# Lab Course: Robot Vision WS 2012/2013

**Philipp Heise, Brian Jensen, Sebastian Klose**

**Assignment 2 - Due: 19.11.2012**

## Exercise 1 Simple Image Feature Descriptors and Matching

In the previous assignment sheet you implemented a method for extracting key points from images using the Harris corner detector. In this exercise you will expand upon this central component and match key points between different images of the same scene with the help of feature descriptors. The methods employed in this exercise are based upon techniques described in "Multi-Image Matching using Multi-Scale Oriented Patches" by Brown, Szeliski, and Winder (see http://research.microsoft.com/apps/pubs/default.aspx?id=70120).

1. *(Simple Image Patch Based Feature Descriptor)* To get started you are going to implement a basic feature descriptor containing values taken from a square image patch centered around the keypoint. This feature descriptor is characterized by a single parameter, `window_size`, that specifies the side length of the square patch. Image values are taken directly from the grayscale image and stored in an array.

   - For this exercise you should either extend your `harris` ROS package from the previous exercise or create a new ROS package named `feature_<Group Name>` (Recommended).
   - Create a feature descriptor base class that will serve as the interface for all of the feature descriptor implementations in this exercise. The feature descriptor base class should have member variables for encapsulating the following:
     - Information about the Harris key point associated with the descriptor, such as location and magnitude.
     - A data buffer for storing the descriptor values.

     You should also define a couple of abstract interface methods (with default implementations where appropriate) that all feature descriptors have to implement as a minimum:
     - An instance method that calculates the distance between the feature descriptor instance and a second descriptor passed as an argument, returning a scalar value. The distance metric used should be appropriate for the two feature descriptor types.
     - An instance method that draws the feature descriptor on an OpenCV image argument.
   - Extend your feature descriptor base class for square image patch feature descriptors. This class should have an additional parameter specifiying the square side length. Upon construction this patch descriptor class should extract the correct values from the grayscale source image determined by the square size, optionally performing bounds checking if appropriate.

- Implement the distance instance method of your patch feature descriptor using a metric of your choice. One simple way to implement the distance is to treat the descriptor's array as a high dimensional vector $v$ and build a *sum of squared differences* $\sum_i (v_i - u_i)^2$ or a *sum of absolute difference* $\sum_i (v_i - u_i)$ between each individual values $u_i$, $v_i$ in both vectors.
- Implement the draw instance method for your patch feature descriptor. This method takes an OpenCV image as an argument and draws a square centered at the key point location. The square size should match the descriptor size.

2. *(Simple Feature Descriptor Matching)* Now that you have implemented your first feature descriptor you are ready to start finding matches between images. In this exercise you are going to create a generic matching function for all feature descriptor types that looks for matches for each feature descriptor contained in a source array with feature descriptors in a destination array.
   - Define a type for representing a match between two feature descriptors.
   - Create a matching function that takes two feature descriptor arrays, a source and destination array, and returns an array of feature descriptor matches using the type defined in the previous subtask. The feature descriptor arrays should be generic and work for all of your feature descriptor types.
   - The matching function should use a brute force search approach ($O(n^2)$ search algorithm). Each feature descriptor in the source array should caculate its distance to each descriptor in the destination array. The element in the destination array with shortest distance is treated as a match and added to the matches array.

3. *(Template Matching and Tracking Node)* With your first feature descriptor and feature matching implementations complete its time to put them to the test. You will create a new node for template matching and tracking that will be used throughout the rest of the assignment sheet. The new node takes a template image file and subscribes to an image topic, calculating feature matches between the template image and each new incoming image message. The node then publishes an output image containing the template image and the incoming image drawn side by side, as well the feature descriptors and matches.
   - Create a new ROS node for performing feature matching and visualization.
   - The new ROS node should support one non dynamic parameter `file`: the file name of template image.
   - The node should load the template image upon start.
   - The node should subscribe to the image topic `in`.
   - The node should use the Harris implementiation from the previous assignment sheet to extract key points from each new image. Use your patch feature descriptor from the previous task to generate a feature descriptor for every keypoint. Use the matching function to find matches from the template image to the incoming image.
   - The output image image should contain the template image on the left and the incoming image on the right. Use the draw function of your patch feature descriptor to draw the feature descriptors from both the template and the incoming image on the output image. Draw lines between the feature descriptor matches on the output image. Publish the resulting output image on the topic `image` *Note: the OpenCV ROI extraction functions* `cv::Mat::operator()` *may be helpful here.*
   - Your implementation should initially support one dynamically reconfigurable parameter: `window_size` the side length of the square window used by your patch feature descriptor. A good default here is 5 or 7.

4. *(Orientation)* Once simple way to improve the accuracy and robustness of your template matching node is to use oriented patch feature descriptors. Like the patch feature descriptor the oriented patch descriptor samples the grayscale image values in a sqaure window centered around the key point. However, the oriented feature descriptor first rotates each point in the window's grid around the key point by the key point's orientation before perorming sampling.

   - Extend your Harris implementation to additionally output the orientation with each detected key point. The orientation $\theta$ of a key point at $(x, y)$ can be estimated by the local image gradient vector $u(x, y)$ where:

     $$(\cos\theta, \sin\theta) = \frac{u}{\|u\|_2}, \qquad u(x, y) = \nabla I(x, y)$$

     To improve the robustness of the orientation estimation it may be necessary to use finite difference operator $\nabla$ with a larger kernel size.

   - Extend your feature descriptor base class for oriented patch festure descriptors. The oriented patch feature descriptor also has an additional parameter `window_size`. Each coordinate in the window's grid is transformed according to the formula:

     $$w(x, y)' = Rw(x, y) + p \qquad R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

     where $w(x, y)$ is the window grid coordinate relative key point image location $p$. Use nearest neighbor interpolation for sampling image values. Also use the same distance metric as with your simple square patch feature descriptor.

   - Add a dynamically reconfigurable parameter to your template matching node from the previous task to optionally use oriented feature descriptors: `oriented_desc`.

   - Extend the patch extraction from the previous subtask to optionally use bilinear interpolation. Your implementation should support one additional dynamically reconfigure node parameter `linear_interpolation`.

5. *(Adaptive Non Maximal Suppression)* Another method for improving matching results is based on the idea of having the key points more evenly spread throughout the image, a technique known as *Adaptive Non Maximal Suppression*. With a threshold approach, key points with the highest response are often clumped together in the image. With ANMS instead of taking key points with highest response, key points are chosen that are locally maximal inside of a radius $r_i$ around the point $x_i$ as specified by the formula:

   $$r_i = \min_j |x_i - x_j| \quad \text{where} \quad H(x_i) < c_r H(x_j) \quad x_j \in I$$

   where $H(x)$ is the Harris response at a point $x$ and $c_r < 1$ a constant ensuring that the next largest neighbor is significantly larger to cause suppression (typically a value of 0.9 is used). In other words we are looking for key points with the largest radius to the next significantly larger neighbor, which will result in the accepted key points being more evenly distributed throughout the image.

   - Extend your Harris implementation by adding an additional filtering function for adaptive non maximal suppression as an alternative to thresholding.

   - The ANMS function should take an array of key point candidates and either return an array of filtered key points, or modify the key point array argument in place.

   - Your ANMS implementation should start by sorting the key point candidates according to their response strength.

- Starting with the point with the second strongest response, you need to calculate the distance to all points with a significantly higher response. The minimum distance for each point should then be saved.
- After processing each point return the points with the largest minimum distance (by either modifying the input array in place or returning a new array).
- Your implementation should support two dynamically reconfigurable parameters:
  - `ANMS`: a boolean parameter determining whether thresholding or ANMS is used.
  - `ANMS_points`: The number of key points your ANMS function should output.

6. *(Improved matching)* One key weakness of your matching function implemented in the previous task is that it always returns a match for each descriptor in the source array, leading to many erroneous matches. One simple way to improve this is to compute a match confidence for each source feature descriptor by comparing the distance to the nearest neighbor $d_{1-NN}$ with the distance to the second nearest neighbor $d_{2-NN}$ and only accepting matches below a certain threshold $t_m$:

$$\frac{d_{1-NN}}{d_{2-NN}} < t_m \qquad \text{where} \quad t_m < 1$$

and thus only accepting matches whose distance is significantly lower than the next best candidate.

- Modify your matching function to only accept matches where the ratio of nearest neighbor neighbor to second nearest neighbor is below a threshold parameter. A good default value here is to only accept matches where the ratio is below 0.75.
- Extend the template tracking node to support a dynamically reconfigurable parameter: `ratio_threshold` the ratio threshold for accepting matches.

7. *(Multiscale Detectors and Descriptors)* As a final addition to your key point detector and feature descriptors you will add multi scale support to gain some scale invariance. Using a downsampled image pyramid of the input image reduced by increasing powers of two, you should be able to add some amount scale invariance with minimal changes to your Harris and feature descriptor implementations.

- In your template matching node create a downsampled image pyramid using the `cv::pyrDown()` function for both the template and the incoming image. Each level in the pyramid should be half the size of the previous level.
- Add a new function to your Harris implementation that takes an image pyramid as an argument and extracts key points at level in the image pyramid. Filtering (thresholding or ANMS) should be run over all key point candidates at all levels (not for each level individually). The function should return an an array of key points additionally including information about the scale at which the key point was detected.
- Generate feature descriptors for the appropriate for key point scale using the correct level of the image pyramid.
- Add support to the template tracking node for a dynamically reconfigurable parameter: `scales` the number of scales to use.

## Exercise 2 Homography Estimation

In this exercise, you are going to use the matches of the last exercise to estimate the homography between the two point pairs.

1. *(Direct Linear Transformation) See slides for more details on DLT*

- Implement a function that computes the $3 \times 3$-Homography matrix from a number of matches (*Note: you need at least 4 matches!*)
- Add the computation to the node from the previous exercise. Also draw the warped rectangle from the template into the debug output image from the previous exercise.
- Add a dynamic reconfigure option to enable and disable the homography estimation

*Note: As data input for this and the following exercise, you can either use your calibrated webcam or the bag file and template available at*

- http://www6.in.tum.de/~kloses/rvc/graffity.bag
- http://www6.in.tum.de/~kloses/rvc/graffity.png

.

## Exercise 3 Template Tracking/Detection

When using Homographies together with calibrated cameras, the Homography can be ugraded to a full 3D pose (Rotation and Translation) using a simple trick: When transforming a point on a plane (say $z = 0$)) it follows that:

$$\begin{bmatrix}\mathbf{KR} & \mathbf{Kt}\end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix}\mathbf{r}_0 & \mathbf{r}_1 & t\end{bmatrix} \sim \mathbf{H}$$

with $\mathbf{r}_i$ being the i-th column of the rotation matrix $\mathbf{R}$. By further exploiting the orthonormality of the rotation matrix, one can recover the full Rotation matrix and the translation from a Homography matrix.

1. *(Homography to 3D)*
   - implement a function to recover the 3D rotation and translation from a homography matrix
   - add a subscriber to your node for the `sensor_msgs/CameraInfo` topic published from your calibrated camera and copy the intrinsics from the message
   - extend the node from the previous exercise to recover the 3D pose from the estimated Homography
2. *(Result Publishing)*
   - In order to visualize your results, you have to publish a `tf` frame from your code containing the 3D transformation - you can either use `tf` or `tf2`
     - http://ros.org/wiki/tf
     - http://ros.org/wiki/tf2

     *Note: To convert the* $3\times3$ *Rotation Matrix to a Quaternion, you can use e.g.* `bullet` *or* `eigen`*, or implement your own conversion*
   - Use your node from assignment sheet 1 to visualize the 3D logo following the template movements of the `tf` frame in `rViz`. Note: in the provided bag-file we moved the camera instead of the template, so you probably want to switch the frames to see the moving frame.