Computer Vision
Lab Exercise 6
Chaining & 3 D Reconstruction pipeline

In a video sequence, a single point is unlikely to remain visible in all the frames. Thus we need to keep track of the points that are present from one frame to the next. We do this by means of a point view matrix (also called observation matrix) which is constructed for all the video frames. It saves the coordinates of the correspondence between two consecutive frames. Later Tomasi-Kanade factorization (introduced in lab 4 for structure from motion) can be performed on the sub-blocks of the point view matrix for a 3D reconstruction. The sub blocks of the point view matrix each represent point correspondences across a particular set of views, you can obtain multiple point clouds from factorizing these sub blocks. Then you can use procrustes analysis to compute transforms to align the point clouds (and perform bundle adjustment if necessary) to obtain the final 3D shape.

1 Chaining

Construct point-view matrix with the matches found across image pairs for all consecutive tedy bear images (1-2, 2-3, 3-4, ..., 15-16, 16-1). The point-view matrix has views in the rows, and points in the columns: $\#frames \times \#matches$. You can construct it with the `ChainImages.m` function which takes as input, a cell array containing matches(each cell contains matches between a pair of images). The steps are elaborated below.

1. Initialize the point view matrix (The variable 'PV') with rows equal to the number of frames you have + 1. (The additional frame is to ease our effort and will be removed later)

2. Iterate over the frames saving the matches between each pair of images to the variable 'newmatches'.

3. Initialize the point view matrix with zeros and size equal to

$$1 + number of frames x the number of matches$$

between the image pairs. For the first pair, simply add the indices of matched points to the same column of the first two rows of the point-view matrix.

```
1       if i==1
2           PV(1:2,:) = ...
```

4. Use the `intersect` function of MATLAB to find points that have been previously found. The indices from the intersection (IA and IB) give the column indices for assigning the correspondences of 'newmatches' and 'prevmatches' to the next frame (i+1th frame).

```
1       % Find already found points using intersection on ...
            PV(i,:) and newmatches
2       [¬, IA, IB]  = ...
3       PV(i+1,... ) = intersection(...  ,...  )
```

5. Use the `setdiff` MATLAB function to detect new matches('diff') and their respective indices that are not present in the previous set. For subsequent image frames until the last first frame pair, you will progressively grow the size of your point view matrix. You can 'grow' your point-view matrix by simply appending zeros. For example `PV = [PV zeros(size`$_x$`,size`$_y$`)]`. Since you will need to add newly found matches to this matrix the zeros should have columns equal to the number of your new matches(diff) and rows = frames + 1

```
1       % Find new matching points that are not in the ...
            previous match set using setdiff.
2       [diff, IA] = setdiff(...  ,...  )
3
4       % Grow the size of the point view matrix each ...
            time you find a new match.
```

```
start = size(PV,2)+1;
PV    = [PV zeros(frames+1, size(diff,2))];
PV(i, start:end)   = ...
PV(i+1, start:end) = ...
```

Assign the newly found matches to the current view PV(i) and the intersecting points with the previous frame is assigned to the next PV(i+1)

6. The very last frame-pair, consisting of the last and first frames, requires special treatment. Take the common matches between the first and last frame and move it to the corresponding columns in the first frame and delete the last frame.

```
% Process the last frame-pair. This part is ...
    already completed by TAs.
% The last frame-pair, consisting of the last and ...
    first frames, requires special treatment.
% Move matches between last & 1st frame to their ...
    corresponding columns in
% the 1st frame, to prevent multiple columns for ...
    the same point.
[¬, IA, IB]      = intersect(PV(1, :), PV(end, :));
PV(:, IA(2:end)) = PV(:, IA(2:end)) + PV(:, ...
    IB(2:end));  % skip 1st index (contains zeros)
PV(:, IB(2:end)) = [];  % delete moved points in ...
    last frame
```

7. Find indices of non-zero elements in the first and last rows of the point-view matrix using the MATLAB function `find`. You need to copy the non zero elements from the last row which are not in the first row to the first row. Here you can use the not ismember function `ismember`

```
% Copy the non zero elements from the last row ...
    which are not in the first row to the first row.
nonzero_last  = find(PV(end, :));
nonzero_first = find(PV(1, :));
no_member     = ¬ismember(nonzero_last, ...
    nonzero_first);
nonzero_last  = nonzero_last(no_member);
tocopy        = PV(:, nonzero_last);
```

8. To make the whole process 'circular', we move points indicated by the 'tocopy' variable to the first row.

```
% Place these points at the very beiginning of PV
PV(:, nonzero_last) = [];
PV                  = [tocopy PV];

% Copy extra row elements from last row to 1st ...
    row and delete the last row
PV(1 ,1:size(tocopy, 2)) = PV(end, 1:size(tocopy, ...
    2));
PV                       = PV(1:frames,:);
```

We provide on BrightSpace an example of a point-view matrix for your references.

2 3D Reconstruction Pipeline

In this section you are provided with the 3D reconstruction pipeline that you have to follow to obtain your final reconstructed 3D object from the provided images. The pipeline uses the following set of steps:

1. Given a directory with images: first read those images, then detect interest points and extract SIFT features (Lab 3). Then use these features to compute matches between all consecutive images, as well as between the last and image. The matches are found using the 8-point algorithm (Lab 5). Write these matches in a cell array.

```
[C,D, Matches]=ransac_match(directory);
save('Matches.mat', 'Matches');
```

2. Then you have to create a point-view matrix as explained in the above "Chaining" section.

```
PV = chainImages(Matches);
```

3. You have to loop over images by taking 3 consecutive images at a time and using the corresponding point-view matrix block to reconstruct the $3D$ points. From these chained matches you have to select only the column indexes that do not have non-zero entries. You also have to check that the number of correspondences in all views is larger than 8:

```
    for iBegin = 1:n-(numFrames - 1)
        iEnd = ...

        % Select frames from PC to form a block
        block = ...

        % Select columns from the block that do not ...
           have any zeros
        colInds = find(...);

        % Check the number of visible points in all views
        numPoints = size(colInds, 2);
        if numPoints < 8
            continue
        end

        ...
```

4. Then you have to create a measurement matrix (as the one used in Lab4) from these correspondences points, where the measurement matrix has the size: $2\#frames \times \#matches$, and the odd rows correspond to x coordinates and even rows to y coordinates.

```
    % Create measurement matrix X with coordinates ...
       instead of indices using the block and the ...
       Coordinates C
    block = block(:, colInds);
    X = zeros(2 * numFrames, numPoints);
    for f = 1:numFrames
        for p = 1:numPoints

            X(2 * f - 1, p) = ...
            X(2 * f, p)     = ...
        end
    end
```

5. The we reconstruct the $3D$ point clouds for the current 3 images using the structure from motion implementation (from Lab4). Sometimes finding a reliable solution may not be possible between the image matches. In this case the Choleski decomposition will return a non-positive matrix error.

```
% Estimate 3D coordinates of each block following ...
    Lab 4 "Structure from Motion" to compute the ...
    M and S matrix.
% Here, an additional output "p" is added to deal ...
    with the non-positive matrix error
% Please check the chol function inside sfm.m for ...
    detail.
[M, S, p] = ... % Your structure from motion ...
    implementation for the measurements X

if ¬p
    % Compose Clouds in the form of (M,S,colInds)
    Clouds(i, :) = {M, S, colInds};
    i = i + 1;
end
```

We only add the current points to the cloud cell array if the Choleski decomposition was successful.

6. We now need to stitch the obtained point clouds into a single point cloud. We use the point correspondences to find the optimal transformation between shared points in the $3D$ point clouds. The *procrustes* analysis is used for this. We start by initializing the merged cloud points with the main view, given by the first set of point clouds.

   We then find the points that are both in the new cloud and in the already merged cloud. The *procrustes* analysis needs a minimum of 15 points. We then apply the *procrustes* analysis to find the transformation between the points. $Procrustes(X, Y)$ finds the linear transformation of the set of points in Y, that best conforms them to the points in X.

```
% Get the points that are in the merged cloud ...
    and the new cloud by using "intersect" ...
    over indexes
```

```
2          [sharedInds, ¬, iClouds] = intersect(...)
3          sharedPoints             = ...
4
5          % Find optimal transformation between shared ...
               points using procrustes. The inputs must ...
               be of the size [Nx3].
6          [d, Z, T] = procrustes(...', ...')
```

7. We then need to select the points that are not yet in the merged cloud using "setdiff". We transforms the points with the found transformation using the formula:

$$Z = T.b * Y * T.T + T.c. \tag{1}$$

Where $T.c$ is the translation component, $T.T$ is the orthogonal rotation and reflection component and $T.c$ is the scale component.

Note: The matlab function *procrustes* expects inputs of the form $[N \times 3]$, so that means we also need to transpose our points when computing their transformed version.

```
1      % Find the points that are not shared between the ...
           merged cloud and the Clouds{i,:} using ...
           "setdiff" over indexes
2      [iNew, iCloudsNew] = setdiff(..)
3
4      ...
5
6      % Transform the new points using: Z = (T.b * Y' * ...
           T.T + c)'.
7      % and store them in the merged cloud, and add ...
           their indexes to merged set
8      mergedCloud(:, iNew) = ...
9      mergedInds           = ...
```

The obtained points can the be visualized with the function *scatter*3.