Computer Vision
CSCI-GA.2272-001
Assignment 1

Lakshay Sharma
ls4170@nyu.edu

1 Image Filtering

This section explores 2D image convolution, a fundamental operation in computer vision.

(a) Consider the following image X and filter kernel F:

$$X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad F = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

1. Write down the output of X convolved with F, using `valid` boundary conditions.
   Solution:
   $$X * F = \begin{bmatrix} a.z + b.y + d.x + e.w & b.z + c.y + e.x + f.w \\ d.z + e.y + g.x + h.w & e.z + f.y + h.x + i.w \end{bmatrix}$$

2. Write down the output of X convolved with F, using same boundary conditions (and zero padding).
   Solution:
   $$X * F = \begin{bmatrix} a.w & a.x + b.w & b.x + c.w \\ a.y + d.w & a.z + b.y + d.x + e.w & b.z + c.y + e.x + f.w \\ d.y + g.w & d.z + e.y + g.x + h.w & e.z + f.y + h.x + i.w \end{bmatrix}$$

(b) Give the output dimensions for a 2D image of size (h, w) convolved with a filter of size (i, j), assuming `valid` boundary conditions.

Solution: Output dimensions $(l, m) = (|h - i| + 1,\ |w - j| + 1)$

(c) Write code to perform approximate 2D Gaussian blurring of an image, using only the filter kernel [1 2 1]/4 (N.B.: you are allowed to transpose the filter). Do not use any functions that precompute a Gaussian kernel. Your code should have two inputs: (i) a 2D grayscale image of arbitrary size; (ii) an integer value specifiying the width of the kernel (in pixels), which must be odd (i.e. even widths are not permitted). The output should be a 2D (blurred) image of the

correct size, assuming `valid` boundary conditions. You are allow to use built-in convolution operations, once you have constructed the approximate Gaussian kernel. Please comment your code to describe it works.

Solution:

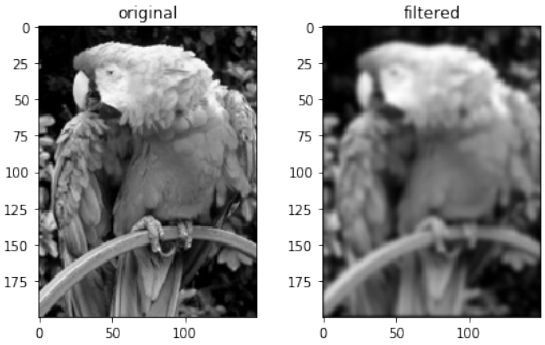
```
        def apply_gauss(img, width):
            # check if width is odd number
            if (width%2 == 0):
                raise ValueError('width parameter must be an odd number')
            # given filter
            g1 = np.array([[1,2,1]])
            g1 = g1 / np.sum(g1)
            filt = np.copy(img)

            # row-filtering, followed by column filtering
            for i in range(width):
                filt = np.apply_along_axis(np.convolve, 1, filt, np.ravel(g1), mode="same")
                filt = np.apply_along_axis(np.convolve, 0, filt, np.ravel(g1), mode="same")

            return filt

```

Listing 1: 1.(c): Guassian filtering



2 Image Alignment

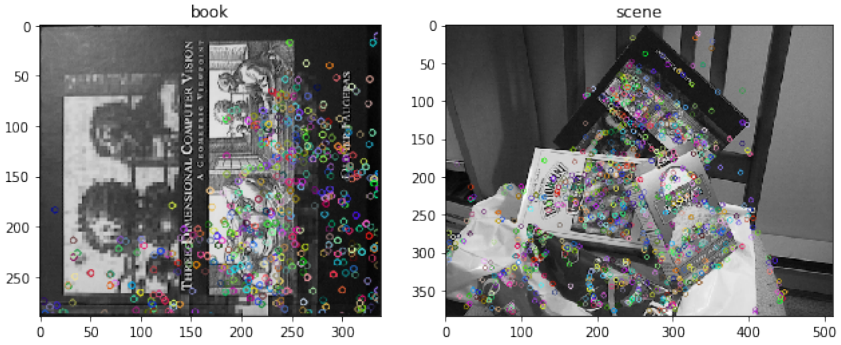
In this part of the assignment you will write a function that takes two images as input and computes the affine transformation between them. The overall scheme, as outlined in lecture 5 and 6, is as follows:

1. Find local image regions in each image
2. Characterize the local appearance of the regions
3. Get set of putative matches between region descriptors in each image
4. Perform RANSAC to discover best transformation between images

2.1

The first two stages can be performed using David Lowe's SIFT feature detector and descriptor representation. You should first run the SIFT detector over both images to produce a set of regions, characterized by a 128d descriptor vector. Display these regions on each picture to ensure that a satisfactory number of them have been extracted.

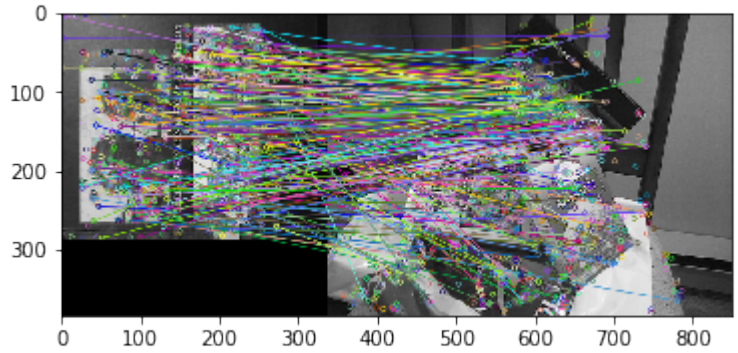
Solution:



2.2

The next step is to obtain a set of putative matches T. This should be done as follows: for each descriptor in image 1, compute the closest neighbor amongst the descriptors from image 2 using Euclidean distance. Spurious matches can be removed by then computing the ratio of distances between the closest and second-closest neighbor and rejecting any matches that are above a certain threshold. To test the functioning of RANSAC, we want to have some erroneous matches in our set, thus this threshold should be set to a fairly slack value of 0.9. To check that your code is functioning correctly, plot out the two images side-by-side with lines showing the potential matches (include this in your report).

Solution:



2.3

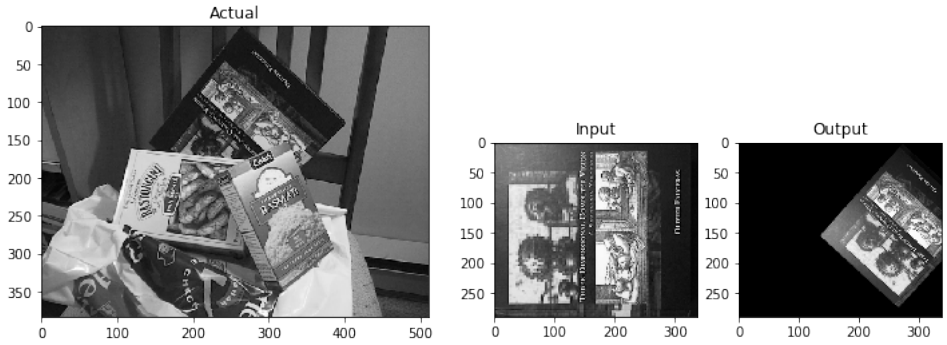
The final stage, running RANSAC, should be performed as follows:

- Repeat N times (where $N \approx 100$):
  - Pick P matches at random from the total set of matches T. Since we are solving for an affine transformation which has 6 degrees of freedom, we only need to select $P = 3$ matches.
  - Construct a matrix A and vector b using the 3 pairs of points as described in lecture 6.
  - Solve for the unknown transformation parameters q.
  - Using the transformation parameters, transform the locations of all T points in image 1. If the transformation is correct, they should lie close to their pairs in image 2.
  - Count the number of inliers, inliers being defined as the number of transformed points from image 1 that lie within a radius of 10 pixels of their pair in image 2.
  - If this count exceeds the best total so far, save the transformation parameters and the set of inliers.
- Perform a final refit using the set of inliers belonging to the best transformation you found. This refit should use all inliers, not just 3 points chosen at random.
- Finally, transform image 1 using this final set of transformation parameters, q.

Your report should include: (i) the transformed image 1 and (ii) the values in the matrix H.

Solution:

$$H = \begin{bmatrix} 0.40775082 & 0.45405538 & 133.50747424 \\ -0.44805113 & 0.41513414 & 154.44531119 \end{bmatrix}$$



3 Estimating the Camera Parameters

Here the goal is the compute the 3x4 camera matrix P describing a pinhole camera given the coordinates of 10 world points and their corresponding image projections. Then you will decompose P into the intrinsic and extrinsic parameters. You should write a simple Matlab or Python script that works through the stages below, printing out the important terms.

Download from the course webpage the two ASCII files, `world.txt` and `image.txt`. The first file contains the (X,Y,Z) values of 10 world points. The second file contains the (x,y) projections of those 10 points.

(a) Find the 3x4 matrix P that projects the world points $\mathbf{X}$ to the 10 image points $\mathbf{x}$. This should be done in the following steps:

1 Since P is a homogeneous matrix, the world and image points (which are 3 and 2-D respectively), need to be converted into homogeneous points by concatenating a 1 to each of them (thus becoming 4 and 3-D respectively).

2 We now note that $\mathbf{x} \times P\mathbf{X} = 0$, irrespective of the scale ambiguity. This allows us to setup a series of linear equations of the form:

$$\begin{bmatrix} 0^T & -w_i \boldsymbol{X}_i^T & y_i \boldsymbol{X}_i^T \\ w_i \boldsymbol{X}_i^T & 0^T & -x_i \boldsymbol{X}_i^T \\ -y_i \boldsymbol{X}_i^T & x_i \boldsymbol{X}_i^T & 0^T \end{bmatrix} \begin{pmatrix} P^1 \\ P^2 \\ P^3 \end{pmatrix} = 0$$

for each correspondence $\mathbf{x}_i \leftrightarrow \boldsymbol{X}_i$, where $\boldsymbol{x}_i = (x_i, y_i, w_i^T)$, w_i being the homogeneous coordinate, and P^j is the j^{th} row of P. But since the 3rd row is a linear combination of the first two, we need only consider the first two rows for each correspondence i. Thus, you should form a 20 by 12 matrix A, each of the 10 correspondences contributing two rows. This yields $Ap = 0$, p being the vector containing the entries of matrix P.

3 To solve for p, we need to impose an extra constraint to avoid the trivial solution $p = 0$. One simple one is to use $||p||_2 = 1$. This constraint is implicitly imposed when we compute the SVD of A. The value of p that minimizes Ap subject to $||p||_2 = 1$ is given by the eigenvector corresponding to the smallest singular value of A. To find this, compute the SVD of A, picking this eigenvector and reshaping it into a 3 by 4 matrix P.

4 Verify your answer by re-projecting the world points $\mathbf{X}$ and checking that they are close to $\mathbf{x}$.

Solution:

$$P = \begin{bmatrix} -1.27000127e-01 & -2.54000254e-01 & -3.81000381e-01 & -5.08000508e-01 \\ -5.08000508e-01 & -3.81000381e-01 & -2.54000254e-01 & -1.27000127e-01 \\ -1.27000127e-01 & 5.41233724e-16 & -1.27000127e-01 & -6.66133814e-16 \end{bmatrix}$$

Average error in $Ap = 0$ calculation: 2.10942374e-16
Average world-to-image projection error: 5.61772850e-15

(b) Now we have P, we can compute the world coordinates of the projection center of the camera C. Note that $PC = 0$, thus C lies in the null space of P, which can again be found with an SVD. Compute the SVD of P and pick the vector corresponding to this null-space. Finally, convert it back to inhomogeneous coordinates and to yield the (X,Y,Z) coordinates. Your report should contain the matrix P and the value of C.

In the alternative route, we decompose P into itâs constituent matrices. Recall from the lectures that $P = K[R|t]$. However, also, $t = -R\tilde{C}$, $\tilde{C}$ being the inhomogeneous form of C. Since K is upper triangular, use a RQ decomposition to factor KR into the intrinsic parameters K and a rotation matrix R. Then solve for $\tilde{C}$. Check that your answer agrees with the solution from the first method.

Solution:

Average error in $PC = 0$ calculation: 7.17019036e-17

$$P = \begin{bmatrix} -1.27000127e-01 & -2.54000254e-01 & -3.81000381e-01 & -5.08000508e-01 \\ -5.08000508e-01 & -3.81000381e-01 & -2.54000254e-01 & -1.27000127e-01 \\ -1.27000127e-01 & 5.41233724e-16 & -1.27000127e-01 & -6.66133814e-16 \end{bmatrix}$$

C (in homogeneous coordinates) = $\begin{bmatrix} -0.5 & 0.5 & 0.5 & -0.5 \end{bmatrix}$

C (in Cartesian coordinates) = $\begin{bmatrix} 1 & -1 & 1 \end{bmatrix}$

4 Structure from Motion

In this section you will code up an affine structure from motion algorithm, as described in the slides of lecture 6. For more details, you can consult page 437 of the Hartley & Zisserman book.

Load the file `sfm_points.mat` (included in `assignment1.zip`). The file contains a 2 by 600 by 10 matrix, holding the x, y coordinates of 600 world points projected onto the image plane of the camera in 10 different locations. The points correspond, that is `image_points(:,1,:)` is the projection of the same 3D world point in the 10 frames.

The points have been drawn randomly to lie on the surface of a transparent 3D cube, which does not move between frames (i.e. the object is static, only the camera moves). Try plotting out several frames and the cube shaped structure should be apparent (the `plot3` command may be useful).

To simplify matters, we will only attempt an affine reconstruction, thus the projection matrix of each camera i will have following form:

$$P^i = \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} M^i & t^i \\ 0 & 1 \end{pmatrix}$$

where M_i is a 2 by 3 matrix and t^i is a 2 by 1 translation vector.

So given $m = 10$ views and $n = 600$ points, having image locations $\mathbf{x}_j^i$,where $j = 1, ..., n$, $i = 1, ..., m$, we want to determine the affine camera matrices M^i, t^i and 3D points $\mathbf{X}_j$ so that we minimize the reconstruction error:

$$\sum_{ij} ||\mathbf{x}_j^i - (M^i \mathbf{X}_j + t^i)||^2$$

We do this in the following stages:

- Compute the translations t^i directly by computing the centroid of point in each image i.
- Center the points in each image by subtracting off the centroid, so that the points have zero mean
- Construct the $2m$ by n measurement matrix W from the centered data
- Perform an SVD decomposition of W into UDV^T
- The camera locations M^i can be obtained from the first three columns of U multiplied by $D(1:3, 1:3)$, the first three singular values

- The 3D world point locations are the first three columns of V
- You can verify your answer by plotting the 3D world points out. using the `plot3` command. The `rotate3d` command will let you rotate the plot. This functionality is replicated in Python within the `matplotlib` package.

You should write a script to implement the steps above. The script should print out the M_i and t_i for the first camera and also the 3D coordinates of the first 10 world points. Cut and paste these into your report.

Solution:

$$M^1 = \begin{bmatrix} -7.50914219 & 3.30837904 & -3.71763726 \\ 0.17858821 & -8.56620251 & -2.47587867 \end{bmatrix}$$

$$t^1 = \begin{bmatrix} 2.36847579e-17 \\ 8.28966525e-17 \end{bmatrix}$$

3-D coordinates of first 10 world points

$$V(1:10, 1:3) = \begin{bmatrix} 5.771626204845541758e-03 & 6.460628198335639782e-02 & -2.497615250861577943e-02 \\ 5.760996884595459382e-04 & 6.885363051285205149e-02 & -3.458150968948404264e-02 \\ -4.293584908198630479e-02 & 6.330478970108092962e-02 & 2.861711308243371221e-02 \\ 4.745038348981937826e-02 & 4.904206534905620335e-02 & -1.257547261612534592e-02 \\ -4.210186026210110261e-02 & 6.789239263177053452e-02 & 1.175163704249897363e-02 \\ 5.961963774737105010e-02 & 4.605179976154472055e-02 & -1.438374286782062583e-02 \\ 9.091667080625415523e-03 & 6.002048918269967809e-02 & -1.229997037439519054e-02 \\ 1.039489479072670808e-02 & 4.602065384765107597e-02 & 3.529274761049663867e-02 \\ -2.589080533495650635e-02 & 5.702972054986122502e-02 & 3.337374752589779975e-02 \\ 1.745597660010627869e-02 & 4.054263693137879349e-02 & 4.731859342052484318e-02 \end{bmatrix}$$