi = np.array([x_coord.flatten(), y_coord.flatten()]).T
        xi_yi = apply_homography(xi_yi, np.linalg.inv(Hs[i]))

        curr_im = map_coordinates(ims[i], [xi_yi[:, 1], xi_yi[:, 0]], order=1, prefilter=False)
        curr_im = curr_im.reshape(panorama[:, left:right].shape)

        # apply blending on panorama:
        if i == 0:
            panorama[:, left:right] = curr_im
            continue
        temp_canvas = np.zeros(panorama.shape)
        temp_canvas[:, left:right] = curr_im

        # create a mask and blend them:
        mask = np.ones(panorama.shape)
        mask[:, borders[i]:] = 0
        panorama = sol4_utils.pyramid_blending(panorama, temp_canvas, mask, 4, 15, 15)
        panorama = panorama[:pan_rows, :pan_cols]

    panorama = panorama[:height, :width].astype(np.float32)  # back to real shape

    return panorama
```

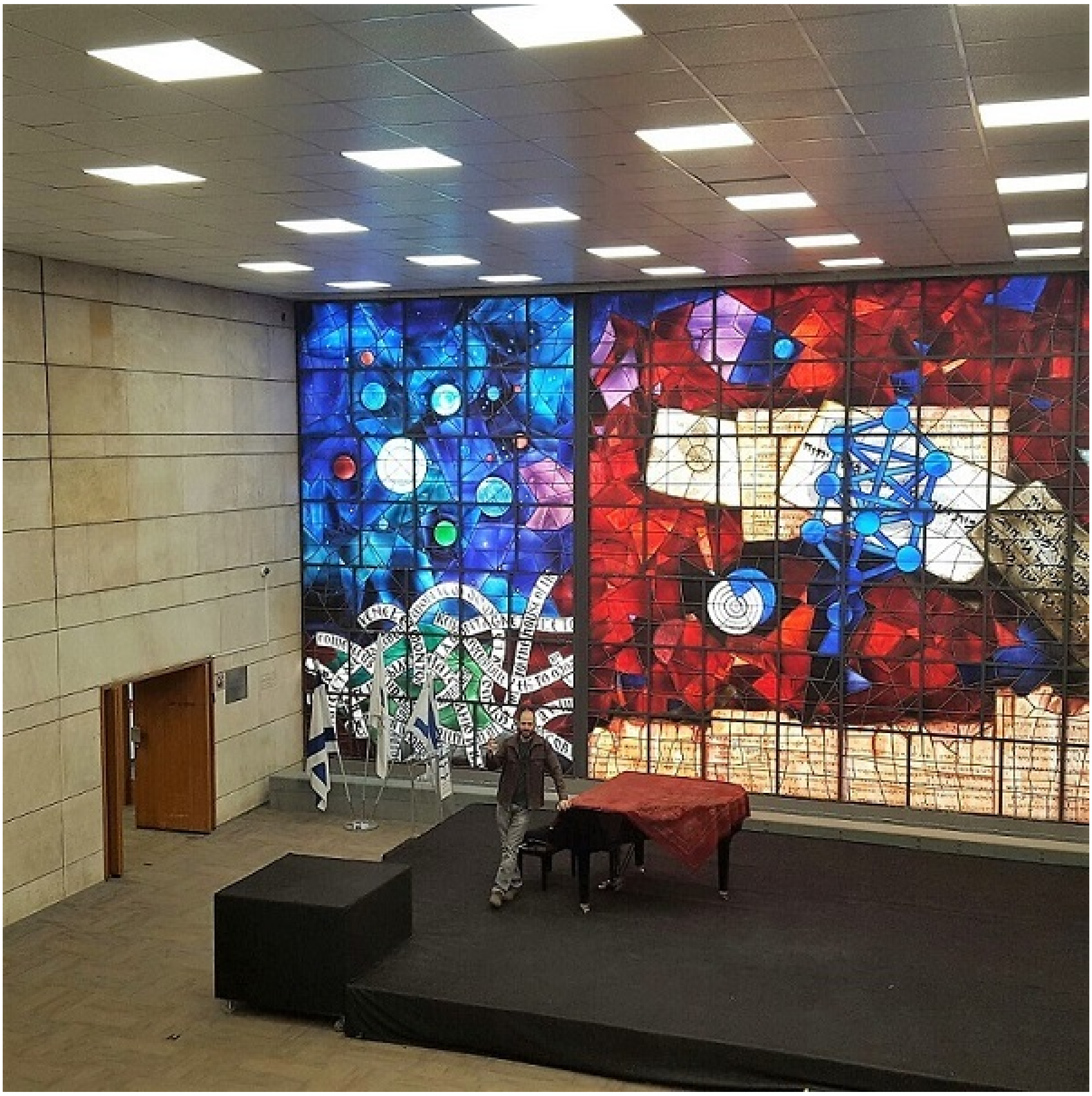
5 sol4 utils.py

```python
import numpy as np
from scipy.misc import imread
from skimage.color import rgb2gray
from scipy import signal as sp_signal
from scipy.ndimage import filters

GREYSCALE, COLOR, RGBDIM = 1, 2, 3
MAX_PIX_VAL = 255
PYR_IDX = 0  # the index of pyr in the tuple returned by build_gaussian_pyramid function
MIN_SIZE = 16  # minimum size of an image
GAUSSIAN_BASE = np.ones((1, 2), np.float32)  # gaussian kernel base - [1 1] vector
SAMPLE_FACTOR = 2  # determine down/up sampling frequency - e.g. when reducing image take one of each 2 pixels


def is_valid_args(filename: str, representation: int) -> bool:
    """
    Basic checks on the functions input
    """
    return (filename is not None) and \
           (representation == 1 or representation == 2) and isinstance(filename, str)


def read_image(filename: str, representation: int) -> np.ndarray:
    """
    Reads a given image file and converts it into a given representation
    :param filename: string containing the image filename to read.
    :param representation: representation code, either 1 or 2 defining if the output should be either a
    greyscale image (1) or an RGB image (2)
    :return: Image represented by a matrix of class np.float32, normalized to the range [0, 1].
    """

    if not is_valid_args(filename, representation):
        raise Exception("Please provide valid filename and representation code")

    try:
        im = imread(filename)
    except OSError:
        raise Exception("Filename should be valid image filename")

    if im.ndim == RGBDIM and (representation == GREYSCALE):  # change rgb to greyscale
        return rgb2gray(im).astype(np.float32)

    elif im.ndim != RGBDIM and (representation == COLOR):
        raise Exception("Converting greyscale to RGB is not supported")

    return im.astype(np.float32) / MAX_PIX_VAL


def gaussian_kernel_2d(size: int) -> np.ndarray:
    """
    create a gaussian kernel
    :param size: the size of the gaussian kernel in each dimension (an odd integer)
    :return: gaussian kernel as np.ndarray contains np.float32
    """
    if not size % 2: # if size is even number
        raise Exception("kernel size must be odd number")

    kernel = GAUSSIAN_BASE
    for i in range(size-2):
```

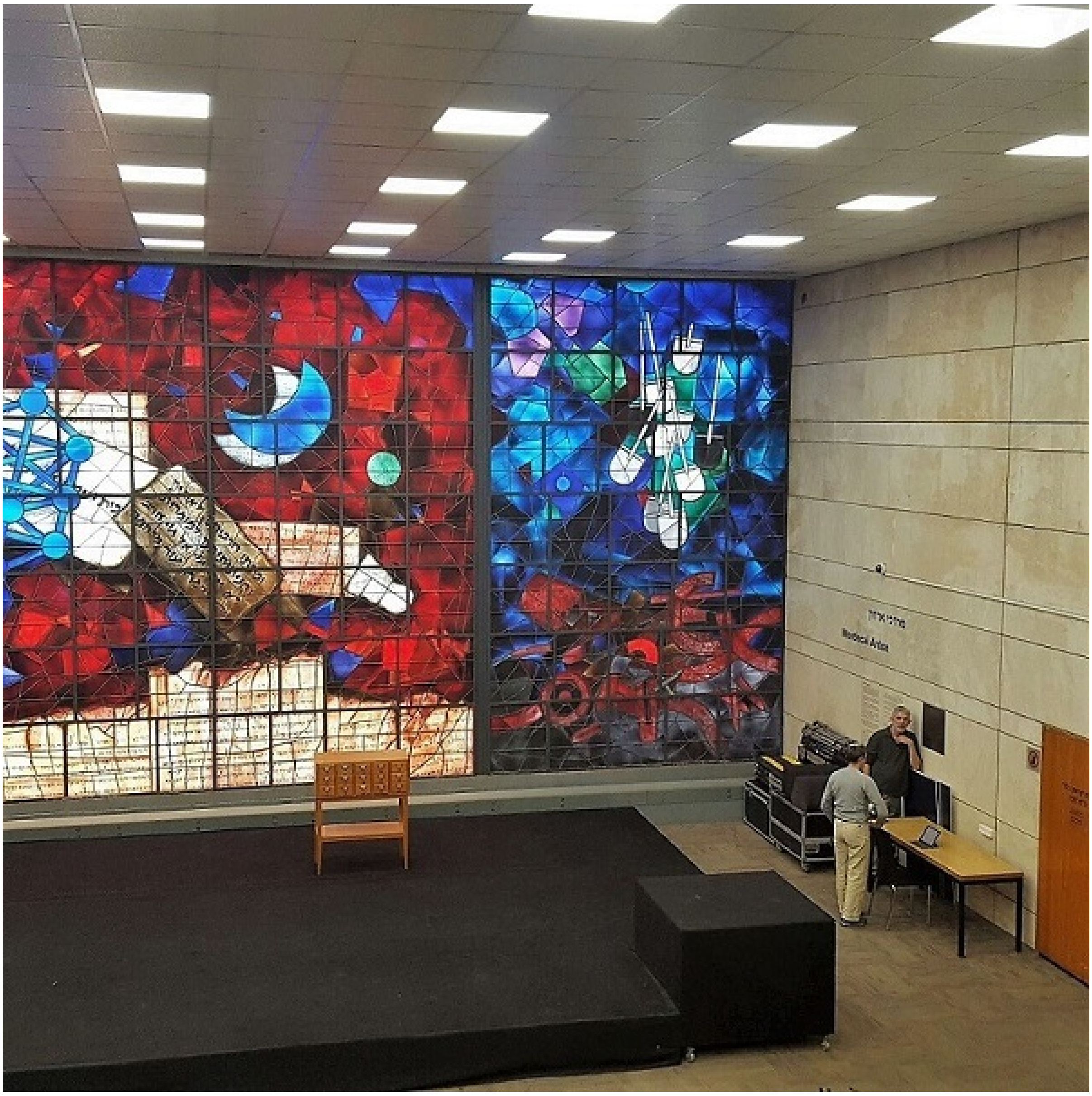
```python
        kernel = sp_signal.convolve(kernel, GAUSSIAN_BASE)
    kernel = sp_signal.convolve2d(kernel, kernel.T,) / np.sum(kernel)  # change 1D to 2D kernel and normalize
    return kernel


def blur_spatial(im: np.ndarray, kernel_size: int) -> np.ndarray:
    """
    perform image blurring using 2D convolution between the image im and a gaussian kernel g.
    :param im: the input image to be blurred (greyscale float32 image).
    :param kernel_size: the size of the gaussian kernel in each dimension (an odd integer)
    :return: blurry image (greyscale float32 image).
    """
    if min(im.shape) < kernel_size: # if kernel_size smaller than min(dimensions) of the image
        raise Exception("kernel_size must be smaller or equal to the smallest dimension of the image")

    g = gaussian_kernel_2d(kernel_size)
    blur_im = sp_signal.convolve2d(im, g, 'same', 'wrap')  # using boundaries='wrap' so we will get periodic
    return blur_im



def reduce(im: np.ndarray, blur_filter: np.ndarray) -> np.ndarray:
    """
    reduce an image by blurring and reducing image size by half
    :param im: image to reduce
    :param blur_filter: filter to use for blurring
    :return: the reduced image
    """
    reduced_im = filters.convolve(im, blur_filter, mode='mirror')
    reduced_im = filters.convolve(reduced_im, blur_filter.T, mode='mirror')
    reduced_im = reduced_im[::SAMPLE_FACTOR, ::SAMPLE_FACTOR]  # take any second element of im (if SAMPLE_FACTOR==2)
    return reduced_im


def expend(im: np.ndarray, blur_filter: np.ndarray) -> np.ndarray:
    """
    expend an image by doubling the image size and then blurring
    :param im: image to expend
    :param blur_filter: filter to use for blurring
    :return: the expended image
    """
    expended_im = np.zeros(([SAMPLE_FACTOR*dim for dim in im.shape]), dtype=np.float32)
    expended_im[1::SAMPLE_FACTOR, 1::SAMPLE_FACTOR] = im  # zero padding each odd index (if SAMPLE_FACTOR==2)
    doubled_filter = 2 * blur_filter
    expended_im = filters.convolve(expended_im, doubled_filter, mode='mirror')
    expended_im = filters.convolve(expended_im, doubled_filter.T, mode='mirror')
    return expended_im


def gaussian_kernel_1d(size: int) -> np.ndarray:
    """
    create a gaussian kernel
    :param size: the size of the gaussian kernel in each dimension (an odd integer)
    :return: gaussian kernel as np.ndarray contains np.float32
    """
    if not size % 2:  # if size is even number
        raise Exception("kernel size must be odd number")

    base = np.ones((1, 2), np.float32)  # gaussian kernel base - [1 1] vector
    kernel = base
    for i in range(size-2):
        kernel = sp_signal.convolve(kernel, base)
    kernel /= np.sum(kernel)  # normalize kernel
    return kernel


def build_gaussian_pyramid(im: np.ndarray, max_levels: int, filter_size: int) -> (list, np.ndarray):
    """
```

```
    construct a Gaussian pyramid of a given image
    :param im: a grayscale image with double values in [0, 1].
    :param max_levels: the maximal number of levels in the resulting pyramid.
    :param filter_size: the size of the Gaussian filter (an odd scalar that represents a squared filter)
    :return: tuple contains list of the pyramid levels and the filter used to construct the pyramid
    """
    filter_vec = gaussian_kernel_1d(filter_size)
    pyr = [im]
    for lvl in range(1, max_levels):
        if min(pyr[-1].shape) <= MIN_SIZE:
            break
        pyr.append(reduce(pyr[-1], filter_vec))
    return pyr, filter_vec


def build_laplacian_pyramid(im: np.ndarray, max_levels: int, filter_size: int) -> (list, np.ndarray):
    """
    Construct a Laplacian pyramid of a given image
    :param im: a grayscale image with double values in [0, 1].
    :param max_levels: the maximal number of levels in the resulting pyramid.
    :param filter_size: the size of the Gaussian filter (an odd scalar that represents a squared filter)
    :return: tuple contains list of the pyramid levels and the filter used to construct the pyramid
    """
    gauss_pyr, filter_vec = build_gaussian_pyramid(im, max_levels, filter_size)
    pyr = [gauss_pyr[i] - expend(gauss_pyr[i+1], filter_vec) for i in range(len(gauss_pyr)-1)]
    pyr.append(gauss_pyr[-1])  # add G_n level as is
    return pyr, filter_vec


def laplacian_to_image(lpyr: list, filter_vec: np.ndarray, coeff: np.ndarray) -> np.ndarray:
    """
    :param lpyr: list of laplacian pyramid images
    :param filter_vec: the filter used to create the pyramid
    :param coeff: vector of coefficient numbers to multiply each level
    :return: the reconstructed image as np.ndarray
    """
    im = lpyr[-1]*coeff[-1]
    for i in range(len(lpyr)-1, 0, -1):
        im = expend(im, filter_vec) + lpyr[i-1]*coeff[i-1]
    return im


def pyramid_blending(im1: np.ndarray, im2: np.ndarray, mask: np.ndarray,
                     max_levels: int, filter_size_im: int, filter_size_mask: int) -> np.ndarray:
    """
    pyramid blending as described in the lecture
    :param im1: first grayscale image to be blended
    :param im2: second grayscale image to be blended
    :param mask: boolean  mask representing which parts of im1 and im2 should appear in the resulting im_blend
    :param max_levels: the maximal number of levels to use in the pyramids.
    :param filter_size_im: size of the Gaussian filter used in the construction of the pyramids of im1
    and im2.
    :param filter_size_mask: size of the Gaussian filter used in the construction of the pyramid of the mask.
    :return: the blended image as np.ndarray
    """
    if im1.shape != im2.shape != mask.shape:
        raise Exception("im1, im2 and mask must agree on dimensions")

    l1, filter_vec = build_laplacian_pyramid(im1, max_levels, filter_size_im)
    l2 = build_laplacian_pyramid(im2, max_levels, filter_size_im)[PYR_IDX]
    g_m = build_gaussian_pyramid(mask.astype(np.float32), max_levels, filter_size_mask)[PYR_IDX]
    l_out = [g_m[k]*l1[k] + (1 - g_m[k])*l2[k] for k in range(len(l1))]
    im_blend = laplacian_to_image(l_out, filter_vec, np.ones(len(l_out), np.float32)).clip(0, 1)
    return im_blend
```

6 external/lib1.jpg



7 external/lib2.jpg



8 external/res pan.jpg