# Robot Vision Course WS 2013 / 2014

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

**Assignment 1 - Due: 30.10.2013**

## Exercise 0 Getting to know ROS

1. *(ROS Tutorials)* The best way for getting started with ROS, is using the Tutorials provided on http://wiki.ros.org/ROS/Tutorials
   - Walk through all the Tutorials on this page, to get an overview about the ROS system. As C++ will be the language of choice throughout this course, please stick to those tutorials where appropriate.
   - As of ROS Groovy there are currently two separate build systems: `rosbuild`, the legacy build system, and `catkin` the new default build system. Since we are targetting ROS Hydro in this course, we will be using the `catkin` build system, so where appropriate stick to the tutorials for the `catkin` build system.
   - Upon completion you should be comfortable with building ROS packages, running ROS nodes and publishing / subscribing to messages.
2. *(ROS Visualization Tools)* It is important that you make yourself familiar with the core ROS visualisation tools: `rqt` and `rviz`. You will be using both of these tools extensively throughout the course, so be sure to review the user documentation at http://wiki.ros.org/rqt and http://wiki.ros.org/rviz/UserGuide, in particular how to visualise image data in each tool.
3. *(ROS Data Serialization)* This topic is important enough to be handled separately, in ROS robot data of all kinds is saved, replayed and exchanged using `rosbag` files, see http://wiki.ros.org/rosbag. We will be making extensive use of bag files in the first half of this course. Grab a bag file from http://www6.in.tum.de/~jensen/rvc/corner_data/, use the `rosbag` tool to replay the data and visualise the camera data using `rviz` and `rqt`.

## Exercise 1 Harris Corner Detector

In this exercise, you will implement the first fully functional ROS node in the course: a Harris corner detector for key point extraction. This node will subscribe to an image topic, compute the Harris corner response for each valid point in the image, extract key points from the response values, and finally publish to an image topic with the key points drawn on the original image.

1. *(Package Configuration)* Create a package named {*group name*}_features using the `catkin` helper script `catkin_create_pkg` in your catkin workspace.
   - The package should have dependencies on: `cv_bridge`, `image_transport`, `dynamic_reconfigure`, `opencv` and `roscpp`. Check the generated `package.xml` to make sure all dependencies are included correctly.

- In this package `src` create a file named `FeatureDetectorNode.hpp`. This file will contain the node code for subscribing to and publishing images, as well as coordinating the Harris corner detector.
- Create another file named `feature_detector_main.cpp`. This file will contain the main entry point and run loop of the node.
- Create a third file `HarrisDetector.hpp`. This file will contain the actual Harris corner detector implementation.
- Modify the package's `CMakeLists.txt` file so that an executable node named `feature_detector` from the previously described three files is generated. Follow the `catkin` CMake conventions, see http://wiki.ros.org/catkin/CMakeLists.txt. For OpenCV you will need to manually set the appropriate settings since it is not a `catkin` package:
  - Get the OpenCV settings with `find_package(OpenCV REQUIRED)`.
  - Then add the headers to search paths with `include_directories(SYSTEM ${OpenCV_INCLUDE_DIRS})`
  - Finally make sure to link your target with the OpenCV libraries `target_link_libraries(feature_detector ${OpenCV_LIBRARIES})`

2. *(ROS Node)* The first part of your Harris corner detector implementation involves developing some ROS infrastructure code for receiving and publishing image messages. Here we will make use of functionality present in the `image_transport` and `cv_bridge` packages for handling image messages instead creating and processing the raw image messages manually, see http://wiki.ros.org/image_transport and http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages.
   - Create a class `FeatureDetectorNode` in the file `FeatureDetectorNode.cpp`. This class should have the following members:
     - A member of type `image_transport::ImageTransport` for creating image message publishers and subscribers.
     - A member of type `image_transport::Subscriber` for subscribing to incoming image messages.
     - A member of type `image_transport::Publisher` for publishing outgoing image messages.
   - Create a member function with the following signature:
     `void imageMessageCallback(const sensor_msgs::ImageConstPtr& msg)`.
     This function will initiate Harris corner detection on the incoming image, create a duplicate of the image, draw the detected key points on the duplicate, and then publish the resulting image.
   - The `FeatureDetectorNode` class constructor should take a `ros::NodeHandle` reference and initialize all the `image_transport` members.
     - The `Publisher` should publish on a topic named **image**
     - The `Subscriber` should subscribe to a topic named **in**. It should register a call back to the member function `imageMessageCallback` previously defined.
   - Use the `cv_bridge` functionality in the `imageMessageCallback` member function for creating a `cv::Mat` copy of the image message.
     - The function `cv_bridge::toCvCopy` should be used to get a copy of the image message data.
     - The returned instance of type `cv_bridge::CvImagePtr` contains a member `image` that contains the `cv::Mat` data that you can pass on to OpenCV functions. Use the member function `toImageMsg()` for publishing images messages containing the image data.

- In the file `feature_detector_main.cpp` create a simple ROS main function. Here you should instantiate the `FeatureDetectorNode` class and run the ROS message loop indefinitely, as is typical in the ROS tutorials.
- To verify that your `ImagePublisher` and `ImageSubscriber` implementation is working correctly you should modify the `imageMessageCallback` function to use OpenCV to draw a giant circle on the incoming image message and publish the resulting image.

3. *(Harris corner detection)* Recall that the Harris corner detector looks for points in the image where the immediate area enclosing the point has high self dissimilarity in all directions. This can be approximately estimated using image partial derivatives and the *Sum of Squared Differences* method:

$$S(x, y; \Delta x, \Delta y) = [\Delta x \Delta y]\, Q(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

where $S(x, y; \Delta x, \Delta y)$ is the approximated sum of squared differences between the image patch at $(x, y)$ and at $(x + \Delta x, y + \Delta y)$ and where

$$Q(x, y) = \sum_{(u,v) \in W(x,y)} w(u, v) \begin{bmatrix} I_x(u, v)^2 & I_x(u, v) I_y(u, v) \\ I_x(u, v) I_y(x, y) & I_y(u, v)^2 \end{bmatrix} = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$$

is the Harris matrix at $(x, y)$. This matrix contains a sum of the squared image partial derivates over the template window $W(x, y)$ centered at the image location, where each partial derivative term is weighted according to the window weight $w(u, v)$ (which can either be a guassian or a constant factor). Harris corners are characterized by locations which have two large eigenvalues $\lambda_1 \lambda_2$ in the Harris matrix. Instead of performing eigen value decomposition we will use the alternative formulation for checking this trait proposed by Harris:

$$H(x, y) = \lambda_1 \lambda_2 - 0.04(\lambda_1 - \lambda_2)^2 = AC - B^2 - 0.04(A + C)^2 \tag{1}$$

which is known as the Harris response for a point at $(x, y)$.
In this exercise your Harris corner detector will implement the OpenCV feature detector interface as specified by `cv::FeatureDetector` so your implementation can easily be compared with other feature detector implementations present in OpenCV, see http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_feature_detectors.html.

- In the file `HarrisDetector.hpp` create a class `HarrisDetector` that derives from `cv::FeatureDetector`.
- The class should initially contain member variables for storing the **window size** and **threshold** parameters of the Harris corner detection algorithm.
- Create a protected member function for computing the Harris corner response extracting key points. The function should have the following signature:
  `void harrisCorners(const Mat& image, vector<KeyPoint>& keyPoints)`
  where `image` is an input grayscale image and `keyPoints` an output parameter.
  - In this function calculate the matrices $A$, $B$, and $C$ for each valid point:

$$A(x, y) = \sum_W I_x(x, y)^2, \quad B(x, y) = \sum_W I_x(x, y) I_y(x, y), \quad C(x, y) = \sum_W I_y(x, y)^2$$

  where $W$ is the weighted window sum as determined by the **window size** parameter, see http://docs.opencv.org/modules/imgproc/doc/filtering.html#boxfilter and http://docs.opencv.org/modules/imgproc/doc/filtering.html#gaussianblur.

  - Calculate the Harris response at each valid point using equation 1. Determine the maximum Harris response value $H_{max}$.
  - For each point $(x, y)$ where the Harris response $H(x, y)$ is above **threshold**$*H_{max}$, add a new `cv::KeyPoint` instance to the `keyPoints` output parameter. In the case of a very low threshold, it is generally a good idea to only consider points whose Harris response exceeds a certain absolute minimum value, such as 10.0.

- Implement the required member function from `cv::FeatureDetector`: `virtual void detectImpl(const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask=Mat())`. This function should initially just call the protected member function `harrisCorners`.
- Create a function `drawKeyPoints` that takes as input an OpenCV image and an array of key points and draws a circle at each key points location on the image.
- Modify the `FeatureDetectorNode` class by adding a member variable of type `boost::shared_ptr<cv::FeatureDetector>`. In the constructor initialise this member with a pointer an instance of your `HarrisDetector` class.
- Modify the `imageMessageCallback` member function so that it extracts Harris key points using the newly added `cv::FeatureDetector` pointer member variable. Before performing feature extraction the image should be converted to **gayscale**. It should then use the drawing function to draw the detected features on the output image.

*You may **NOT** use any of the OpenCV corner detection functions! However, you may use the Sobel, Gaussian or Convolution filters as well as any of the drawing functions.*
Test your implementation using the bag files listed in Exercise 0.

4. *(Dynamic Reconfigure)* Now its time to get to know a very useful piece of ROS infrastructure: `dynamic reconfigure`, see http://wiki.ros.org/dynamic_reconfigure/Tutorials. Here you are going to make the your `FeatureDetectorNode` and with it your `HarrisDetector` respond to parameter changes at run time.

   - Create a new directory named `cfg` in the your package. Create a new dynamic reconfigure file in that directory named `FeatureDetectorConfig.cfg` and make this file executable. Add an integer parameter **window_size** and a floating point parameter **threshold**, the parameters used by the `HarrisDetector` class.
   - Make the appropriate changes to the your package's `CMakeLists.txt` file for dynamic reconfigure as indicated by the tutorials.
   - Modify the `FeatureDetectorNode` class by adding a new member variable of type `dynamic_reconfigure::Server` class. Add a new member function `updateConfig` for receiving callbacks of type `FeatureDetectorConfig` from the dynamic reconfigure server. In this function you should create a new instance of your `HarrisDetector` class with the updated parameters and then assign this pointer to your `boost::shared_ptr<cv::FeatureDetector>` member.

   Test your implementation using the `rqt_reconfigure` tool.

## Exercise 2 Improved Harris Key Point Detection

1. *(Fixed number of points)* Up until now you have implemented a Harris key point detector that outputs a variable amount of points depending on the strength of the Harris response function $H(x, y)$ and the **threshold** parameter. For many robotics use cases, such as visual odometry and localization, it is preferable for the key point detector to always return a fixed number of key points.

- Add a member variable for storing the **number of points** to your `HarrisDetector` class. Add a **flag** for determining whether to perform thresholding or return a fixed number of key points.
- Implement a member function in `HarrisDetector` that takes an array of key points as input and returns a second list of key points of size **number of points** that contains the strongest Harris key points.
- Move the thresholding filtering to a separate member function.
- In `detectImpl` perform either thresholding or return a fixed number of points depending on the **flag**.
- Make the parameters modifiable at runtime using `dynamic reconfigure` by extending your existing configuration file.

2. *(Multi Scale Detection)* To make the feature detector more robust against camera movement and the resulting scale changes in the scene you will implement a simple form of key point scale estimation using image pyramids.
   - Add a parameter to your `HarrisDetector` class: **octaves**, the number of image octaves to use for feature extraction.
   - Extend your `detectImpl` implementation to create an image pyramid using the OpenCV function `cv::pyrDown` that has the appropriate number **octaves**. For each octave in the pyramid call your protected member function **harrisCorners** with the scaled down image.
   - Add a parameter to your `harrisCorners` member function indicating which octave is being processed. Make sure to include this information in your key point generation.
   - Once all octaves have been processed, perform filtering on the key points from all the octaves together.
   - Extend your key point drawing function to draw key points according to their octave.
   - Make the octave parameter dynamically reconfigurable

3. *(Adaptive Non Maximal Suppression)* Another method for improving the robustness of the feature detector 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:

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

   where $H(p)$ is the Harris response at a point $p = (x, y)$ 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 `HarrisDetector` implementation by adding an additional filtering member function for adaptive non maximal suppression similar to the thresholding and fixed number of points filters.
   - Your ANMS implementation should start by sorting the key point candidates array according to their response strength.

- The point with the strongest Harris response is always in the output array. 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 to a point with a significantly higher response should be saved.
- After processing each point return the points with the largest minimum distance.
- Your implementation should support two dynamically reconfigurable parameters:
  - `ANMS`: a boolean parameter determining whether ANMS is used.
  - `ANMS_points`: The number of key points your ANMS function should output.

4. *(Orientation Estimation)* Aside from location and scale information most robotics applications involving key points require estimation of the relative orientation with respect to the scene.

   - Extend your `HarrisDetector` 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 $\nabla I(x, y)$ where:

   $$\theta = \arctan\left(\frac{\nabla_x I(x, y)}{\nabla_y I(x, y)}\right)$$

   To improve the robustness of the orientation estimation it may be necessary to use a finite difference operator $\nabla$ with a larger kernel size. For improved accuracy you should use the `atan2` function.

## Exercise 3 OpenCV Feature Detectors

1. *(Comparison)* Extend your `FeatureDetectorNode` to support other OpenCV feature detectors that implement the `cv::FeatureDetector` interface, for example the `GoodFeaturesToTrackDetector` that also uses Harris corner detection. How does your HarrisDetector compare?