Robot Vision Course SS 2013

Philipp Heise, Brian Jensen, Sebastian Klose

Assignment 2 - Due: 22.05.2013

Exercise 0 Prerequisites

We have provided some common message definitions and headers required to complete the exercises into a utilities package available here: https://github.com/tum-uav/rvc_utils. For this assignment we will be using the affine covariant features evaluation dataset (see http://www.robots.ox.ac.uk/~vgg/research/affine/) that we have repackaged into rosbag files for convenience, also contained in this repository.

Exercise 1 MOPS - Multi-Scale Oriented Patches Descriptor

In the previous assignment sheet you implemented a method for detecting key points in images using the Harris corner detector, and you have likely tested other alternative key point detectors. In this exercise you will take the next step and extract descriptors for each detected key point that contain discriminative information about the image patch immediately around the key point in order to match key points between different images of the same scene. 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 Patches Feature Descriptor)* To get started you are going to implement a simple feature descriptor containing values sampled from a square image patch centered around the key point. In its most basic form this feature descriptor is characterized by a single parameter, `patch_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 extend your *<Group Name>*`_features` package from the previous assignment sheet.
   - Create a new "MOPS" feature descriptor class in your package. This class should implement the OpenCV interface: `cv::DescriptorExtractor`.
   - Your MOPS feature descriptor implementation (in `computeImpl(...)`) should initially copy values straight from the image for each key point in the patch $r = [k_x - \frac{l}{2}, k_x + \frac{l}{2}] \times [k_y - \frac{l}{2}, k_y + \frac{l}{2}]$ where k_x, k_y are the key point's location and l the **patch_size**. The descriptor values d_i are then defined as $d_i = I(x_i, y_i), \forall\, x_i, y_i \in r$. You can initially ignore the **octave** and **angle** key point parameters. The resulting `cv::Mat` of feature descriptors should have one row for each input key point containing the corresponding descriptor of size **patch_size**^{2} and use the same element datatype as the image.
   - Create a new ros node class for controlling the feature descriptor extraction.
     - This class should maintain a pointer to an instance of `cv::DescriptorExtractor`.

  - It should have a public member function with signature similar to `cv::DescriptorExtractor::compute` that simply calls its `DescriptorExtractor` instance pointer with the provided parameters.
- The parameter **patch_size** should be available via `dynamic reconfigure`. A good default here is a patch size of 15.

2. *(Simple Patches Feature Matching)* Now that you have implemented your initial feature descriptor you are ready to implement the next part of the processing pipeline: feature descriptor matching. You are going to create a generic feature matching implementation that should work for all types of feature descriptors under the condition that they have the same length and data type.
   - Create a new feature matcher class. This class can implement either the OpenCV `cv::DescriptorMatcher` interface or simply a method with the signature:
     `void match(const Mat& srcDescriptors, const Mat& destDescriptors, vector<DMatch>& matches) const`
   - The descriptor matcher implementation should use exhaustive search to find the best matching descriptor in the destination descriptor array for each descriptor in the source array. Since each descriptor can be though of as a high dimensional vector, the simplest way to define the best match between two descriptors is the match with the shortest distance. There are several ways to define the distance between vectors, for the this exercise you should implementing one or both of the following:
     - The ℓ^2 norm (*sum of squared differences*): $\sum_i (s_i - d_i)^2$ where s_i and d_i are the source and destination descriptor values respectively.
     - The ℓ^1 norm (*sum of absolute differences*): $\sum_i |s_i - d_i|$ where s_i and d_i are the source and destination descriptor values respectively.
   - Use the type `cv::DMatch` for representing matches.
   - Create a `drawMatches` function for visualizing the feature matches. It should combine the source and destination images side by side into a new image and draw lines between the matching locations in the source and destination images.
     Note: the OpenCV ROI extraction function `cv::Mat::operator()` and the copy function `cv::Mat::copyTo()` may be helpful here.

3. *(Feature Matching Node)* With your first feature descriptor and feature matching implementations complete its time to put them to the test.
   - Create a new executable in your package named `feature_matcher`. It will use your implementation files from the previous assignment sheet plus the newly created descriptor extractor and matcher.
   - Create a new ROS node for controlling your feature matching implementation.
     - This new node should subscribe to two input image message topics, source and destination, and publish an output image topic visualizing the matched key points between the source and destination images.
     - In order to ease internal sychnronization all corresponding messages published in the bag files for this assignment are time synchronized. You can use the `message_filters::TimeSynchronizer` ROS class for subscribing to sychronized message topics and receiving a single combined call back.
     - The new node should be initialized with instances of your feature detector and feature descriptor nodes.
   - Upon receiving an incomcing image message pair, the node should:
     - detect key points from both images using your feature detector node instance

  - extract descriptors from both images using your descriptor extractor node.
  - match the source image descriptors to the destination image descriptors.
  - visualize the matches using your drawing function `drawMatches` and output the result
- Test your implementation using the provided datasets. The first image pair (index 1) in the bag files always contains the same image in both the source and destination image messages for debugging purposes. Your implementation should therefore extract the exact same keypoints in this image pair and match the exact same key point locations in both images with zero distance between matching descriptors.

4. *(Robust matching)* One key weakness of the matching function implemented in the previous task is that it always returns a match for each descriptor in the source array. One simple way to improve this is to compute a match confidence for each source descriptor by comparing the distance to the nearest neighbor d_{1-NN} with the distance to the second nearest neighbor d_{2-NN} in the destination descriptor array and only accepting matches below a certain ratio 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 implementation to only accept matches below a ratio threshold t_m. A good default value for t_m is 0.75.
   - Add a dynamically reconfigurable parameter to your matching node: `ratio_threshold` the ratio threshold for accepting matches.

5. *(Multi-Scale Patches Feature Descriptor)* One thing you will quickly notice is that the simple patch based descriptor is not particularly robust against different views of the scene. To remedy this you will be adding different invariances to your MOPS feature descriptor. To start with you will add scale invariance to your MOPS descriptor.

   - Build an image pyramid with the same number of octaves as your key point detection.
   - Extract the descriptor at the image octave indicated the key point's **octave**.

6. *(Multi-Scale Oriented Patches Feature Descriptor)* Using the dominant orientation estimated during key point detection it is also possible to make the MOPS descriptor robust against in place rotations of the scene. Instead of sampling along the grid aligned with image axes, the image patch coordinates are first rotated by the key point's orientation before sampling:

$$p' = Rp + k, \qquad R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

where p' are the oriented patch coordinates, p the original orthogonal coordinates, k the key point location, and θ the orientation angle.

   - Add a dynamically reconfigurable parameter to your feature descriptor node, **orientation**, that determines if oriented descriptors should be extracted.
   - Modify your implementation to optionally extract oriented image patches. Use the formula above to obtain the rotated patch coordinates. Initially perform nearest neighbor interpolation, so that $d_i = I(\lfloor x_i + 0.5 \rfloor, \lfloor y_i + 0.5 \rfloor)$.
   - **optional**: Extend your implementation to use linear interpolation when sampling the rotated patch coordinates.

7. *(Normalization)* To compensate for changes to the scenes brightness you should normalize your descriptor by demeaning its values. Modify your descriptor extractor implementation to normalize the descriptors as a post processing step.

Exercise 2 BRIEF - Binary Robust Independent Efficient Features Descriptor

So far this assignment has dealt with extracting and matching descriptors that are based upon densely or uniformly sampling image values around the key point. An alternative both in terms of implementation speed and discriminability are the class of binary descriptors that are based upon statistical properties of image region around the key point. In this exercise we will implement the approach described in "BRIEF: Binary Robust Independent Features" by Calonder et al. (see http://cvlab.epfl.ch/research/detect/brief).

1. *(Binary Feature Descriptor Extraction)* The general idea behind BRIEF is to compare points p_1, p_2 selected at random in the region around the key point with a test $\tau(p_1, p_2)$ defined as:

$$\tau(p_1, p_2) := \begin{cases} 1, & \text{if } I(p_1) < I(p_2) \\ 0, & \text{otherwise} \end{cases}$$

The BRIEF descriptor d_n is then defined as the n-dimensional bit string, for $n \in \{128, 256, 512\}$, resulting from the ordered aggregation of n such tests $\tau(p_1, p_2)$:

$$d_n := \sum_{i=1}^{n} 2^{i-1} \tau(p_1^i, p_2^i)$$

The resulting descriptor is then 16, 32 or 64 bytes respectively, hence the name BRIEF-16, BRIEF-32 and BRIEF-64. Before the tests are performed the image patch is usually smoothed to robustly against noise.

- Add a new class for extracting BRIEF descriptors. Add a dynamically reconfigurable parameter to your feature descriptor node for optionally extraction BRIEF descriptors and selecting between 16, 32 and 64 byte descriptors.
- The first step in implementing your BRIEF descriptor is to define the n tests and their order. To simply things we have provided you with a sample test pattern in *BRIEFPattern.h* contained in the `rvc_utils` package suitable for all BRIEF pattern sizes, but feel free to try out your own pattern.
- Perform smoothing using a gaussian or box filter prior to computing the descriptor. Your smoothing parameter should be dynamically reconfigurable. A good default here is a filter size of 3x3.

2. *(Binary Descriptor Matching)*
   - Modify your descriptor matching implementation so that the hamming distance is calculated for BRIEF descriptors instead of the ℓ^1 or ℓ^2 norm. The hamming distance for bit strings is defined as $\sum_{i=1}^{n} s_i \oplus d_i$ where s_i, d_i are corresponding source and destination bit values and $\oplus$ the binary xor operation and in short is the number positions where two bit strings differ.

Exercise 3 ORB - Oriented Brief Descriptor

The vanilla BRIEF descriptor is not invariant to in-plane rotations. Simply rotating the set of randomly selected test points by the key point's domination orientation has been shown to lead to a net decrease in descriptor performance. An improvement in descriptor performance has been demonstrated with a specific set of test points that are only available at 30 evenly distributed orientations (every 12°) instead of the key point's exact orientation. A fixed descriptor length of 32 bytes was also chosen. This idea was proposed in "ORB: an efficient

alternative to SIFT or SURF" by Rublee et al. (see http://www.willowgarage.com/papers/orb-efficient-alternative-sift-or-surf).

1. *(ORB Descriptor Extraction and Matching)*
   - Add a new class for extracting ORB descriptors and modify your feature descriptor node to also optionally extract ORB descriptors at run time.
   - The ORB tests are not random in contrast with BRIEF so we have provided you with a file containing the ORB test patterns *ORBPattern.h* in the `rvc_utils` package. Choose the test pattern orientation closest to the key point's orientation.
   - Just as with BRIEF use the hamming distance metric for matching ORB descriptors.

Exercise 4 ROC Challenge

In order to evaluate the performance of feature descriptors, ROC (Receiver operating characteristics) curves are often used to compare different descriptor implementations. These curves show the True Positive Rate (TPR) against the False Positive Rate (FPR) evaluated for certain descriptor distance thresholds.
Inside the `rvc_utils` repository, you find a node which will publish an image including these curves, once you've provided your results. The node subscribes to the topic `~roc_result`. Publish a `ROCCurveMessage` as it's defined in the `rvc_utils` repository (inside the `msg` folder).
The message contains the following fields which you should fill out and publish:

- `team_name`: you should put the name of your team here
- `dataset_name`: you should be the one from the `DataSetMessage`
- `dataset_idx`: the index of the current image inside that dataset (also from `DataSetMessage`)
- `source_points`: a vector containing the 2d homogeneous (z coord set to 1) feature point locations in the source (dataset index 1) image
- `dest_points`: a vector containing the 2d homogeneous (z coord set to 1) feature point locations in the destination (current) image
- `distances`: a vector containing your calculated descriptor distances, corresponding to the matches in the above vectors

The `roc_challenge_server` node will publish result images for the 3 datasets. Once it receives a result message on the `roc_result` topic, it will update the information for the corresponding team and dataset and republish the image with the updated curve. Note, that the ROC curve is always for one complete dataset, as we calculate the combined ROC-curve for all the matches.

You can run this node locally on your machine for testing and improving your matching performance. Experiment with your matching pipeline implementation to determine the techniques that bring the best results for the datasets. In two weeks, we'll run the ROC server on a single machine and compare the results of the individual teams.

→ LET THE GAMES BEGIN!!