Final report

Project #18
Programming a robotic manipulator for a computer vision application

Date: 19.12.2016

Niemelä Matias
Pakola Leo
Seppälä Mikko
Tiira Tuomas

Information page
Students
Niemelä Matias
Pakola Leo
Seppälä Mikko
Tiira Tuomas

Project manager
Niemelä Matias

Official instructor
Hazara Murtaza

Other advisors
Kyrki Ville

Starting date
10.1.2016

Approval
The instructor has accepted the final version of this document
Date: 19.12.2016
1. Objective

In this project, we were dealing with a KUKA LWR4+ robot (shown in figure 1) that is located in the intelligent robotics lab at Aalto University. KUKA LWR4+ is a lightweight robot arm that can be used in a wide range of different tasks that require precise movement and positioning with force control. For example, this kind of tasks could be laser welding or assembling complex objects. To achieve such accuracy, we need to have appropriate controllers that require a lot of data from different sensors inside the robot.

The system uses Orocos (Open Robot Control Software Project) and ROS (Robot Operating system) in its controller implementations. Both Orocos and ROS are software packages on Linux, and their essential contents are C++ libraries and tools for creating applications for robots. The interface where the controllers were initially implemented was Ubuntu 12.04.

Originally, the objective of the project was to update the external robot control system to Ubuntu 14.04, update the Robot Operating System (ROS), implement controllers for real-time and non-real-time systems and compare their performances on the KUKA LWR4+ robot. An additional goal of the project (if time would allow) was to provide a demo with the new updated system. However, halfway through the project we decided only to implement the real-time controller for the updated system and to focus on providing a proper demo application. The idea was to demonstrate that the updated system is working properly by transforming a picture of a person's face into robot arm movement trajectories.

![Figure 1. KUKA LWR4+ robot arm](image-url)
2. Updating the system

The system update has three main parts: Installing needed hardware, installing the software and testing the new system. There was no hardware budget as only the storage disk given for the new system was installed. Installing software took most of the time as the documentation on the existing system was imperfect, which caused many trial and error situations. Testing the new system consisted of controlling the KUKA arm in monitor and command modes from the external computer with the new software.

2.1. Installing hardware

As the old system was still in active use, a new drive was installed to the control computer. The drive in question was a 2.5 inch Samsung solid state drive. 2.5 inch to 3.5 inch and 3.5 inch to 5.25 inch adapters were needed to install the drive as the case of the computer had only one 2.5 inch slot and it was used by the SSD of the old system. 3.5 inch bays were also occupied by the quick install dock, which the 2.5 inch drive with adapter was not compatible with. The installation position was chosen to be under the optical drives. This caused some problems as only one side of the drive bay had screw holes for mounting and the other side had bent metal to act as a spring.

Custom aluminum washers were made from scrap sheet for the screws as some were too long for the adapters and pushed into the SSD. One of the connectors for the 3.5 inch quick install dock was disconnected and replaced by normal SATA cable for the Samsung SSD. A power connector reserved for the 5.25 inch bay was used to power the drive.

2.2. Installing software

The software installation and update took most of the time on the system update. For clarity the order is not the same as it was done as there were complications during the update. Some major errors made during the upgrade, such as trying to use the packages from Ubuntu and trying to get them works. In the end some parts needed to be built from source with specific options, but later they were scripted to ease the process. The system was reinstalled multiple times so the scripts were tested and improved each occasion. During the project the original goal of ROS Jade was changed out for ROS Indigo for longer support period. The original goal of latest features was changed for better support as knowledge of ROS and the system came clearer. Detailed and updated information on installing the system with step-by-step guide can be found in Aalto University's Wiki pages [1].
Installing Ubuntu 14.04 LTS

After installing the SSD storage for the system, the Ubuntu 14.04 LTS was installed from an USB drive with default options aside from Finnish keyboard and United Kingdom locale to keep the system in English. Network configuration and kernel module information were extracted from the old system to ease the install.

Installation of the base system was successful and network connection worked with the old IP address set to static form network options. Non-real-time Ethernet connection to the ATI Mini 45 sensor was also configured and tested. Additional packages were installed, most notably xubuntu-desktop and eclipse. The default desktop was changed to XFCE used by Xubuntu.

Kernel with Xenomai

The old system was built on top of real-time kernel, Xenomai, for control and real-time network drivers, RTNet, for communication. This needed to be replicated for the new system with any possible updates.

After some trials it was found that it was impossible to update the kernel to newer version as was suspected in the risk analysis of our project plan. The used version, 3.5.7, is the newest one supporting the RTNet 0.9.13 driver stack released in 2012. This RTNet version only specified support up to kernels 3.2 but it still works on the 3.5.7.

The building of the kernel was compressed into single shell script that fetches the source files, extracts them, applies kernel patches, copies the preconfigured setup to the build and brings up the kernel menuconfig for user to check the necessary drivers. The premade configuration disables features harmful for the real-time framework, Xenomai, such as CPU frequency scaling and sleep. This configuration file is provided separate from the build script.

After accepting the configuration, the script starts building the new kernel with current date as postfix in the kernel name. The script does not check the current GCC compiler version, but it was found out that the RTNet drivers will not work with the default GCC 4.8 shipped with Ubuntu 14.04. It needed to be downgraded to version 4.4. The kernel would compile with version 4.8, but loading the real-time network modules resulted in either the network not working or freezing the system. The script only builds the kernel packages to be installed but it does not install them. First the Xenomai user space needs to be built.
Xenomai userspace

The kernel build process has already downloaded the needed sources. But as the tools will be built as debian packages some configuration was needed. The debian/rules file under the source root needs to be modified to include “--enable-dlopen-skins” in the CONFIG_OPTS. Unless this is done, the real-time kernel cannot load needed libraries for operation with the KUKA environment. In xenomai3 this has been renamed to “--enable-dlopen-libs” to clarify it.

The file debian/changelog also needs to be checked. It is used by the debian packaging process to determine the version to build. The topmost version should be version being built and the status should not read “UNRELEASED” but “unstable” is acceptable. After the needed modifications the packages can be built with “dpkg-buildpackage -us -uc” command.

Installing Xenomai

Now if the instructions from the Intelligent Robotics Wiki have been followed both the Xenomai kernel and userspace packages can be found in the parent directory. All the packages can be installed with following command:

```
sudo dpkg -i *.deb
```

This will install the packages built in earlier steps, the kernel package will also create the init image containing needed modules to be loaded during boot process. A usergroup for xenomai is also created during the install and it needs to be configured to work with the system. The group ID number can be acquired with:

```
cat /etc/group | sed -nr "s/xenomai:.:([0-9]+):.*/\1/p"
```

This GID needs to be but into the kernel boot parameters. This can be done by modifying the grub, default Ubuntu bootloader, configuration file /etc/default/grub.

The option “GRUB_CMDLINE_LINUX_DEFAULT” needs to include a parameter for the xenomai:

```
xeno_nucleus.xenomai_gid=GI
```

After saving the file grub needs to be updated with “update-grub” command. This creates the /boot/grub/grub.cfg file. By opening this file, the name of the xenomai kernel can be found. This can be copied to /etc/default/grub to be the default option. For example:

```
GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 3.18.20-xenomai-2.6.5-15072016"
```
After modifying the file, “update-grub” needs to be ran for the changes to take effect. The current user also needs to be added to the group with command:

```
gpasswd -a <login_name> xenomai
```

Ubuntu has some xenomai packages in the repositories and they might overwrite the custom built ones. Therefore, the packages installed need to be marked to be on old with `apt-mark` commands.

**RTNet**

Xenomai2 does not include RTNet drivers so they need to be installed after the kernel. The system needs to be booted into the Xenomai kernel. RTNet version 0.9.13 needs to be extracted to `/usr/src`. A configuration file with proper drivers for the control computer has been provided and it can be loaded with “make menuconfig” and selecting the file. After the drivers have been configured, they can be compiled and installed with “make install”.

Scripts from the old system were used for bringing up the real-time network with minor modifications. The Ubuntu network manager was disabled for the ATI Mini 45 and the KUKA and the script takes care of assigning network configuration. A call to stop the real-time network was added to `/etc/rc.local` to bring up the normal network during boot process.

**PEAK PCAN-PCI Express**

The pcan interface used to communicate with the Barrett hand can be installed from peak-systems. The binaries are installed into `/usr/local/bin` so it may be added to the `PATH` environment variable.

To use the correct drivers, following lines can be added to `/etc/rc.local` to be executed at boot:

```
rmmod xeno_peak_pcan xeno_can_peak_pci xeno_can_sja1000 xeno_can || true
rmmod pcan || true modprobe pcan || true
```

**Libbarrett**

The library for controlling for the Barrett hand can be installed by the instructions given in libbarrett documentation. `/usr/local/bin` may be added to the PATH environment variable and `/usr/local/lib` may be added to `/etc/ld.so.conf.d/` for the commands and the libraries to be found by the linker. To refresh the linker cache after adding the path, run “ldconfig”.

Page 7 of 19
ROS Indigo

Indigo is one of the supported ROS distributions on Ubuntu 14.04 so installing it can be taken straight from the roswiki. It is recommended to install the package `ros-indigo-desktop-full`. After ROS is installed, a supplied script should be used to install all necessary dependency packages for OROCOS and the KUKA development environment. After this some of the ROS tools should be initialized by running following commands:

```
gem install rake hoe
rosdep init
rosdep update
```

OROCOS

Ubuntu provides ready OROCOS packages, but they are compiled wrong. The target for those packages is gnulinux instead of xenomai so they cannot run real-time operations. In the project plan’s risk analysis this problem was considered. The packages also miss some transport layers used by the KUKA environment so the used software will not compile with it. Roswiki has instructions for installing OROCOS, but they were found out to be wrong. Instead of the recommended version 2.7 the next version 2.8 should be used for messages to work [2, 3]. 2.8 has also fixed a problem with catkin, the ROS build tool, regarding integration with OROCOS [3].

A script was written to automate the installation of OROCOS and the ROS integration packages for OROCOS. The script includes all needed build options and switches. The script also removes `rtt_dynamic_reconfigure` which does not compile, but it is not used by any package and the system works without it. GCC version should still be 4.4 from compiling the kernel, the script does not check it. Sources are downloaded from GitHub, but it was found out that sometimes the download would fail and skip some needed files causing the build process to fail. Removing the source files and rerunning the script resolves the issue. The script prints out info which should be added to the shell init script, by default `~/.bashrc`, to include the OROCOS environment. Checking if the installation was successful, running the following command should give a list of rtt integration packages:

```
rospack list |grep rt
```

KUKA Environment

The packages used to control the robot arm make use of more recent C standard, C++11, so GCC should be upgraded back to version 4.8 as version 4.4 does not support it. The old system used rosbuild with ROS Groovy, but in Indigo rosbuild has no longer proper support. Rosbuild has been replaced by catkin. As suspected in the project plan’s risk analysis,
converting the packages to the new build system was difficult as understanding the system took some time. A short guide [4] was written for converting the packages.

In the old system using rosbuild meant that every package was built manually with CMake, paths to the packages needed to be added manually to the environment variables for ROS and OROCOS to find them. The source files were also mixed with compiled content and manually generated message typekits. Catkin automates all this and the source files are kept separate. Catkin has workspaces for each source repository, from the source a separate development and install environments are built automatically. Loading the workspace sets the environment variables for ROS and OROCOS to find the needed packages.

Most of the work was fixing the dependencies between the packages and the system. Linker errors were common and the CMake macros had changed from rosbuild. A set of tools called catkinize was used to convert the packages, but the tool was not perfect so some problem solving was needed. To solve the linker errors a small script was written to find function references inside shared libraries so the needed package could be added as dependency in cmake files. The build system was also automated to generate the OROCOS message typekits instead of the need to manually build them for communication between ROS and OROCOS to work. The workspace parameters and libraries were also exported to catkin environment so OROCOS components could work across different workspaces. This took some experimentation to work without issues and there are still some manually set paths, but they are referenced to the workspace environment path so they work. Only KUKACommander needed the manual paths as there is something wrong with the source files. Some source file modifications were also needed, mainly missing include guards in header files, but some hard-coded paths were also found that needed to be fixed to work with catkin environment variables. Renaming the workspace no longer breaks the whole system.

The system is printing warning as the naming scheme of the KUKA environment does not comply with the catking naming convention. Capital letters in package names are not recommended in catkin and KUKACommander and FTSensor packages break this. The errors are not fatal, just warnings and no switch to turn this behavior off has been found. The messages are printed to stderr and this was found to cause problems with boost.process, which was used in some applications to launch commands. Boost.process exists if it sees something in stderr. Workaround for this is to include “2>/dev/null” to ignore errors or “2>&1” to redirect the stderr to standard output in the command to call with boost.process.
OROCOS behavior has changed with the upgrade from version 2.6 to 2.8. The dependencies between components do not always load and this seems to only happen with ROS message types used by OROCOS. Before the message types did not need to be imported in the deployment, but with the new system, some lines needed to be added to the component loading phase to import all needed components. Streaming messages from OROCOS to ROS would otherwise give unknown transport type errors. Pure OROCOS components with no ROS interaction did not need any modifications for deployment.

2.3. Testing the new system

The robot arm can work in two main modes with FRI. Monitor mode and command mode. In both modes the data from the robot can be read. In monitor mode simple PTP and LIN commands can be executed like they were written on the robot controller. KUKACommander can send these PTP and LIN commands and the system was tested by sending joint angles to each joint and the speed as a PTP command. After sending the commands the robot moved to the requested angles. Communication between the control computer and the robot was therefore working. LIN command was also tested but it was found out that while the robot moved, the speed could not be controlled. The fault was found to be on the FRI program running on the robot controller. A fix was found for this and example FRI program can be found at the robot controller in the Programs/PbD/ directory with “linspeed” in the name.

In command mode the robot has multiple control modes: joint angle, joint impedance and Cartesian impedance. The communication between the FRI running on the robot controller and the control computer needs to be synchronized for the robot to move. The control computer needs to send message each cycle the FRI on the robot controller runs. This was also tested with KUKACommander. It has two gravity compensation modes of the two impedance control modes. It sets the robot stiffness to zero and commands the current position from the robot pack into the commanded position. These were tested and found to be working, the robot could be moved by hand without errors. Later a test with other converted packages from Learning from Demonstration was used to plane wood with the project instructor and this was also found to be working after some corrections to the code.
2.4. Important notes

Ubuntu
System is safe to update. All the packages that might break are marked to be on hold, mostly xenomai packages, and system is set to boot automatically to xenomai kernel. If the updater asks to overwrite files, keep the old as some files were modified manually. The system is using nouveau drivers, nvidia proprietary drivers will not work. External graphics card is needed as the most common intel graphics driver is not supported by the xenomai kernel with Ubuntu 14.04. The driver is too old to work properly with the intel libraries. USB3 is working, but the xenomai kernel and libusb cause some issues so Kinect 2 will not work on the computer. A workaround is to use another computer with Kinect 2 and connect it via network.

Kernel and Xenomai
The source files for the kernel or xenomai cannot be removed or moved. The Makefile creates symbolic link to source files of copying them with install command like most Makefiles. Moving the source files causes the symbolic links to header files to break and compiling applications using xenomai to become impossible due to missing header files.

The used kernel, 3.5.7, does not support Intel graphics drivers on Ubuntu 14.04. It also does not have new enough USB3 support to work with Kinect 2. Newer kernel with xenomai3 could be used as xenomai3 has merged and updated RTNet drivers. Xenomai3 is however not supported by any current OROCOS version so controlling the robot is not possible. Patches include 4.x series kernel support but they did not compile.

RTNet
Xenomai3 includes updated RTNet drivers and they were tested to be working when trying to find a working configuration. This meant a newer kernel with proper driver support for Ubuntu 14.04. Only OROCOS support was missing.

FRI
Old documentation from Intelligent Robotics wiki suggests that the real-time control can operate with hard real-time up to 1 kHz. During the testing and reading the code it was found out that the FRIServerRT running on the external control computer could be commanded to run that fast, but in reality it would never run that fast. The update frequency comes from the FRI running on the robot controller. The communication is synchronized and the FRIServerRT will wait for communication and not run the update loops at the commanded
rate. The update rate can be set in the robot controller FRI program by adjusting the period in milliseconds.

The shortest period the robot worked reliably was three milliseconds, but that was with simple task of commanding the current position straight back. Any shorter period caused the robot to not accept command properly, although it sometimes moved. Therefore, a safe value of 4 milliseconds for the period should be stable as 3ms might be marginal. When observed for thousand samples, the standard deviation was 0.10ms while the real loop rate was 2.94ms when FRI period was set to 3ms. The hard real-time is never achieved from the perspective of the control computer. The communication runs as fast as the FRI on robot controller does. The FRIServerRT needs to be set to run faster than the FRI on the robot controller, otherwise the communication goes out of synchronization. The update rate of the FRIServerRT is safe to set to 1 kHz as the component waits for communication and runs at the same speed as the FRI on the robot controller.

### 3. Demo application

The functionality of the updated system was demonstrated with a demo application. The application acquires points of the facial features from the webcam images by using a computer vision program. The facial features were then transformed into points and then sent to KUKA. The robot then draws the detected face based on these points using a controller program written by us. Part demo application’s setup is shown in figure 2.

![Figure 2. Setup of the demo application](image)
3.1. Pen holder

A new pen holder was made to replace the old one that could only hold a marker. The new tool was made so more detailed features could be drawn. The new holder could house pens of different size and in different angles. In the end a marker was used as the features to draw did not need sharper pen. The pen holder was modeled and 3D-printed. Model of the pen holder is shown in figure 3.

![Figure 3. New pen holder.](image)

3.2. Facial recognition

For facial feature recognition we utilized an open source program called Attention Tracker [5] which is based on OpenCV computer vision library and Dlib machine learning algorithm library. The detection is based on the histogram of gradients (HOG) method and fitting a 68-point face model to the processed image. In figure 4 we can see a picture where the original image and the detected face pattern are combined.
The original program was developed to estimate the pose of a head so we had to modify the program code to extract and send the facial feature points to KUKA. After extracting the points of different facial parts, they were then divided in groups so that KUKA would know to lift the pen after drawing each part of the face. To transfer the points to the robot, we modified the program so that it writes the points to a text file that was then read by the robot controller program.

3.3. Demonstration controller

A demonstration controller was written for the robot. The robot was commanded in Cartesian impedance mode so X/Y coordinates from the picture could be translated to the drawing plane without complications.
The robot is taught three points so the origo and direction and length of each axis can be calculated. These are points in Cartesian space and they can be saved and resent to the controller if so desired. The controller does the calculations from the points, but does not do sanity checks such as if the robot collides on the origo. Teaching the points each time ensures no such condition should happen as paper is used as the template for corners. As the path to be drawn is extracted as series of X/Y coordinates, the control algorithm simply scales the given value on each axis to the desired position. The input is expected to be a floating point value between 0 and 100, a percentage of the axis. Figure 5 illustrates the implementation of the developed program for teaching the controller the shape and size of the drawing plane.

The drawing of the path is divided into separate steps. The robot starts to move the pen in air over the X/Y position of the first point. Next the pen is commanded downwards until the force exceeds set threshold for predefined time. As the control is in Cartesian impedance mode, the X and Y directions are set stiff, but the Z direction is left loose. In Cartesian
impedance the robot arm is driven with force control, the Z axis is commanded down until it hits the drawing surface, but the position is commanded under the drawing surface and the impedance controller exerts a force on the pen defined by the commanded distance and axis stiffness. After the pen has touched the surface with enough force the next point in X/Y plane is commanded while still commanding the Z axis to be under the surface to keep the force on the pen. During the drawing phase the X/Y coordinates are calculated for each point, but the distance to the point is checked so that it does not exceed the maximum speed given to the controller. The update rate of the controller is known so the maximum distance per update is known and the commanded distance is checked against it. The sum vector of the commanded position is divided into X/Y components so they can be limited if there is need to do so.

The completion of final points is not checked as the robot is not working in position control mode. If the commanded distance is small, the force control may not reach the target. Therefore, the point is assumed to be reached if the distance to be commanded does not need to be limited, the point is assumed to be reached in the last update. On the final point in the path the pen is brought up from the drawing plane before returning the pen to the starting position. The controller can only draw one line at the time so the routine is repeated for every line extracted from the OpenCV program. Figure 6 illustrates the implementation of the developed computer vision application and its result.

![Diagram](image_url)

**Figure 6.** Implementation of the developed computer vision application.
4. Project management

A relatively large amount of time was spent during the first month to produce a project plan regarding the expected output, timeline, milestones and work breakdown of the project. Unfortunately, this document ended up being less useful than we hoped. As explained in chapter 3, the system upgrade took significantly more time than predicted so the timelines and milestones regarding it were not reached. Additionally, as explained in chapter 1 the objective of the project was partly changed as we decided to drop the comparison between real-time and non-real-time controllers and to focus more on providing a demo application. Since the project was performed quite differently than originally planned, it seems that some agile method for project management with lighter planning documentation would have been a better fit for this project.

The workload of this project was quite hard to divide for four people since everything had to be completed in a certain order and often it was difficult to work with the tasks in parallel. Also we originally started with five team members but one member quit this project course after 3 months. Due to the nature of the project and busy schedules of the team members, we arranged face to face meetings for the whole team roughly once a month, only when absolute necessary. Especially towards the end of the project most of the required documentation work was done remotely using Google Drive. Most of the communication within the team was done via Telegram Messenger which proved to be sufficient enough. With the instructor we communicated face to face and via email when needed.

5. Conclusions and lessons learned

The external robot control system was successfully updated from Ubuntu 12.04 to Ubuntu 14.04 although it was time-consuming. In addition, ROS was updated and the previously used packages were converted to catkin build system. Trying to estimate the workload and dividing it into separate tasks for a system update in this scale was truly difficult and proved to be a bit useless. Numerous problems occurred which were difficult to predict without having plenty of experience from a similar task. As explained in chapter 2, finding a working configuration took time as documentation was lacking or erroneous and dependencies between different parts (Real-time kernel, network drivers, ROS and OROCOS) needed different workarounds. Rosbuild and catkin are quite different so converting components from rosbuid (ROS Groovy) to catkin (ROS Indigo) was rather difficult. Following the system update, the real-time controller was configured, implemented and tested.
In addition to the system update, a real-time OROCOS controller “KUKA_Drawdemo” designed for the demo application where the robot is able to draw the facial outlines and edge features of a given picture was successfully implemented. The idea of this demo was to show that the KUKA robot with its updated control system is working properly and provides good support for creating these kinds of applications. In this way, the research work conducted using the robot will be more straightforward in the future.
References


