Optimizing the Universal Robots ROS driver.

Andersen, Thomas Timm

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Technical University of Denmark

Thomas Timm Andersen

Optimizing the Universal Robots ROS driver

Technical report, November 2015

DTU Electrical Engineering
Department of Electrical Engineering

Abstract

In this report I will examine both the current and the possible performance of one of the most popular robotics platforms in research, the Universal Robot manipulator. I will solely focus on the ROS based approaches and show how the current driver can be improved. I will look at performance improvement both in terms of faster reaction as well as making it possible to control the robot using either ros_control or ordinary joint speed commands, which is required for many types of sensory based control like visual servoing. The developed driver is compared to the drivers already existing in the ROS framework to prove the improved performance.

Table of Contents

Introduction

Planning and executing trajectories on a robot manipulator can be a challenging task if we want to have a saying on how and when the trajectory is carried out. To ensure maximum control, researches usually have to build their own manipulator or spend years on reverse engineering an existing industrial manipulator.

With the introduction of ROS (Robot operating system) from Willow Garage in 2010 and especially the forming of the ROS-Industrial organization in 2013, researches and robotics implementers were now given better than ever access to other researches work, helped a long way by public repository and revision services like Github. Today, it is possible to set up a robot and start doing advanced trajectory planning, including collision avoidance and grasp planning in a matter of days or weeks.

By using ROS and ROS-I compatible manipulators, end effectors, perception systems/sensors, mobility platforms, and peripherals, users know their equipment can all speak one language (ROS messages) and interoperate regardless of OEM brands or communication bus.

One of the biggest advantages of ROS might also prove to be its biggest disadvantage. When a driver, being it for a sensor, an actuator or a complete robot package, has been part of the ROS ecosystem for some time, it will naturally have grown to have a solid user-base. Most of these users would like to see the driver evolve and implement new features, but no one wants breaking changes introduced that breaks existing code or projects.

The driver for the Universal Robots industrial manipulator is a prime example of this issue. It has been part of ROS ever since ROS Electric, which was released in august 2011. Since then, significant contributions within trajectory planning and execution have been added to ROS, like OpenRAVE with IkFast, MoveIt for manipulation tasks, and ros_control for ease of controller implementation. While support for the first two has been added by adding to the growing single-file Python implementation, the latter cannot be added due to the use of Python. Other features like IO control has also been added on top of the implementation over the years.

As further development of the driver thus requires a completely rewritten implementation in another programming language (C++), it is worth to simultaneously evaluate the current driver and how it works to research whether it is possible to improve the performance; preferable without breaking backwards compatibility.

Thus in this report I will examine the current and research the possible performance improvements of one of the most popular robotics platforms in research, the Universal Robot (UR) manipulator. I will solely focus on the ROS based approaches, as this is what is mostly used for controlling the robot. While evaluating the current drivers, I will also take notice of other areas in which these can be improved. The combined effort will be put into a new driver that can be used both stand alone or in a supplied ROS wrapper.

The Universal Robot controller

In this chapter I will describe how the controller works and what possibilities there exist for controlling the robot via network, which is required for doing a ROS-based driver.

Programming

There are three different ways of programming the UR: Via the teach pendant, using URScript or in C. The teach pendant can only be used for offline programming and that programming method will thus not be covered in this report

The internal controller which sends commands to the joint servos runs at 125 Hz and thus evaluate an instruction each 8th millisecond.

URScript

URScript is a Python-based programming language that makes it possible to quickly program the robot using simple instructions like *movej* (move in joint space) and *movel* (move linear in Cartesian space). URScript also supports more advanced instructions like conveyor tracking, socket and Modbus communication, threading, and force/torque based control.
The latest controller (at the time of writing), the CB3, also supports RPC calls.
The commands can be joined together in functions to form an entire program with syntax very similar to python.

Some of the implemented motion methods react in suboptimal ways when interrupted; this is explained further in [1].

C programming

Universal Robot also exposes a C library to control the robot with. This makes it possible to make a custom controller that runs on the robot and thus has very little delay. It is still not possible to send commands to the robot faster than 125 Hz though.

A disadvantage of using the C API is that it is necessary to install the program on the controller and run the custom program instead of the original firmware. While not especially difficult, some users, especially in the industry, might fear bricking their robot by doing so. It is also a bit difficult to change between the stock firmware and any custom developed, as this required root access to the controller.
Another drawback of a custom program is that it is not possible to use the teach pendant, unless a custom GUI is also developed or the current protocol between the original firmware and the original GUI is reverse engineered.

Ethernet connections

When the stock firmware is running, the controller is providing data representing the robots state, positions, temperatures, etc. through a few TCP/IP server sockets in the controller. A custom program can be written to read these streams.

Robot state interfaces

The controller exposes two interfaces, which a client can connect to; on port 30001 and port 30002. They are intended to provide information about the state of the robot. The information is statuses like joint position, voltages and temperatures across the system, IO status, TCP force, Cartesian position and so on. The primary interface (on port 30001) publishes both robot state, version information and messages of various kinds to the GUI, while the secondary interface only publishes state and, as of firmware version 3.1, version information. The robot state includes information like emergency and protective stop, temperature of hardware, I/O states and positional information. While it is possible to send commands to the robot via these interfaces, it is not recommended, as they are not read in a deterministic manner.

Real time interface

The real time communications interface (also known as the Matlab interface) is found at port 30003. It publishes a limited set of robot state information at 125 Hz. The information includes both actual and target joint position and joint speed, as well as actual joint moment, joint current and joint temperature. Digital input values and a few status bits is also published. Measurements shows that the interface has a command buffer, so sending joint speed commands at a rate faster 125 Hz will not degrade performance. Note, though, that sending motion commands faster than 125 Hz can lead to undesired operation like buffering, as explained in [1] page 6.

It should also be noted that there seems to be congestion control activated on all the interfaces in the sense of using the Nagle-algorithm. This can be circumvented by using setting the TCP_QUICKACK on the socket upon reception of a package. Universal Robots should have fixed this from software version 3.0.15547 or 3.1.16828.

Dashboard server

The dashboard server is found on port 29999. It is used by the robot GUI to load and execute programs, and can thus not be used directly to control the robot. As of firmware version 3.0 it can be used, though, to power the arm on and off, release the brakes and from version 3.1 unlock the protective stop. From version 3.1.17136 it can also change the user role of the GUI, making it impossible to move the robot with the teach pendant.

Summary

In this chapter, I have looked at the advantages and drawbacks of the different ways of programming the UR. The URScript is the easiest and best documented way of programming the robot.

The C API opens up for direct control of all the joints, but requires using an API with little documentation, and installing and running the custom program is not a user-friendly task, just as the touch pendant is useless with this solution.

I have also looked at the different TCP/IP ports, which enables communication with the robot, and the data available through these connections. It is possible to get all the state information desired from the robot while having a separate connection for commands, which also serves the most essential data for controlling the robot.

Existing ROS solutions

In this chapter I will examine the existing drivers for the UR to evaluate pros and cons as well as how they perform.

ur_driver

The ur_driver has been part of ROS since ROS Electric and was originally written by Stuart Glaser. Since ROS Groovy, Felix Messmer, Alexander Bubeck, and Shaun Edwards have made significant contributions to further improve on the design to bring it up to where it is today.

Joint states and force/torque are supposed to be published at 125 Hz, while the I/O states are published at 10 Hz. Practical experiments sometimes show a drop down to 62.5 Hz of the states, though.
The driver exposes a follow_joint_trajectory action server interface for sending complete trajectories in joint space to the robot, making it compatible with MoveIt.
The driver listen for service calls for changing the state of the I/O on the robot controller.

The driver is written in Python, which is plenty fast for communication at 125 Hz. It works in several threads

- Two threads connect to port 30002 and 30003 respectively, where they parse the received robot state information. This information is published into ROS
- The driver also sends a URScript program to the robot, which instructs it to open a new socket on port 50001. Byte-encoded instructions are sent to this port and then parsed by the 230 line long custom program which calls the appropriate URScript functions.
  This script handles both joint position changes and requests for I/O changes.

When a trajectory is sent to the action server interface, the ROS node interpolates between the trajectory points and then sends all the points to the custom script, which executes them as *servoj* instructions. The interpolation is done in time steps of 20 ms, while the *servoj* instruction is executed with a 't' parameter of 80 ms. This 4 times oversampling makes sure that there is always a updated value for the next *servoj* instruction to give a smooth trajectory execution, while leaving the closed-loop position-based control to the controller. A pseudo code URScript is shown in Appendix A. This is a stripped down version of the script used in the driver to show how the treaded approach works. The actual script is much longer and handles both servoj commands, movel commands, setting output of the controller and sending debug messages back to the calling python script.

This approach introduces some significant drawbacks:

- By writing a client in URScript, the amount of code needed to be maintained and synchronized with the ROS server side is significantly increased. The loop for calling *servoj* is run at a 12.5 Hz, although the robot should be controllable at 125 Hz. The rate can be increased, but old tests shows instability and jerking motion starts to show around 15 Hz. Later tests have showed that the loop can be executed at 31.25 Hz or, with some previously smoothed trajectories, at 62.5 Hz. The increased performance is speculated to come from a firmware upgrade, but this is not confirmed.

- The script introduces significant delay of more than 160 ms from a command is received at the action server until the robot has actually moved. This can be seen in Figure 1 below.
- As a script is always running on the controller, even when the robot sits idle, it is very inconvenient to use the teach pendant's touch screen to move the robot. Using the touch screen stops execution of the script, which is detected by the ROS driver and then resend, which again makes the touch screen unresponsive until the user lifts his finger and presses the screen again to once again stops execution. This result is slow jagged movement as the robot can only be moved in a step of about 1 second at a time.
- As the controller is written in Python, it is incompatible with ros_control
- Recent changes in the protocol of the robot state interface for firmware version 3.1 seems to have broken compatibility with this driver

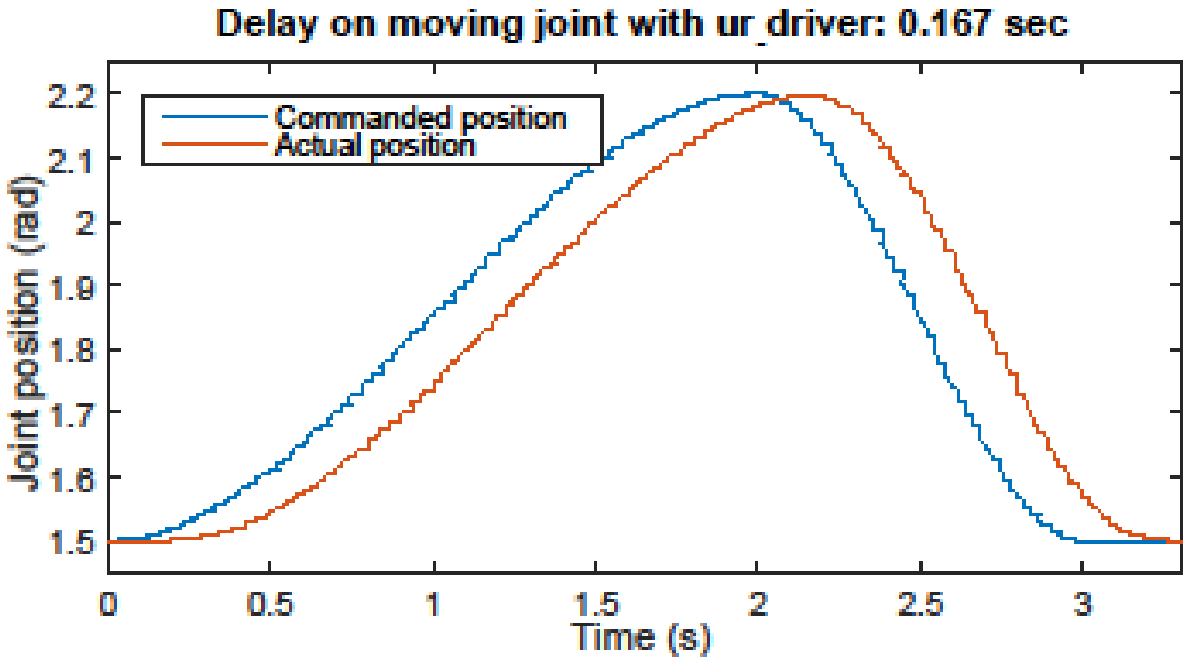

Figure 1: Delay when using the old UR driver

ur_c_api

The ur_c_api is written by Kelsey Hawkins from Georgia Tech in an attempt to remedy some of the drawbacks of the Python/URScript based driver; in particular the low data rate as well as the inability to do speed control.

The server side (ROS) is based on the ros_industrial client and reuses much of the protocol, and the client-side, running on the robot controller, is written using the before-mention C API. The driver also supports controlling the robot using ros_control.

The idea of reusing code from ros_industrial is advantageous in maintaining the code, but at some point ros_industrial changed some network byteorder, so now a separate branch of the ros_industrial client is needed.

The use of the C API also introduces the earlier mentioned problems of having to install programs on the robot controller and disabling the teach pendant.

The ur_c_api is at the time of writing (September 2015) only supporting control box CB2 and not the newer CB3 which was introduced with the UR3 and is used for all new shipped robots. Except for a few bugfixes the last commit to the code on Github over a year old, and it is not officially a part of the ROS distribution, so the future of this driver is unknown.

Summary

In this chapter, I have evaluated the features as well as the limitations of the currently known driver for the UR in ROS.

Feature-wise they both provide an interface that makes it possible to control the trajectory of the robot using the standard ROS interface for moving industrial robot arms; MoveIt. The Python-based driver also makes it possible to toggle I/O on the robot, while the C API based can send speed commands to the joints. The Python-based also have support in the UrScript to send a few other hardcoded types of commands like *movel*, but no ROS interface is exposed for this.

There exist quite a drawbacks and limitations with the two current drivers, the main one, which they both share, being the apparent incompatibility with the newest firmware. Both drivers also have several hundred lines of code to be executed on the robot controller-side, which needs to be maintained alongside any changes to the ROS side code.
For the Python-based, the need for maintaining a large custom script with a byte-encoded self-defined protocol is certainly a drawback. The ROS side driver is also mostly lumped together in one big Python file, except for the parsing of packages receives on the two interfaces on port 30002 and 30003. It would also be desirable to do away with the relatively big delay from a command is sent until the robot starts to react. From a usability point of with, the lock-up of the touch screen is one of the most commented issues of the driver on github as well as the ROS-I mailing list. It would also be great if it was possible to do joint speed control of the robot for visual servoing and other kind of sensor-based control. Finally the use of Python makes it incompatible with ros_control unless some kind of bridging software is written.

The C based driver is especially limited by the fact that it needs to be installed on the actual robot. This means either the users need the root password for the computer to start the alternative controller, or the

robot needs to be hardcoded to always launch the alternative controller, which makes it impossible to use the robot without the specific ROS driver. As the C driver does not utilize the touch screen on the robot, the robot cannot be used at all without using the ROS interface.

Optimal control of the robot

In the previous chapters I have looked at the different ways to program the robot, how to communicate with it, and how the current drivers work. In this chapter I will research the various ways to control the robot and evaluate which one performs the best.

Trajectory control

In doing said evaluation, it is important to determine the metric by which the performance is to be evaluated by. The robot needs not only to be able to move from point A to point B, but also be able to continue in an arbitrary trajectories through points C, D and E with an arbitrary time of arrival, velocity, and acceleration at each point. These points are not necessarily on a straight line in either Cartesian or joint space. In executing the trajectory, accuracy in terms of position, velocity, and acceleration is obviously crucial measuring points; both at any intermittent points in the trajectory, but especially at the end position. The delay from the trajectory is sent until execution is started, also known as the reaction delay, is also taken into account.

Note that in this chapter, as well as the rest of this report, I will not consider a C API-based solution. An approach based on this API is not considered sufficiently user friendly to be appealing for a broad user base, but should rather be used for small specialized projects where the use of the C API might give the last bit of performance needed.

Commands available

In the URScript language, there are four different methods of moving the robot: move, servo, speed and force

Move

The move commands are meant for simple movements from one point to another. Motion can be linear in Cartesian or joint space, or circular in Cartesian space. While several move commands can be executed in succession, the robot will always come to a hold at each point, unless a blend value is given. The blend parameter will make the robot blend two trajectories together, but the robot will not reach the points in the middle of the trajectory. This makes the move commands unsuitable for precise trajectory control, where it might be necessary to reach each point in the trajectory to avoid obstacles or do a proper welding seam.

Servo

The servoj command is intended for much tighter control of the robot. From firmware version3.1, the URScript manual directly states that “Servo function [is] used for online control of the robot”. In this firmware revision, a lookahead_time and a gain parameter can also be specified in the call to the function, which “can be used to smoothen or sharpen the trajectory”.
The command moves the joint to the desired position and then stops all motion, so it is up to the user to use several servo commands to make the arm come to a soft stop.

Speed

The speed command makes the robot move with a constant velocity in either joint, Cartesian base or Cartesian tool space. Once the desired velocity is reached, the robot keeps moving at that velocity until a new command is issued, or a joint limit is reached.

Force

For control makes it possible to make the robot compliant along one or more axes while still moving along other axes. It is thus not useful for trajectory control.

From the above analysis, it is clear that trajectory control should be done either with servo or speed commands. There are several different ways to do this, though. I will analyze these possibilities in the rest of this chapter.

Commanding a trajectory

There are three different ways of commanding a trajectory: continuously stream the command, precompute the trajectory and send it all as one program, or have the command execute continuously on the controller in one thread and then just send the new parameters which is read in another thread. The three methods are listed in increasing level of difficulty to implement and maintain. The tests are carried out on the end joint to minimize the influence of dynamics.

Streaming commands

Streaming a new command at a fixed time interval is probably the most intuitive method for controlling the robot. It goes well with the ordinary way of thinking about robot problem solving; do this, then do this, then this etc. etc. Therefor it is the first programming strategy that I will examine.

Servo-based

The result of the servo-based streaming of commands can be seen in Figure 2. Three things should be noted here: Firstly there is an apparent delay of 170ms between a position is send until the joint is at the

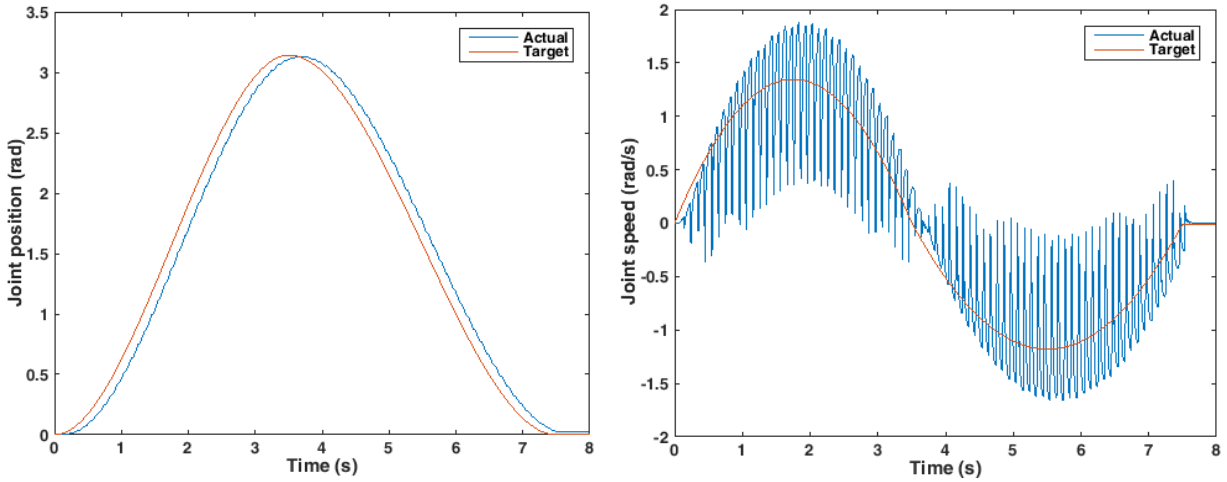

Figure 2: Streaming servoj commands. a: Joint position. b: Joint velocity

position. Secondly the robot does not reach the desired extremum positions, neither at the middle of the trajectory nor at the final position. Finally, and most problematic is the oscillating velocity of the joint. This oscillating behavior, which is also mentioned in [1], makes streaming of servoj commands impractical.

Speed-based

The result of the speed-based streaming of commands can be seen in Figure 3.

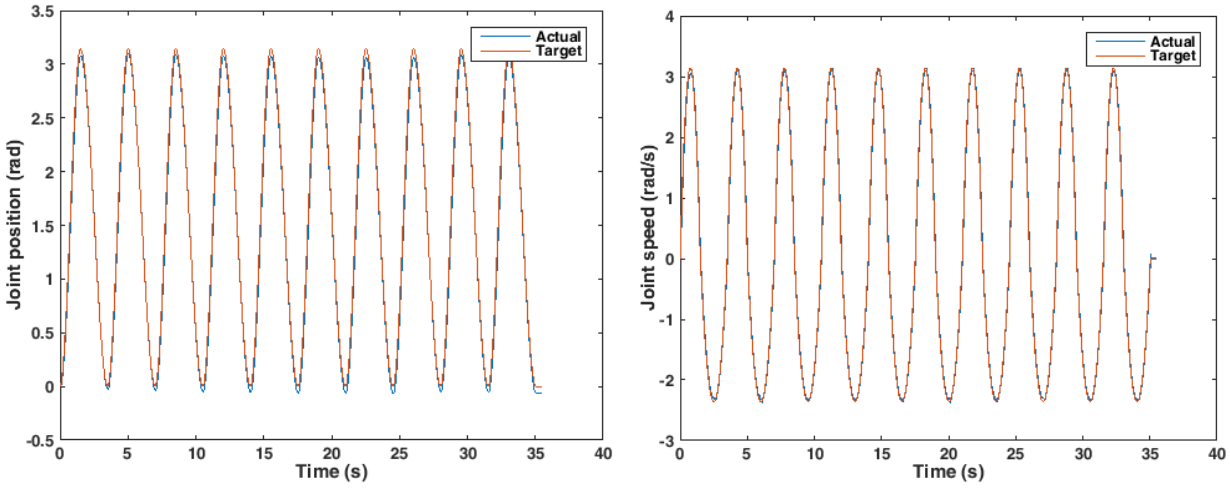

Figure 3: Streaming speedj commands. a: Joint position. b: Joint velocity

As it can be seen, the reaction time is really good. The delay will always be larger than 1 sampling period, and will optimally be somewhere between 1 and 2 sampling periods, depending on when in the control cycle new values are read. The reason for this is that the logged data is the status message, indicating the actual position/velocity/effort/etc. and the target values that the internal controller used to get to that state. Thus even if data is read by the controller just before the next control cycle starts, the new data will only be reflected in the status message sent after that control cycle. With our setup I have found that data should be sent no more than 3 ms after a status message is received to be reliably executed in the next control cycle (and thus show up in the status message received 13 ms later), but this is highly dependent on network topology and –load.

The velocity accuracy is also good, although not spot on. Looking on the position, it is clear that some drifting occurs. The trajectory is executed 10 times to more clearly show the drifting over time. As this approach is based on using speed to control position, some kind of controller is necessary to limit this drifting, but as the velocity accuracy is good and the reaction delay is low, it should be a feasible solution with a properly tuned controller.

Precomputed trajectory

In traditional trajectory-based control, the end-point is known when the trajectory execution starts. This makes it possible to upload the entire trajectory as one program in a sort of fire-and-forget manner. An obvious shortcoming of this method is the fact that it is not possible to do online corrections of the path, i.e. if the target pose, or obstacles, change position. As this is seldom possible in any other control methods and as ROS does not support this either, this method is still investigated despite this shortcoming.

Servo-based

The result of the precomputed servo-based program can be seen in Figure 4. It is clear that this method introduces a significant delay. The reason for this delay is that the controller needs to validate the entire program before executing it. As the program is 8 sec * 125 Hz ≈ 1000 lines long, this takes a while. For a

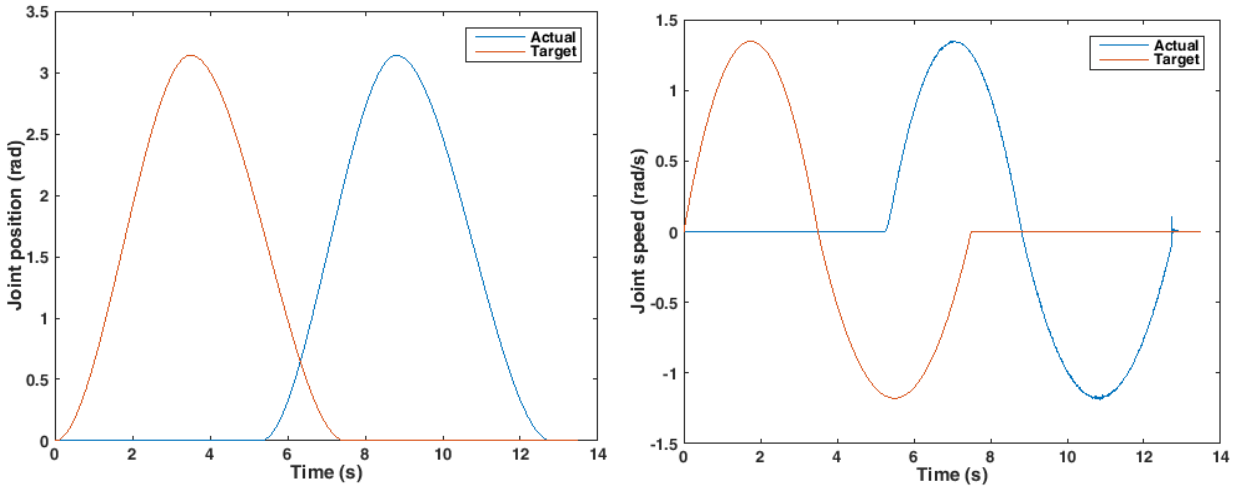

Figure 4: Precomputed trajectory with servoj commands. a: Joint position. b: Joint velocity

shorter program, the delay was not as significant. But as the delay scales linearly with the length of the trajectory, the method is not applicable to the current problem, as the trajectory can be arbitrary long.

Speed-based

As mentioned above, a controller is needed to avoid positional drift with a speed-based approach. Therefore it does not make sense to test a fire-and-forget method based on a speed command as that would exhibit the exact same behavior as when streaming speed commands in open loop.

Threaded solution

The treaded solution is the one used in the current Python-based driver. It works by having two thread; a main thread that communicates with a host PC and a worker thread that calls servoj in an endless loop with a time parameter of 0.08 ms, equivalent of repeating the loop at 12.5 Hz. The host PC interpolates joint positions at 50 Hz and sends them to the main thread, which updates some global variables with the new joint positions. These positions are read by the worker thread and used for calling servoj. This is shown in Appendix A.

Thus the position input is oversampled a 4 times the execution speed, meaning that the robot controller and the host PC does not need to be synced, but new and fairly accurate positions are always available to the worker thread, even with varying network load.

As mentioned in the previous chapter, this method introduces a delay of ~160 ms.

As it is also mentioned it should be possible to run the loops on both the host PC and the worker thread significantly faster. Increasing the rate in the current driver too much does lead to some instability of the joints though; the nature of which is unknown. The highest rate I've been able to attain without problems is 62.5 Hz for the worker loop and thus 250 Hz on the host PC. This was done on a UR10 with a controller running firmware version 1.8. In the URScript manual for firmware version 3.1, they recommend a time parameter of 0.008 sec (note the extra zero; this is 10 times faster than what the current driver does). It should thus be possible to run the worker thread at 125 Hz. The reason for instability could be that the main thread is not fed data fast enough and that Python is not fast enough, although the idea is sounding unlikely. If it was the main thread that could not read data fast enough from the socket and thus queued up

data in the input buffer, the delay would grow unbounded, but should not lead to jerky motions – unless the input buffer is filled, overflowed and cleared several times per second. This sounds just as unlikely.

In the tests I've conducted for this chapter I haven't been able to reproduce this problem. In my minimal implementation, I ran the host PC loop at 500Hz and the worker loop at 125 Hz without problems. The URScript executed on the controller is very similar to the one in Appendix A, that is, without all the extra 170 lines of code from the original script. As the target outcome of this project is a new driver in C++, further time will not be spend on debugging a problem related solely to the old driver.

A disadvantage of the threaded approach is that a program always has to run on the controller, and that program is killed if the robot is moved with the teach pendant. As the user should not attempt to move the robot while it is being moved externally, this can to some degree be circumvented by stopping the script once execution is done. With the dashboard server, it is also possible to lock the screen and thus prevent

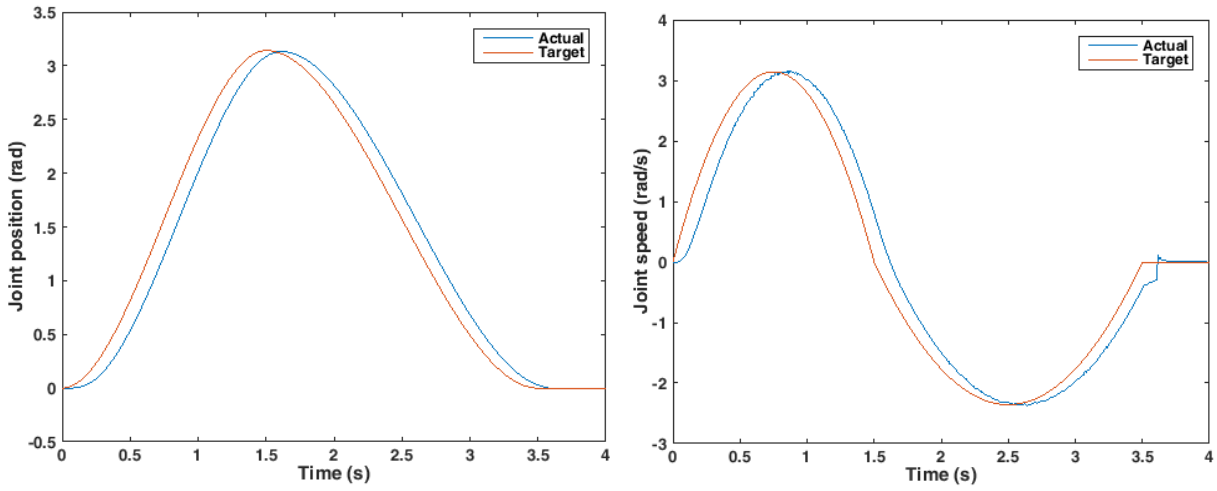

Figure 5: Executing servoj commands in a separate worker thread at 125 Hz. a: Joint position. b: Joint velocity

the user from killing the program.

Servo-based

The result of the threaded servo-based program can be seen in Figure 5. As mentioned, servoj is called at 125 Hz. Note that the trajectory is faster than in the other tests to try and show the unstable behavior known from the old driver with fast trajectories. As it can be seen, the reaction delay is about 124ms. This is better than what is seen with the old driver, but still quite high compared to what is desirable. Note that this is with a UR10 running firmware version 1.8.14035.
When the last parameter is sent, the program is stopped as described above. Thus, when the last servoj command has been executed, the robot just stops with no deceleration, as described in [1]. This explains the abrupt change in velocity seen at the end of the trajectory.

As previously mentioned, in firmware version 3.1 the servoj command has two new parameters; lookahead_time and gain. These default to 0.1 and 300 respectively. If the servoj function in firmware revision 1.8 is called with a lookahead time of 100 ms, that could explain a lot of the delay which this method exhibit. According to the URScript manual, the lookahead time can be specified to a value between 0.03 and 0.2 seconds. Below in Figure 6 the same trajectory is executed on a UR3 with firmware version

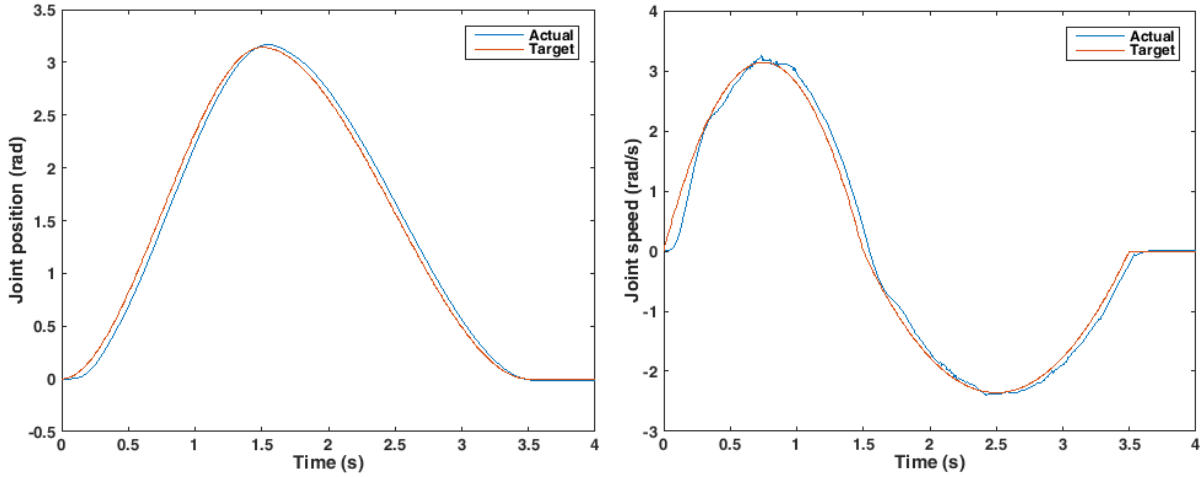

Figure 6: Executing servoj commands on a CB3 controller with firmware version 3.1.XX. a: Joint position. b: Joint velocity

3.1.18213. As it can be seen, the use of a lookahead time of 0.03 seconds does away with most of the delay, confirming the above hypothesis. As the results are made on another size robot with another controller running a different version of the firmware compared to the other tests in this chapter, comparative observations regarding this shorter lookahead's influence on velocity profile and overshoot is not necessary valid and thus skipped.

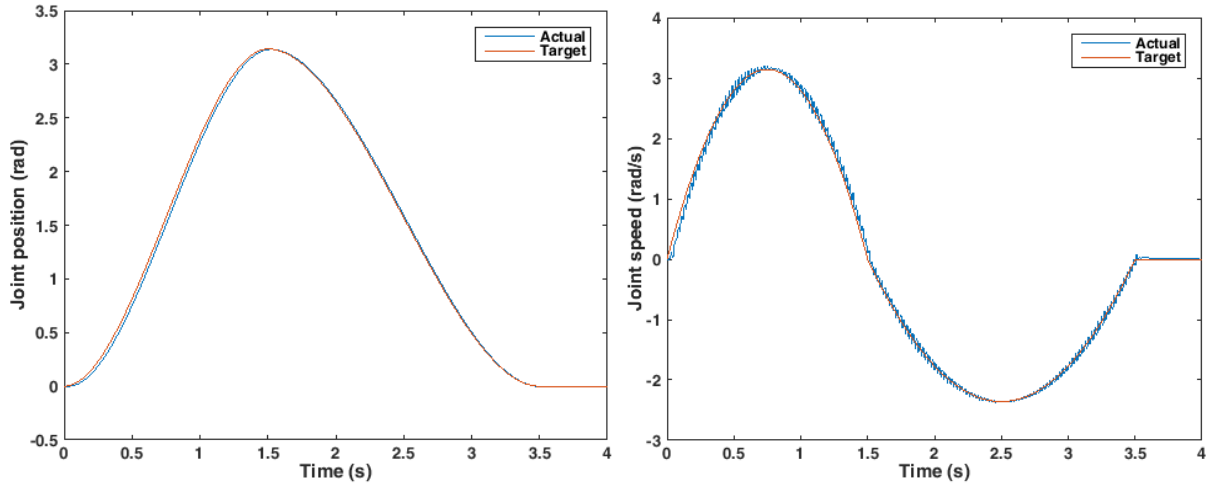

Figure 7: Executing speedj commands in a separate worker thread at 125 Hz. a: Joint position. b: Joint velocity

Speedj

The result of the threaded speed-based program can be seen in Figure 7. As it can be seen, the speedj command does away with the actuation delay, and the robot follows the trajectory really well. To see how good the robot is at following a speed trajectory, the robot was commanded to move back and forth for 750 seconds or 12.5 minutes in open loop. Furthermore the shoulder joint was used to increase the torque. After the 750 second long trajectory, the offset error was only 0.18°. This is really impressive, although at intermittent points, the error was as large as 1.6°.

Summary

For doing trajectory control, it is clear that either the servoj command in threaded mode or the speedj command in either streamed or threaded mode should be used. For the speedj approach, a positional controller should be used for closed loop control. This can be supplied by ros_control.

As the servoj command does not necessarily need an external controller, it is a strong candidate for a future driver. Using servoj also ensures backwards compatibility as such an implementation would mimic the existing driver. As ros_control can do position-based control just as well as velocity-based control, servoj is also a viable solution with respect to ros_control. The disadvantage of the servoj command is the high actuation delay, but for newer versions of the controller, which can run firmware 3.1 or higher, much of this can be removed with a low lookahead time value.

The speedj command has a much lower delay, and is as such also a great candidate for a future driver. The delay can even be made marginally smaller by implementing a threaded solution. The speedj command is also a necessity if the robot is to be used for visual servoing. The drawback is that ideally it needs some kind of controller to close the loop, although my simple results showed very little drifting.

Auxiliary control

Besides controlling the movements of the robot, it is also desirable to be able to read and switch the status of the robot's joint states and I/O.

Reading the robot status

The status of the I/O can either be read from the status stream on the robot state interface or be polled with an URScript command. The same can be said about the joint states, just as their status also can be obtained from the real time interface.

As the status is continuously broadcasted by the robot on the existing interfaces, the optimal way of reading the status is to parse the data stream form the controller. This method does not take up any processing time on the controller nor generate extra network traffic, as the stream is published as soon as a connection to the controller is established.

A disadvantage of this approach is that the communication protocol varies with the different firmware versions. It is thus necessary to have the firmware version and parse the data streams accordingly.
A version message is broadcasted as the first package when a connection the port 30001 is made, as well as on port 30002 as of firmware version 3.1. To ensure backwards compatibility, a connection to port 30001 should be made to establish the firmware version.
The primary interface on port 30001 also includes other messages like information to the GUI, which is not relevant for ROS users. Thus to minimize network traffic, usage of the secondary interface on port 30002 is preferred.

Setting the robot state

Controlling the joint states was covered in the previous section. It is also desirable to look at how the digital and analog output of the robot can be controlled.

The only way to switch the output is with URScript commands. Executing a URScript kills any running command or program, and can thus interrupt the execution of a trajectory. One way to handle this is to wait with the switching of output to when the robot is not moving, but this is quite limiting.

Another option, which is used in the current driver, is to make the program running on the controller able to switch the output without interrupting trajectory execution. This makes it possible to switch output states while the robot moves, but adds to the code that needs to be maintained on both the robot and the ROS client.

The optimal method, in my opinion, is a method for running secondary programs on the controller. A description of the method has only recently been added to the support homepage of the robot, and is not found in the URScript manual. It makes it possible to execute instructions, which does not influence the arm and thus doesn't require processing time, to be executed in a separate thread without affecting any running programs.
To execute a secondary program, the program definition should be prepended with the keyword 'sec' instead of 'def'.

Summary

In this chapter I have evaluated the different methods for controlling the robot in order to find the optimal methods for interaction.

Based on the considerations in the Trajectory control section summary, the optimal implementation would be implementing both a threaded servoj interface and a speedj interface. This will allow using the robot for visual servoing, while still preserving backwards compatibility. It also gives the most diversity and best user experience, as the user is not limited by the driver in what can be done.

The optimal strategy for reading the state of the robot is first to connect to port 30001 to determine the firmware version, then disconnect and re- connect to port 30002 to get the I/O status. By connecting to port 30001, backwards compatibility is maintained. Port 30003 should be used to get the joint states at a higher data rate.

For controlling the outputs, as it has already been argued a secondary program is the optimal solution.

Implementation

Based on the previous assessment, I have created a more modern driver. It is written in C++11, and all the ROS functionality is kept in separate ROS wrapper files, while all the functions related to the actual robot is kept in their own files, so that the driver can be reused in other project without requiring ROS. All output is also kept in its own file and a compile flag select if output should be done via ROS or simple `printf()` statements.

The requirement for C++11 is due to the usage std::recursive_mutex and std::condition_variable for threading, and std::chrono for delays and timing.

In this chapter I will first describe the selected strategy for implementation, to wrap up the accumulated knowledge gathered in this report so far. After that I will give a summary of the most relevant functions of the driver. The main driver class will be explained in details, while the other classes, which aren't meant for standalone usage, will only be described by their intention. Next I describe how it interacts with ROS, before finally showing how the driver performs.

The source code can be found on https://github.com/ThomasTimm/ur_modern_driver/. This report is based on commit 58352a950274a2bf8df4b7da027dcde9e9311c45 from October 30th 2015.

Strategy

For interaction with ros_control, the driver is written in C++. It first connects to the robot on a runtime specified IP address on port 30001 and parses the first data packages received to determine the robot firmware version. To limit the amount of network traffic transmitted the program then disconnects and reconnects to port 30002 to continue to monitor the robot's I/O and state. The program spins up a new thread which continuously reads data from the socket, parses the information and via a condition variable notifies the main thread when new data is available.
The main thread then connects to port 30003 and spins up yet another thread which reads joint data from this socket which also uses a condition variable to notify the main thread of new available data.

For trajectory execution, the main method is based on the old driver approach with a threaded servoj script running on the controller. When a new trajectory is received, the program uploads the script to the robot, and when the trajectory is done the robot is signaled to stop the script. The servoj approach is chosen as this is the only method which works standalone, since any speedj approach requires an external controller. As the script is only running while a trajectory is being executed, it is also possible to control the robot with speedj commands when this is desired. To facilitate this, the program has methods for sending speedj commands and to send a zero velocity speedj command, if a non-zero command has been sent and isn't followed by another command within 0.1 second. This is to protect against a client which crashes or forgets to stop the robot.

For I/O interaction, the secondary methods are implemented. These allow to change both analog and digital outputs, as well as to set the analog voltage level at the tool.

Driver functionality

UrDriver class

The top level class is the `UrDriver`. It implements communication interfaces and methods for trajectory execution and I/O manipulation.

Constructor

The constructor takes a string with the host name and two condition variables which is used to notify the implementing code of new data on the secondary and real time interface, respectively. Optionally values for the minimum and maximum payload which can be set can be supplied, just as the port used for the reverse connection used for the threaded servoj solution. These values, except for the reverse port, can also be changed at runtime using setters. The constructor also takes an argument for the number of times the real time interface can send status updates while the robot is being controlled via speedj, without the control value for speedj is updated. This way, if an external controller sets the joint velocity values but forgets to stop the arm again or it crashes, the driver will set the joint velocities to zero.

The constructor then calls the constructors for a real time interface and a secondary interface before it starts to listen on the reverse port for a connection.

start() and halt()

The `start()` method calls the start method of the secondary interface object. If that is successful, the robot firmware version is retrieved and passed to the real time interface, before this interface is connected. The local computer's IP address is then read from the real time interface, so it can later be sent to the robot controller in the code needed for the threaded servoj solution.

The `halt()` method closes the two connections to the robot and makes the driver stop listening for connection on the reverse port.

doTraj() and stopTraj()

This method is used for trajectory executing using the servoj method. It takes vectors of timestamps, positions and velocities as parameters. At first it calls uploadProg() to start the servoj thread on the controller and uses the `interp_cubic()` method to do cubic interpolation between the target positions. To oversample the values for the `servoj()` call, the loop sleep time is a fourth of the servoj time. When the trajectory execution is done, the URScript is signaled to exit its loop.

If the trajectory needs to be aborted, the method `stopTraj()` can be called. It breaks the loop in doTraj and sends `stopj(10)` to the real time interface.

uploadProg(), openServo(), closeServo() and servoj()

The `uploadProg()` method generates a long string with URScript for the controller to execute. The script starts a thread which calls servoj repeatedly, and then opens a connection back to the driver on the specified reverse port. Though this connection it reads new position values which are passed to the servoj thread.
The URScript is sent to the real time interface and then the method returns with a call to `openServo()`.

`OpenServo()` accepts the incoming connection from the URScript and assigns the connection to a socket file descriptor. It the connection is successful, the method returns true, otherwise false. This result is also saved in a class variable `connected_`.

`Servoj()` takes as argument a vector with the 6 joint target positions and a keep alive flag to signal if the URScript should continue to loop on the controller or if it should break out of the loop and stop.

When the trajectory is done, the method `closeServo()` should be called. It wraps the `servoj()` method and calls it with the keep alive flag set to 0. It takes as argument a vector of doubles. If the vector size is different than the number of joints on the robot (six) it uses the current joint positions as end goal positions instead. It also unsets the `connected_` flag.

Interp_cubic()

This method does cubic interpolation between two points, given the absolute time between the, the time at which the position should be found, and the velocity at the two points. Note that this function does not adhere to any constraints and, as it is a cubic interpolator, is not jerk continuous.

Setters and getters

The driver exposes setters for digital outputs, the tool voltage, analog outputs and for flags. It also has setters for the payload as well as the payload range. Finally it exposes a setter and getter for the joint names.

Furthermore, the class has a setter for the servoj time.

Although not a setter in the usual context, the method setSpeed can be used for speedj control of the arm. It takes 7 doubles as arguments; the velocity of each joint and the maximum acceleration. If a non-zero velocity is set, and no new values are sent within 0.1 second, all the joint velocities values are set back to zero to protect the robot against crashed callers.

UrRealtimeCommunication class

The UrRealtimeCommunication class implements an interface for the real time communication interface. When calling `start()` it spins up a thread which continuously reads data from the robot's real time interface. The thread monitors the available of data, and if a timeout occurs it assumes the connection has been lost, closes the socket and tries to reconnect. This adds to the robustness of the implementation.

The class also has methods for writing to the socket and setting the speed. The latter is a wrapper function which takes 6 velocity values and a max acceleration, creates a speedj command and sends it via the former method.

Finally it has a public `robot_state_` member which is a class representation of RobotStateRT.

RobotStateRT class

The RobotStateRT class is mostly a struct representing all the values sent via the real time interface from the robot. Furthermore it has a method for unpacking a message and populating the members, taking into

account the firmware version and byte encoding of the data.
It also exposes getters for all the data.

UrCommunication class

The UrCommuncation class, much like the UrRealtimeCommunication class, implements an interface for communicating with the robot. Upon calling start, this class first connects to port 30001 to get the version, disconnect and then connect to port 30002. Afterwards it spins up a thread which does handles reading from the socket and monitors for lost communication.
Unlike the realtime class, this class does not expose a method for writing, as that could lead to simultaneously writing two different commands to the robot which could lead to a race and crash the controller.

This class also has a public member robot_state_ which is a class representation of RobotState.

RobotState class

The RobotState class implements several structs to represent the data sent via the robot state interfaces. Currently it only parses the masterboard data, the robot mode data and the version data, but it can easily be extended if the need for configuration data, kinematics info, tool data, Cartesian info or joint data is required.

The masterboard data is used to read the current input and output status.
The robot mode is used to detect if the emergency stop or protective stop is triggered.
The version data is used to determine how all the other data read from the robot should be interpreted.

ROS integration

For integration with ROS, the `UrDriver` class is wrapped in a separate class which exposes the driver's functionality to the ROS infrastructure.

The driver can be configured with parameters at startup using ROS's parameter server. First of all it looks for the parameter `robot_ip_address` to get the IP address of the robot, in compliance with the standard defined by ROS_Industrial. If this parameter is not found, it examines the command line argument, as this is how the IP address was specified in the old driver. If an address is given at the command line, a warning is issued, but the driver continues. If no address is specified, the program exits. The program also reads the parameter `reverse_port` to see if the user has specified which port to use for the connection from the robot to the computer when using the threaded option. If no value is given, port 5007 is used as default as this was also used in the old driver.

Just as the old driver, the program looks for parameters for minimum and maximum payload (`min_payload` and `max_payload`, defaults to 0 and 1), maximum velocity (`max_velocity`, default 10) and if a prefix (`joint_prefix`) should be applied to the joint names, i.e. *left_arm*.

Finally it checks for the parameters `servoj_time` and `use_ros_control`. The former sets the time value for calls to servoj in the threaded implementation, and should be set to 0.08 to get the driver to behave exactly like the old driver. If no value is specified, a value of 0.008 is used for optimal performance.

If the Boolean `use_ros_control` is set to true, the driver sets up a hardware interface and register it with the controller manager for ros_control. It also spins up a thread which publishes the real time data to ros_control to update the robot's joint positions. If using ros_control, the parameter `max_acceleration` (default 15rad/s^2) is also read to limit how fast the controller can accelerate the robot. This can be necessary to avoid protective stops on the robot.
If use_ros_control is false, the program instead starts an action server to function just like the old driver and for compatibility with MoveIt, although MoveIt can also use ros_control. It also spins up a thread that publishes the robot's joint states as well as the TCP force as a wrench message. As per ROS standard these values are published to the topics `/joint_states` and `/wrench`, respectively. The method uses one of the condition variables to get notified of a new value.

Regardless of parameters, the driver will publish the I/O states to the topic `ur_driver/io_states`. This is similar to the old driver, except for the prefix of *ur_driver*.
As a new addition, the driver subscribes to the two topics `ur_driver/joint_speed` and `ur_driver/URScript`. The first topic takes 6 joint speeds and uses the driver's setSpeed method to forward them to the robot. Remember that values should be streamed to this topic faster than 10 Hz to avoid the driver to reset the joint velocities to 0.
The latter topic takes strings and forwards them directly to the robot without any verification. This is intended to send commands for i.e. conveyor tracking and Modbus client interaction, as well as a fast way of moving the robot to verify that connection is working.

The driver also exposes two services; `ur_driver/set_io` and `ur_driver/set_payload`. These works like in the old driver to toggle the controller outputs and to set the payload of the robot.

Finally the driver has an action interface, when the parameter `use_ros_control` is set to false, which is the default value. When a new goal is received, it verifies the input to check that it the values are sane, before transferring it to the driver. For trajectories received via the action interface, the threaded servoj method is used to ensure compatibility with the old driver.

When `use_ros_control` is true, three interfaces are exposed to ros_control; a joint state interface, a force torque sensor interface, and a joint velocity interface. The two first informs ros_control of the joint states as well as the forces and torques at the TCP of the robot, so this can be used in a controller.
The last interface is for controlling the robot. It sends speedj commands to the controller, so the controller configuration parameters are set to run the controller at 125 Hz.

Performance

In this section I will evaluate how my new driver performs. To do this I have made four tests. In the first test, I used the old driver to get a comparison. In the next test, I used my new driver with servoj_time set to 0.08, before another test with servoj_time set 0.008, which should be the optimal value. Finally I did a test using the velocity interface with ros_control. Each test was executed 10 times and the joint values logged. The trajectory in each test was a 180° negative rotation of the end joint followed by a positive 180° rotation. Each rotation should take 2 seconds, resulting in a 4 second trajectory. The results, with all 40 trials, can be seen in Figure 8.

For the controller in ros_control, all joints used a proportional gain of 20 and a differential gain of 0.1. The integral gain was set to 0. These values are chosen as they seem to give stable performance, but more optimal values might give better performance.

Although it can be difficult to see, especially in the position plot, all the trials are actually plotted on top of each other, and that is the first thing to notice; the execution variance for all the different implementation methods is very small. At first it might seem surprising that this is especially true for the one method which uses velocity rather than position for control, but on as it is the only method which uses feedback, it is not that unexpected. Rather, it goes to show that the feedback controller works very well.

Another thing to notice is the actuation delay. When my driver receives a new trajectory, it uploads the threaded URScript to the controller and then executes the trajectory. When the trajectory is done, the URScript finishes. This is done to make it possible to use the touchscreen to move the robot around when it

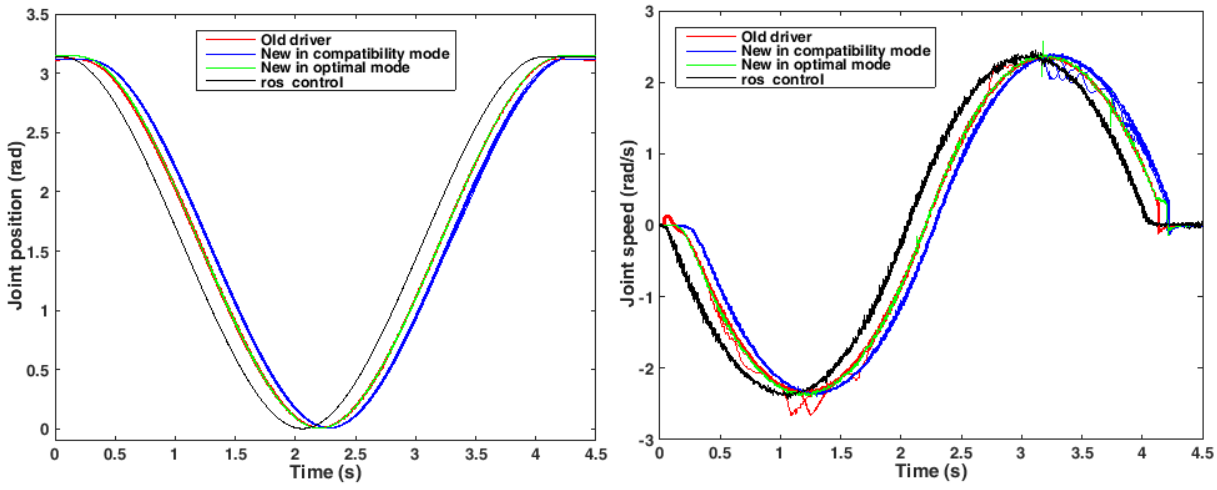

Figure 8: Performance trials a: Joint position. b: Joint velocity

is not executing a trajectory, something that as mentioned previously is not possible with the old driver. But as it can be seen, this adds a small delay, which actually causes my driver to react a bit slower when in compatibility mode. If a user desires performance from my new driver that is similar to the old driver, servoj_time should still be 0.08 seconds, even if that adds to the actuation delay.
The reason being that servoj_time also influences how the driver interpolates between the waypoints in the trajectory, as the interpolator samples the trajectory. A trajectory with waypoints spaced i.e. 0.1 seconds apart, with a velocity of 0 at each waypoint, would be executed very differently if sampled each 0.08 second compared to being sampled every 0.008 second.

If the community wants the new driver to behave exactly like the old, a flag could be added that the URScript program should be uploaded when the driver connect, and not stop the program when the trajectory is done.

The final thing to notice is the trajectory accuracy. Again the ros_control based approach is superior and is the only method where the robot ends up at the desired position. As this method uses feedback, it is expected that the steady state error is zero, but as the other methods are based on positional control, it is still strange that these do not end up at the desired position. The other methods have a steady state error of -1.5° for the old driver and the compatible test, and 0.3° for the test with servoj_time=0.008.
The end of the trajectory's velocity should also be noticed; the velocity based approach is the only one with a smooth deceleration at the end of the trajectory.

While executing the trajectories used in this section, the outputs of the controller were also toggled at random times using the driver. As expected it did not affect the execution it is difficult to show this in any way, but it should be noted by the reader that this part was also tested and worked. Also the automatic reconnection, detection and proper handling of emergency stop as well as protective stops have been verified to work.

Summary

In this chapter, the implementation of the new driver has been described. The best practices found in the previous chapters have been implemented and shown to work as described.

The main class has been described in detail, making it easy to use the driver in any project. As the main use case is with ROS, the driver has been wrapped in a standalone file, which exposes the driver's functionality to ROS, adhering to the standards defined in the ROS ecosystem.

It is also shown that the driver actually performs as expected. In compatibility mode with servoj time set to 0.08sec the actuation delay is around 70ms higher though. Note this is measured on an old controller running firmware 1.8. On newer controllers with firmware version >= 3.1, this will be handled by the lower lookahead time, and the driver will perform just like the old. Setting a low servoj time of 0.08 sec instead of 0.008 sec is only relevant in those rare cases where a trajectory was not optimally defined beforehand, and performance exactly like the old driver is desired. For proper trajectories, which are not affected by the frequency with which the trajectory is sampled, there is no reason to use a servoj time other than 0.008sec.

As expected, the best performance is with speedj control via a controller like ros_control.

Conclusion

In this report I have thoroughly examined the Universal Robot and its controller to design a new driver that can get the optimal performance from the robot, while still maintaining backwards compatibility with legacy code depending on certain performance from the old driver.

In examining the different ways to interface to the robot, I argued that a solution based on URScript is preferable. This is both due to considerations of user friendliness and of maintenance. Starting and running a custom C driver on the controller has shown to be a hassle, especially in a research environment with many different users. That a C program also disables the touch screen and perhaps some of the security features is a further drawback. Maintaining a complete driver written with a poorly documented API, which can only be built on the controller is also a difficult task. As it was later shown, an URScript based approach with speedj can deliver just as good performance as any C based approach in terms of actuation delay, so the choice of using URScript is justified.
The communication interfaces for the robot has also been described, and the possibility to detect the controller firmware version has been added. This was not included in the old driver, which has caused some problems with supporting newer firmware, as there are some breaking changes in the protocol.
The description of the dashboard server has also been included. This is a relatively new addition to the UR support page, and as such its functionality has not been explored nor included in the current version of the driver. This is left as future work.

I have also evaluated the currently available drivers to determine drawbacks, hardware limitations and features that a new driver should support. All the resources exposed by the old drivers are included in the new, and a lot of effort has been put into making the driver work as a plug-in replacement.

A central part of this report is the evaluation of the different ways to control the robot. Different approaches were examined and the performance documented. Based on this it was found that a threaded solution works the best, as it allows for asynchronous and very fast update rates of the target values. While the controller is limited to execute commands at a 125 Hz, the threaded solution makes it possible to update the target pose arbitrarily fast, only limited by the host computer and the network speed.
I also found a way to toggle the output without interrupting the execution of the trajectory using a secondary command. This was tested and verified to work as expected.

Finally I have tested my implementation and verified that it works. The servoj based approach has a slightly larger actuation delay than with the old driver when used with firmware versions lower than 3.1. This is caused by the uploading of a program to the controller as the start of each trajectory. This can be fixed by having a flag that instructs the controller not to stop the script at the end of a trajectory.
As expected the speedj method is superior both in terms of actuation delay and tracking error, as it is the only method that uses feedback.

Future work

As already mentioned, the dashboard server gives some interesting possibilities for enhancing the driver. This includes exposing a service that could recover the robot from a protective stop, and locking the screen when a trajectory is executed, or the robot is controlled by ros_control, so the user doesn't accidently crash

the system. ros_control is always controlling the robot when the controller is running, so moving the robot with the teach pendant will cause that input to struggle against ros_control, which doesn't get the new target pose. The resulting conflicting motion will cause the controller to crash.

Another enhancement that I have already touched lightly upon is making it possible to upload the servoj threaded program to the controller and have it running continuously. This would enhance the backwards compatibility, but does not add any value to new implementations, that should be based on ros_control due to its superior performance.

Finally the velocity based controller could be made to a threaded solution. I have shown that there is a small performance improvement to get from that.

Works Cited

[1] O. Ravn, N. A. Andersen and T. T. Andersen, "UR10 Performance Analysis," Technical University of Denmark, 2014.

Appendix A – Pseudo URScript for threaded servoj command

```
def driverProg():
  HOSTNAME = "%(driver_hostname)s"
  PORT = %(driver_reverseport)d
  MULT_jointstate = 10000.0
  MULT_time = 1000000.0
  SERVO_IDLE = 0
  SERVO_RUNNING = 1
  cmd_servo_state = SERVO_IDLE
  cmd_servo_q = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
  cmd_servo_dt = 0.0
  def set_servo_setpoint(q, dt):
    enter_critical
    cmd_servo_state = SERVO_RUNNING
    cmd_servo_q = q
    cmd_servo_dt = dt
    exit_critical
  end
  thread servoThread():
    state = SERVO_IDLE
    while True:
      # Latches the new command
      enter_critical
      q = cmd_servo_q
      dt = cmd_servo_dt
      id = cmd_servo_id
      do_brake = False
      state = cmd_servo_state
      cmd_servo_state = SERVO_IDLE
      exit_critical
      # Executes the command
      if state == SERVO_RUNNING:
        servoj(q, 0, 0, dt)
      else:
        sync()

  socket_open(HOSTNAME, PORT)
  thread_servo = run servoThread()

  while True:
  mtype = ll[1]
  if mtype == MSG_SERVOJ:
    # Reads the parameters
    params_servoj = socket_read_binary_integer(6+1)
    # Unpacks the parameters
    q = [params_servoj[1] / MULT_jointstate, params_servoj[2] / MULT_jointstate,
         params_servoj[3] / MULT_jointstate, params_servoj[4] / MULT_jointstate,
         params_servoj[5] / MULT_jointstate, params_servoj[6] / MULT_jointstate]
    t = params_servoj[7] / MULT_time
    set_servo_setpoint(q, t)

```