ARPA: Autonomous Robotic Pointer Arm

Helgerud, Erlend Håland, André

April 4, 2018

Contents

0.1 System description

0.1.1 Architecture

The idea of ARPA is to have a 1:1 relationship between a physical robotic arm, and a simulated model in Gazebo. This means that the model in Gazebo should contain (as good as) the same measurements as the physical robot. They will also receive the same control signals leading to synchronized movement. Figure 1 show this relationship between the real world and the simulated world.

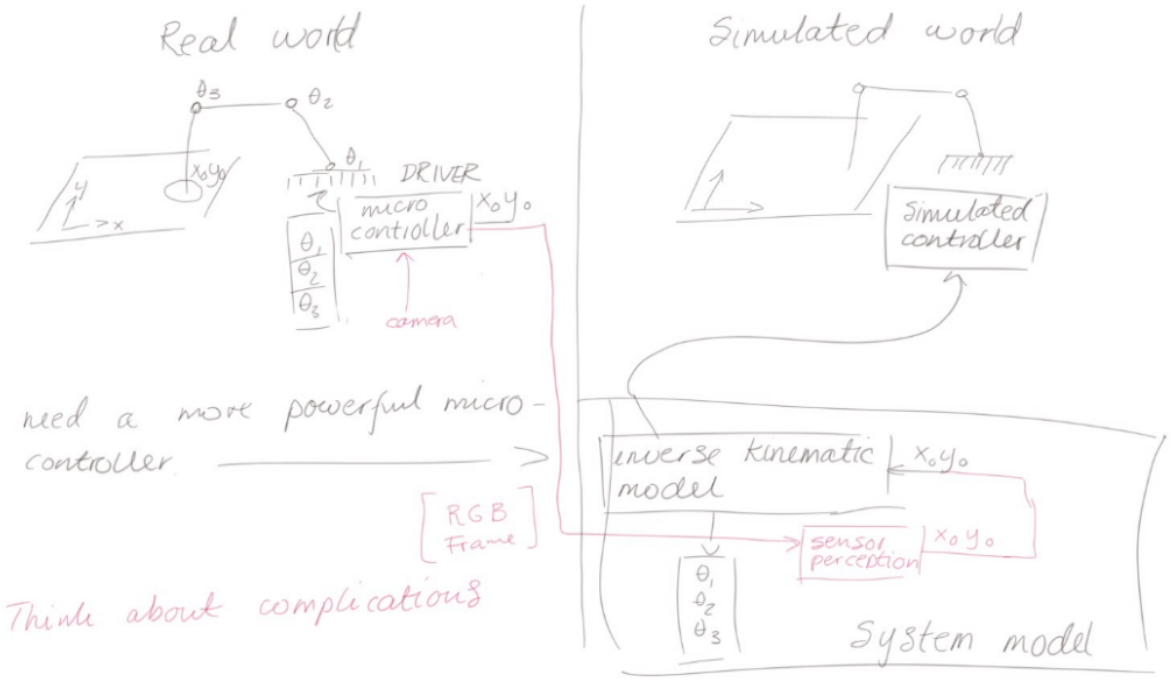

Figure 1: System architecture

ARPA is divided into nodes. These nodes each has its own tasks and communicate with each other on different topics to exchange information. Figure 2 shows the nodes of ARPA, and the topics they communicate on. A circle represents a node, whilst a rectangle represents a topic. The controller node is the center of all actions. It handles the interface between the camera, MATLAB, the physical robotic arm and the model in gazebo. The camera node sends a Cartesian point in space to the controller. The MATLAB node handles inverse kinematics. It receives Cartesian coordinates from the controller, and sends back five joint angles. The 5DOF Robot node controls the physical robotic arm by writing received angles to the servo motors. The Gazebo node listens to the same topics as 5DOF Robot, and will simulate a robotic model with a 1:1 relationship with the physical arm.

As described in **todo: refer to XX**, we had some challenges with finding the inverse kinematics and have to depend on the Robotics System Toolbox from MATLAB. When running MATLAB as a ROS node, we discovered that it ran quite slowly together with Gazebo. For this reason, we decided to run the ROS network over two computers to separate computing power. This is done by setting a couple of environmental variables in every terminal that uses ROS. Two shell scripts were made for this, and can be sourced to achieve the distributed ROS network. The different colored nodes in figure 2 represents which computer the node runs on.

Camera

The camera package contains the python-script that is responsible for handling object detection in ARPA. By placing red markers in the physical coordinate system of ARPA, the user will be able to track the markers with the live video feed from OpenCV. The script will after calibration continuously map received coordinates from the OpenCV coordinate frame to coordinates corresponding to the

physical coordinate frame on the rig. The calibration and usage during runtime will be elaborated in finer detail in **XX: SECTION**.

The color-detection is done by applying a red mask in the HSV-colorspace on each frame from the live video. By finding the contours of the image the marker can be detected and the script further finds the center of the detected point. The script is incorporated in the ROS environment as a node that publishes the mapped cartesian x and y coordinates of a desired point. The script can be extended to detect more colors by adding additional

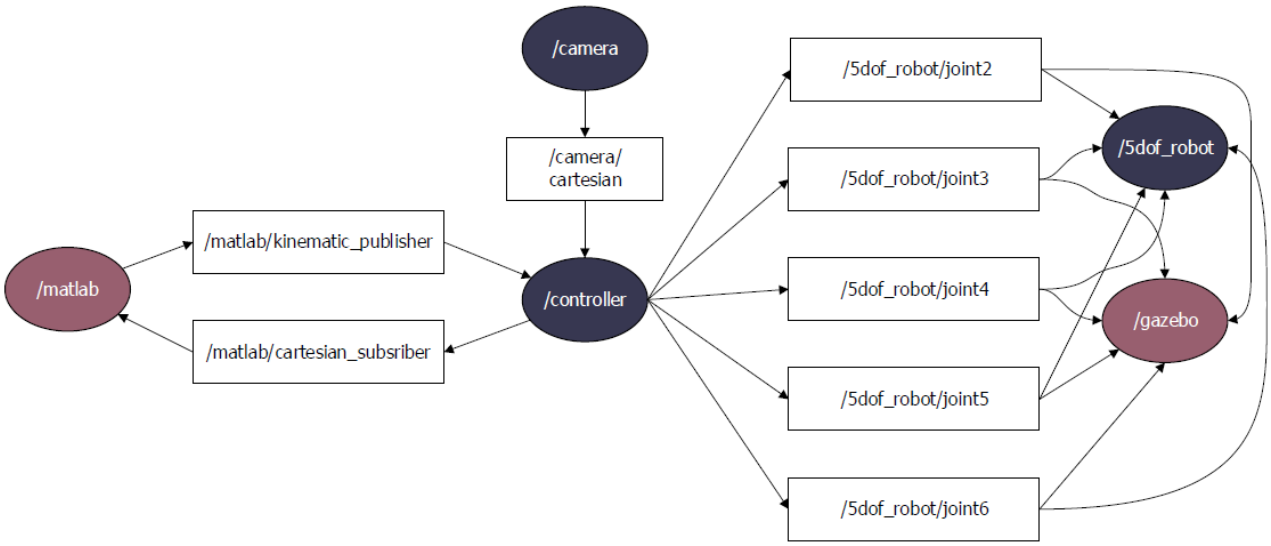

Figure 2: Node graph

0.1.2 Information flow

The flow of information in ARPA starts when the user sends a ready signal to the controller node. This signals that the controller will accept a Cartesian coordinate from the camera. This coordinate further sent to MATLAB for processing. MATLAB returns five joints angles to the controller. These are the angles used to control both the physical arm and the Gazebo model. Hence, the controller publishes them to their respective topics, where they are picked up by both the Gazebo node and the 5DOF Robot node. After the control node has sent the joint angles, it waits for 5 seconds before it sends a new set of joint angles used to set the physical and simulated arm back to starting position. This process can be repeated by sending a new ready signal. Figure 3 shows the flow of information in the form of a UML Sequence diagram.

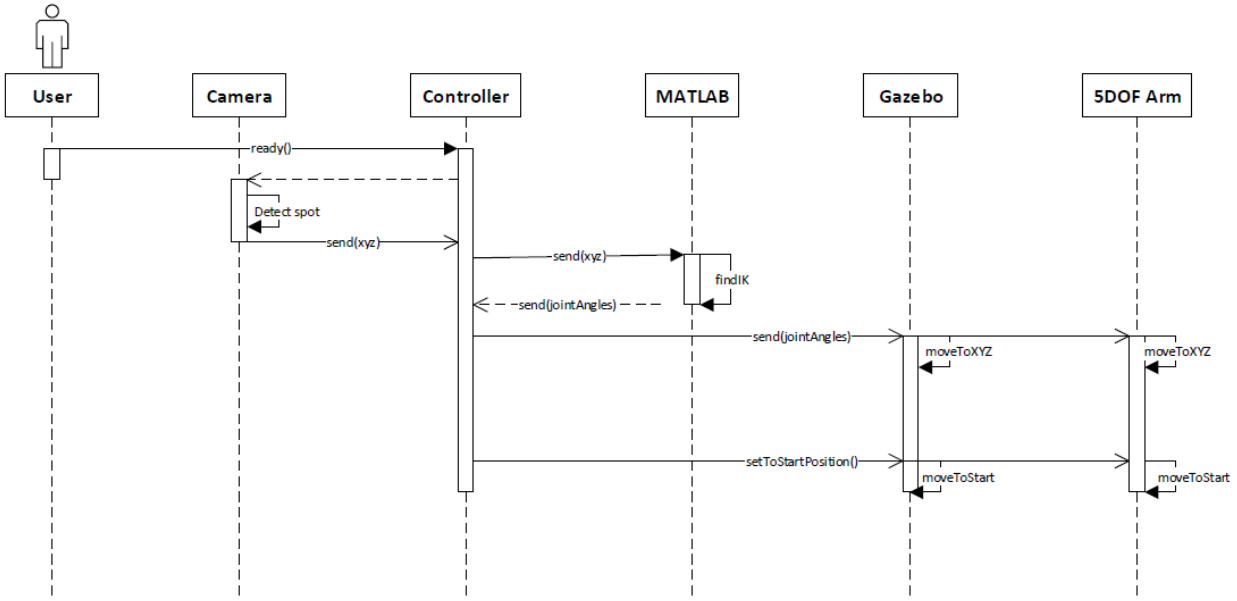

Figure 3: UML Sequence diagram

0.1.3 Starting program

When running ARPA over two computers, one of them must be chosen to run as the ROS master. The selected master computer must source the master.sh script in every terminal used for running ROS commands. Next, the slave.sh script must be modified to contain the IP-address of the master computer. The address can be read from the output of the master.sh script. When the slave script is modified to contain the correct address, it must be sourced in all terminals on the slave computer that runs any ROS commands.

Next up is to launch all ROS nodes. First of all, the setup.bash script in the devel directory of the catkin workspace must be sourced. When this is done on the master computer, the Arduino controlling the servos of the robotic arm must be connected to the computer. When it is connected, the command *roslaunch robotic_arm master.launch* can be run. Note that the launch file is configured to setup the ROS node on /dev/ttyACM0. This means that either the Arduino running the code must be connected before the Arduino used for power supply (see), or the launch file must be modified to contain the correct device parameter. The roslaunch command starts the camera, controller and the 5DOF robot node.

Calibration of camera

The detection system used in ARPA is based in Python and the OpenCV library, and can be used together with an internal or external camera. The python script is in the beginning made so that a user has to go through a calibration phase to ensure mapping from the OpenCV coordinate frame to the coordinate frame of the physical rig.

Before running *roslaunch robotic_arm master.launch* there should be a red marker placed in ARPA's origo. The x-axis on the physical rig should also be parallel with the bottom border of the OpenCV window frame. The camera will during the first five seconds self-calibrate for lighting and focus, before masking the picture to find the red marker representing origo. After capturing 50 frames there is a five second pause in which the user will have to move the marker from origo to x-max, after an additional 50 frames the same will be repeated for y. When the script has defined all needed points of the coordinate frame, the values from the OpenCV frame is mapped to the lengths of the physical coordinate frame defined in the script. This method ensures that the same calibration method can be used independent of the cameras height above the rig, as long as origo, x-max and y-max is in the camera-frame.

As with the master computer, the slave must also source the setup.bash script in the devel directory of the catkin workspace. When this is done, the command *roslaunch simple_robotic_model joints.launch*

can be run. This starts both the Gazebo and MATLAB node.

0.1.4 User interface

The user interface of ARPA is split into two: one on the master computer and one on the slave computer. Figure 4 shows the user interface of the master. The window to the left is a live feed of the camera. It shows the user the detection of a red dot, i.e. where ARPA will move when given a ready signal. The window to the right allows the user to send this ready signal. When the ready.sh script is executed, the controller node will accept the next incoming coordinates and execute all necessary communications and actions for both the physical arm and the Gazebo model to move.

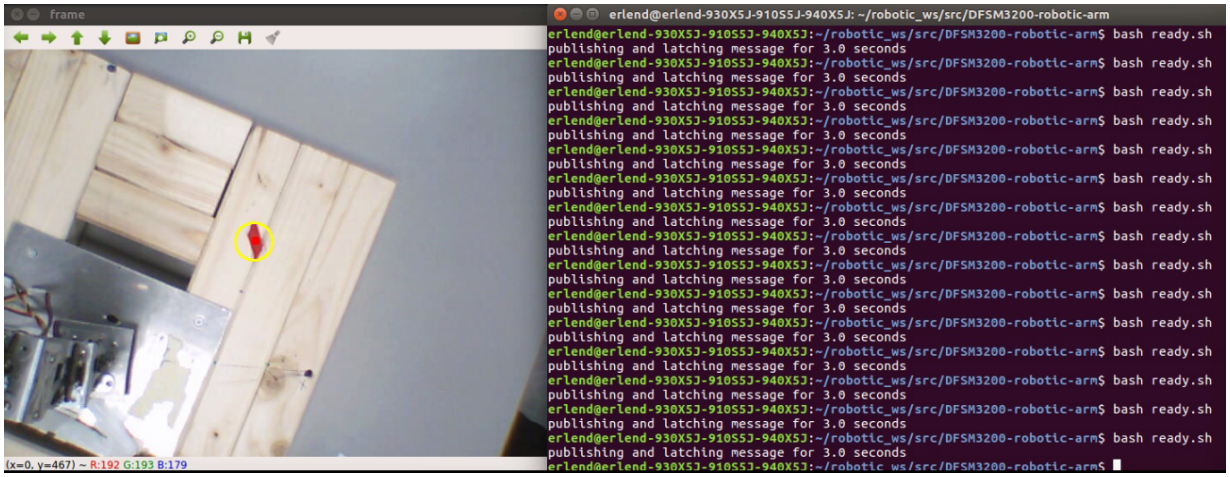

Figure 4: User interface: Master

Figure 5 shows the user interface of the slave. Gazebo is launched as described in *0.1.3 Starting program*, but the rqt window on the left is not included as a part of the launch file. The user can choose to start rqt from a new terming through the *rqt* command. The preferred setup for this project was to have rqt run the console plugin to show all the ROS messages in a GUI.

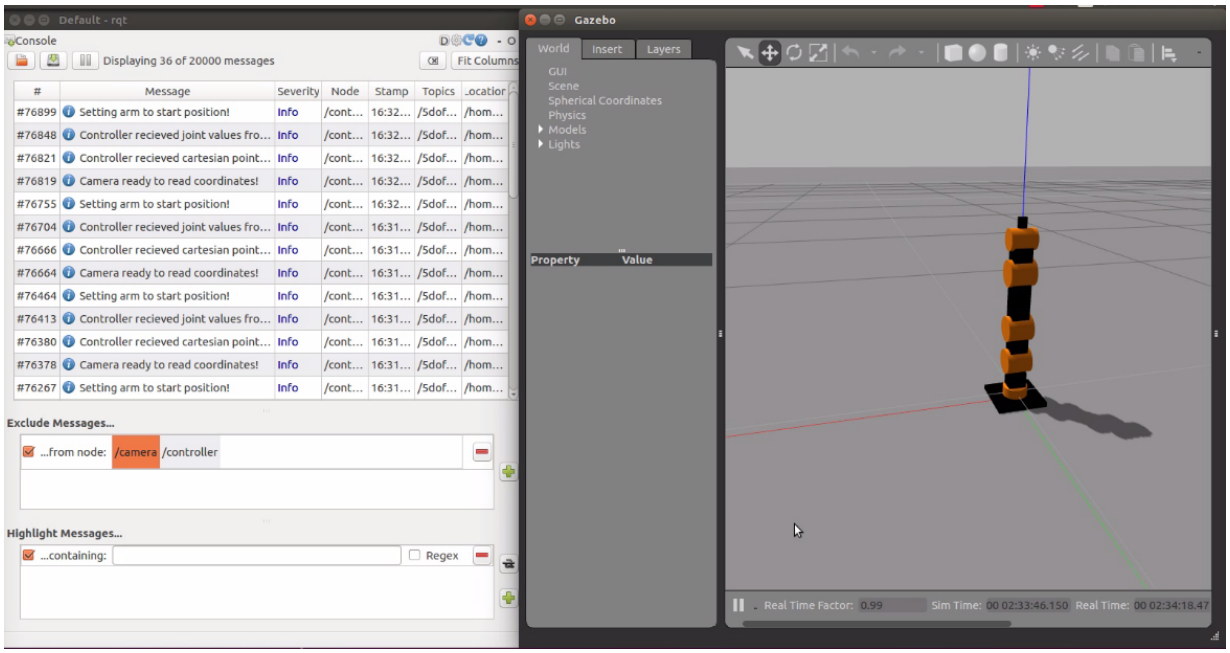

Figure 5: User interface: Slave

0.1.5 Dependencies

This section will summarize all the software dependencies used in the development of ARPA.

- **Ubuntu 16.04 LTS:** Operating System [http://releases.ubuntu.com/16.04.4/]
- **Robotic Operating System(ROS) Kinetic:** Framework for robot software [http://wiki.ros.org/kinetic/In
- **rosserial_arduino:** Include Arduino into ROS node network [http://wiki.ros.org/rosserial_arduino]
- **MATLAB:** Programming platform [https://se.mathworks.com/products/matlab.html]
- **Robotics System Toolbox:** MATLAB add-on providing different robotic software [https://se.mathworks.c
- **OpenCV2:** Computer vision library [https://opencv.org/]
- **Python:** Programming language [https://www.python.org/]

0.1.6 Hardware

During the project we had to make some adjustments to parts of the hardware related to the robotic arm. This section will go explain which adjustments has been made, and why.

Rig

When we were handed the robotic arm at the beginning of the course, we did some testing of it. First of all, we discovered that it had both had some loose screws, and also missing some. This lead us to spending some time on making the arm more rigid and secure. Figure 6 shows this stage of the project.

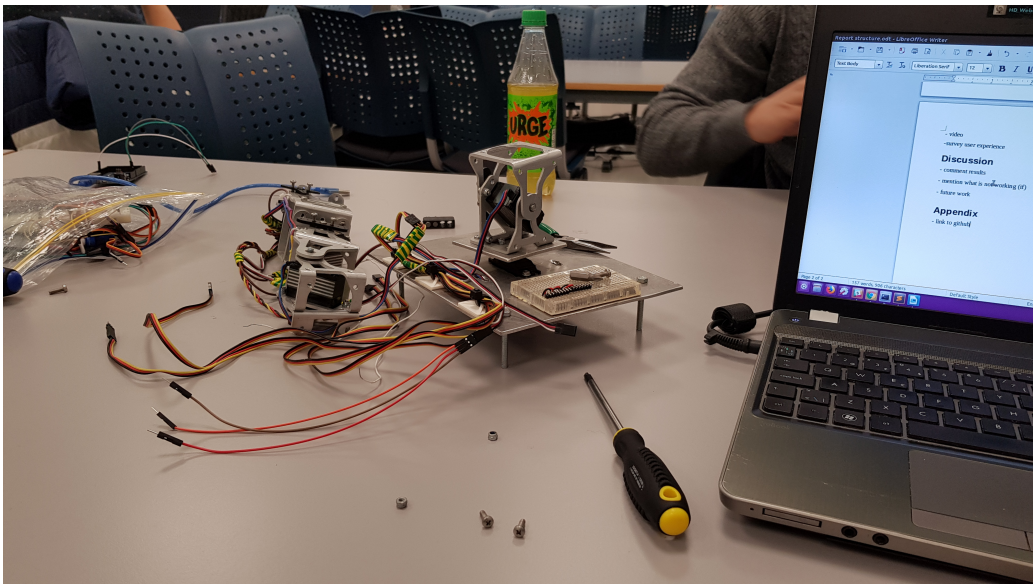

Figure 6: Early stage of the hardware

During the first tests of the robotic arm, we discovered that the torque would make the robot fall over. We decided to make a more stable mechanical rig, by attaching the robotic arm to a stable wooden rig. Figure 7 shows the process of making the wooden rig.

Power supply

The servo motors are running nominally at 6V. As the arm consists of five motors, we could not power them all from the Arduino. We decided to buy battery holders to run 4 AA batteries in series. Since an AA battery is 1.5V, running four of them in series adds up to 6V.

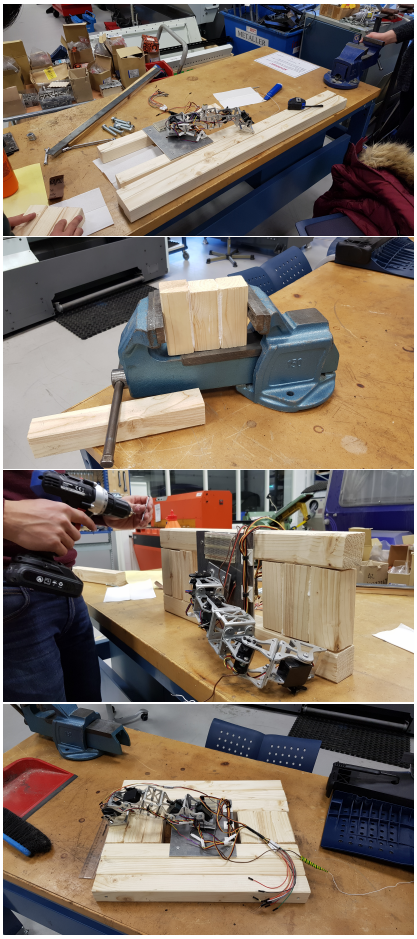

Figure 7: Process of making a wooden rig

Due to the bigger torque on the bottom motors, we decided to supply the bottom three motors with 6V each, having them run with nominal voltage level. The upper two motors of the arm are powered by one Arduino each, meaning they get 5V.