YuMi low-level motion guidance using the Julia programming language and Externally Guided Motion Research Interface

Bagge Carlson, Fredrik; Haage, Mathias

2017

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

YuMi low-level motion guidance using the Julia programming language and Externally Guided Motion Research Interface

Fredrik Bagge Carlson
Mattias Haage

LUND
UNIVERSITY

Department of Automatic Control

Technical Report TFRT-7651
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

A generic robot library written in the Julia programming language is presented. A bridging framework, also presented, exposes the ABB externally guided motion research interface (EGMRI) low-level robot motion correction interface to the Julia language and other entities, such as Python and ROS. A usage example on Julia contact force estimation on an ABB YuMi robot concludes the document.

1. Introduction

This document outlines a set of software tools written to facilitate general robot controller development in the Julia programming language. In particular, we make use of this framework to implement controllers for ABB robots and also for connecting to the ROS robot middleware. ABB provides the communication and control interfaces Externally Guided Motion (EGM) and its relative EGM-Research Interface (EGMRI)[1] for sensor-based control of their industrial robots. The term sensor can be freely interpreted and incorporates, e.g., a computer. The framework detailed in this report makes use of EGMRI to read measurements from the robot and update setpoints for position, velocity and torque signals.

2. Software

The software is provided in three git repositories. This section explains the contents of each repository. Further details are provided in the README file of respective repository.

2.1 Robotlib.jl

Robotlib.jl [Bagge Carlson, 2015] is an open-source package for the Julia programming language containing functions and algorithms useful in the robotics field. Below is a list of current content.

- Forward kinematics (Product of Exponentials (POE) and Denavit-Hartenberg (DH) formulations)
- Inverse kinematics (iterative methods)
- Jacobians (POE/DH)
- Calibration (using force sensing and/or laser scanning [Bagge Carlson et al., 2015a], POE and local POE methods [Chen et al., 2001])
- Model estimation (position-dependent friction [Bagge Carlson et al., 2015b], gravity)
- Frame construction (projection and line/curve/plane fitting utilities for optical tracking systems)
- Orientation representations
- Kinematics representations
- Logging
- Plotting

Further reading is available at the Robotlib.jl GitHub repository:
`https://github.com/baggepinnen/Robotlib.jl`

2.2 Robotlab.jl

Robotlab.jl is a repository containing device-specific algorithms and methods written in the Julia language. As such it complements the generic Robotlib.jl package. The YuMi part of the package support contact force estimation from motor torques. Further reading is available at the Robotlab.jl git repository:
`https://gitlab.control.lth.se/cont-frb/Robotlab.jl.git`

[1] EGMRI is only available on ABB YuMi at the time of writing

2.3 DynamicMovementPrimitives.jl

Basic functionality for estimation of Dynamic Movement Primitives (DMPs) is provided in the package DynamicMovementPrimitives.jl [Bagge Carlson and Karlsson, 2016]. This package also implements the method from [Karlsson et al., 2017]. Further reading is available at the DynamicMovementPrimitives.jl github repository:

`https://github.com/baggepinnen/DynamicMovementPrimitives.jl`

2.4 Bridge

The bridge is a piece of software written in C++ that connects the ABB YuMi robot and/or other equipment to the Julia language, and other entities (currently to Python and ROS). The bridge implements a row-based network transfer format well-suited to expose traffic from several sources fused into a callback with a simple custom domain-specific parameter list. Further reading is available at the bridge git repository:

`https://git.cs.lth.se/mathias/labcomm2egmri`

3. Installation

Installation is done by following instructions available at the respective repository. Installation of Julia packages requires an installation of the Julia language of specified version. The bridge require checkout of a git submodule before compilation.

3.1 Julia installation notes

A Julia package is installed using the built-in package manager. If the package is registered in the official Julia package repository, the command for installation is

```
julia> Pkg.add("PackageName")
```

Unregistered packages are installed by cloning the repository, e.g.,

```
Pkg.clone("https://gitlab.control.lth.se/cont-frb/Robotlab.jl.git")
```

Packages are imported into the current Julia session with the command

```
using PackageName
```

Robotlib.jl is registered as an official Julia package and can be installed with the command `Pkg.add("Robotlib")`.

Robotlab.jl is not registered as an official package, and must therefore be installed with the command

```
Pkg.clone("git@gitlab.control.lth.se:cont-frb/Robotlab.jl.git")
```

To have Robotlab.jl loaded automatically every time Julia starts, add the line

```
using Robotlab
```

in the .juliarc file. For more features, see the package Robotlib.jl, use it by typing

```
using Robotlib
```

For help, type

```
?Robotlib.
```

3.2 Bridge installation notes

Installation requires the installation and compilation of a submodule. Typically:

```
git clone https://git.cs.lth.se:mathias/labcomm2egmri.git
cd labcomm2egmri
git submodule init
git submodule update
```

This is followed by compilation of the submodule and then the main code base.

3.3 Paths

Both Julia and the bridge need environment variables to function properly.

/home/user/.bashrc For the bridge a number of environment variables need to be set, on Ubuntu preferably in a .bashrc file:

```
# For labcomm2egmri
export LABCOMM_HOME = __YOUR_PATH__/labcomm2egmri/submodules/labcomm
export LABCOMM_C_INCLUDE=$LABCOMM_HOME/lib/c/2014
export LABCOMM_C_LIB=$LABCOMM_HOME/lib/c
export PATH = $PATH:__YOUR_PATH__/labcomm2egmri/bin
export LD_LIBRARY_PATH = $LABCOMM_C_LIB:__YOUR_PATH__/labcomm2egmri/bridge
```

/home/user/.juliarc The Julia client scripts are located in `/home/user/.julia/v0.6/Robotlab/src/connections`.

```
using Plots
plotlyjs() # plotlyjs is the recommended backend for plotting. If you want
↪  to plot in realtime, use gr
# gr()
using Robotlab
ENV["BRIDGE\_PATH"] = ".../labcomm2egmri/bridge/" # This is the path to the
↪  labcomm2egmri folder that contains the julia_client .so file
```

4. Robotlib notes

4.1 Stateful operations

Functions that require introduction of a state, such as integration, are provided in the form of macros. When the code of the macro is parsed, state variables are introduced behind the scenes to facilitate the stateful calculations needed by the function. A macro can be called with either of the syntaxes

```
retval = @macroname arg1 arg2 arg3...
retval = @macroname(arg1, arg2, arg3...)
```

Integration is handled by the macro
`integral = @integrate signal init`.
This macro creates a state variable with the size determined by `init`. Example:
`torque_integral = @integrate torque zeros(n_joints)`

Differentiation is handled by the macro
`diff = @differentiate signal init`.
This macro creates a state variable with the size determined by `init`. Example:
`torque_diff = @differentiate torque zeros(n_joints)`

Exponential filtering is handled by the macro
`filt = @expfilt signal alpha init`.
This macro creates a state variable with the size determined by `init`. Example:
`torque_filt = @expfilt torque 0.99 zeros(n_joints)`.
The value of $\alpha \in (0, 1)$ determines the amount of filtering (higher is smoother). The time constant of the filter is approximately $\frac{1}{250(1-\alpha)}$ seconds.

4.2 Save and read logs

A binary logfile is automatically saved with a period of `save_period` (defined in the client files). When the bridge is terminated, the file is once again saved. The names of these files are `logleft` or `logleft`. If you want to save the log in another format, you may use the `savelog` function. Options include .jld files[2] (recommended), .mat and .csv. You may read a saved logfile using `readlog`. To read a logfile produced by `orca_log`, see Robotlib.jl functions `orcalog2mat, getData`.

Log struct Robotlab.jl provides a custom log struct that contains the data recorded during an experiment. The data structure has one field for each signal that is logged. An example is shown below

```
type Log{T} <: AbstractLog
    pos         ::Matrix{T}
    vel         ::Matrix{T}
    trq         ::Matrix{T}
    pos_o       ::Matrix{T}
    vel_o       ::Matrix{T}
    trq_o       ::Matrix{T}
    timestamp   ::Vector{Float64}
    seqnum      ::Vector{Int}
    length      ::Int
    n           ::Int
end
```

The data structure is parameterized with type parameter `T` to allow for different floating point precision etc. Data is added to the log using the command `push!`, with a type signature corresponding to the fields of the log type, e.g., `push!(log, pos, vel, trq, current_time, tm)`. Internally, the log object holds preallocated arrays which grow automatically when filled. To implement a new log type, make sure the type is a subtype of `AbstractLog` and implement a constructor. User defined log-types must contain the field `length` or override the functions `length` and `capacity`. The `push!` method for the user defined type is automatically generated and the call signature will follow the order of the vector and matrix fields of the type. `Matrix` fields expect an `AbstractVector` as input whereas `Vector` fields expect a scalar. All log types are expected to follow the convention for vector valued signals, each row in the matrix storage is a signal, each column is a vector valued sample. This is motivated by Julia's column-major storage convention.

4.3 General plotting

Plotting for Julia is provided by several packages. We recommend using the package Plots.jl, which provides a common interface to a large number of backends.
Example of plotting syntax using Plots.jl

```
x = linspace(0,10)
y = sin(x)
plot(x,y,xlabel="x", ylabel="y", title="Title", c=:red, linewidth=2,
↪  label="legend")
```

Plotting a signal from the log object is simple

```
plot(log.pos', label = ["Position $i" for i in 1:n_joints]')
```

Note that matrices in the log object have to be transposed before plotting.
To plot in several windows, you may use any of the following

[2] https://github.com/JuliaIO/JLD.jl

Algorithm 1 How to define a custom log type.

```
#How to define a custom log:
@extend mutable struct MyLog <: AbstractLog
    mymatfield::Matrix{Float64} # Matrices are suitable for storing vector
    ↪  valued data
    myvecfield::Vector{Float64} # Vectors are suitable for storing scalar data
end

#You can now easily push data to the log
mylog = MyLog(n_joints, initial_capacity)
push!(mylog, pos,vel,trq, pos_o, vel_o, trq_o,start_time, tm, myvec, myscalar)
# Note that the fields `log, pos,vel,trq, pos_o, vel_o, trq_o,start_time, tm`
↪  are inherited from the standard `Log`
# You also need to define the constructor for your log type.
```

Algorithm 2 Custom log type example. We now define a custom log type that holds an additional field with estimated forces, $fext$.

```
# Define a type that extends AbstractLog, it will automatically inherit all
↪  fields from standard Log
@extend mutable struct ForceLog  <: AbstractLog
    fext::Matrix{T} #Be sure to annotate the field. The estimated wrench is a
    ↪  vector, so we need a matrix to store it.
end
ForceLog{T}(::Type{T},n,s=250*1000) =
↪  ForceLog{T}(zeros(T,n,s),zeros(T,n,s),zeros(T,n,s),zeros(T,n,s),
↪  zeros(T,n,s),zeros(T,n,s),zeros(Float64,s),zeros(Int,s), 0,n,zeros(T,6,s))
# The only thing that differs between the constructor for ForceLog and regular
↪  Log is the last array

const log = ForceLog(Float64, n_joints, 250*500) # You instantiate the new log
↪  type in the same way as the standard Log
```

```
plot(log.pos', title = ["Position $i" for i in 1:n_joints]' layout = n_joints)
plot(log.pos', title = ["Position $i" for i in 1:n_joints]' layout =
↪  (1,n_joints))
```

See Plots documentation for more plotting details.

4.4 Robotlib plotting

A few special plotfunctions are made available in Robotlib. Those include

```
trajplot(T) # T in R(4,4,N)
trajplot3(T)
plot3smart(x,args...) # Makes a 3d plot of a matrix
plot(f::Frame)
plot(l::Line)
plot(p::Points)
```

5. Force estimation

Primitive force estimation is provided with the help of the function `gravmodel`, which is a combined friction and gravity model. The controller files in yumi_controller_forceest illustrates how this function is used for force estimation. To replace the gravity model, run

the code in FRIDAestimateGravity.jl and set the path to an appropriate logfile. Save the parameter vector k as a binary file using

```
serialize(filename::String, k)
```

Make sure this is the file loaded before the function `gravmodel` in the file `globalutils.jl`.

Estimation of external forces is implicitly provided through functions to estimate and evaluate a gravity model and detailed friction models [Bagge Carlson et al., 2015b]. These methods are provided in Robotlib.jl Bagge Carlson (2015). The quality of the force estimate is critically dependent on these models, and especially the friction model should be estimated on the individual robot on which the estimations is to be carried out.

The force estimation considered neglects robot inertial forces by making the following assumption

$$\tau = G(q) + F_f(q, \dot{q}) + J(q)^\mathsf{T} F_{ext}$$

hence, the estimation will be inaccurate in the presence of large accelerations and/or velocities.

We introduce the notation $\tau_{ext} = J^\mathsf{T} F_{ext}$, where $J = J(q) \in \mathbb{R}^{6\times n}$ and n is the number of joints. If $n > 6$, this system of equations is over determined and we can solve for

$$\hat{F}_{ext} = \arg\min_F \left\| \tau_{ext} - J^\mathsf{T} F \right\|_2^2 = \left(J J^\mathsf{T} \right)^{-1} J \tau_{ext} \tag{1}$$

If prior information about F_{ext} is available, e.g., contact torques are known to be small etc., we can easily incorporate a Gaussian prior on $F_{ext} \sim N(0, C_F)$. We can also incorporate our knowledge of the variances in $\tau_{ext} \sim N(0, C_\tau)$. The optimal $\hat{F}_{ext}$ is then found as [3]

$$\hat{F}_{ext} = \arg\min_F \left\| \tau_{ext} - J^\mathsf{T} F \right\|_{C_\tau}^2 + \left\| F \right\|_{C_F}^2 = \left(J C_\tau J^\mathsf{T} + C_F^{-1} \right)^{-1} J C_\tau \tau_{ext} \tag{2}$$

The covariance matrix of the joint torques, C_τ, can be seen as a design variable. It is possible, for instance, to encode the fact that uncertainty is high due to stiction whenever a joint is standing still. Models that capture this behavior are, e.g., either of

$$C_\tau(\dot{q}) = (C_0 - C_\infty) e^{-\gamma |\dot{q}|} + C_\infty \tag{3}$$

$$C_\tau(\dot{q}) = \begin{cases} C_0 & \text{if } |\dot{q}| < \dot{q}_0 \\ C_\infty & \text{if } |\dot{q}| \geq \dot{q}_0 \end{cases} \tag{4}$$

These models are visualized in fig. 1.

For additional information about noise modeling and redundancy resolution for the purpose of force estimation, see [Linderoth et al., 2013; Wahrburg et al., 2016].

5.1 Further variance reduction

The error in the force estimate consists of three components, random noise, periodic torque ripple [Bagge Carlson et al., 2017] and systematic errors due to static friction. Since the final error along any dimension is a linear combination of n measurements, modeling the resulting error as a Gaussian random variable is a reasonable choice due to the central limit theorem, which is backed by [Wahrburg et al., 2016]. Since the error due to static friction is systematic, low-pass filtering of the signal *does not* guarantee a lower variance estimate. In [Linderoth et al., 2013], this error source was modeled explicitly, which leads to a convex estimation problem that can be solved efficiently in a iterative fashion. The periodic disturbances due to torque ripple were modeled and analyzed in Bagge Carlson et al. (2017).

[3] Note the closed-form expression for the optimal estimate $\hat{F}_{ext}$ in (2).

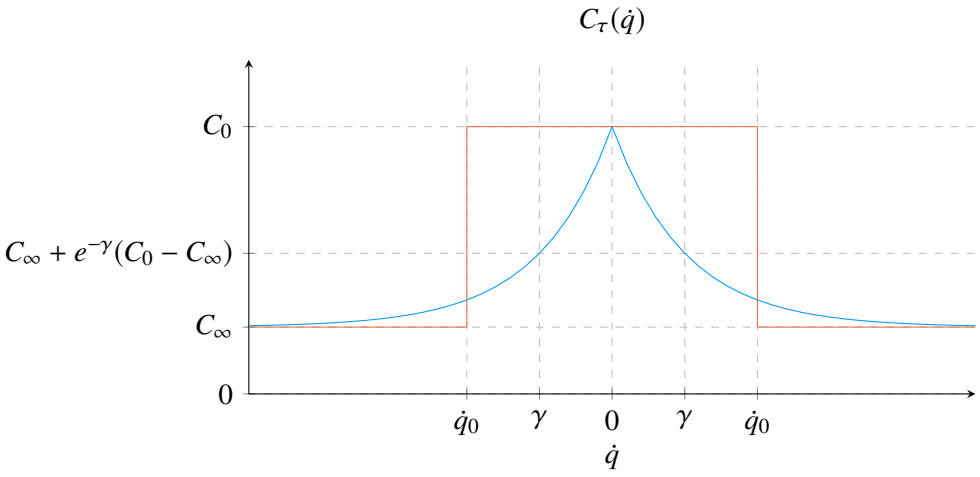
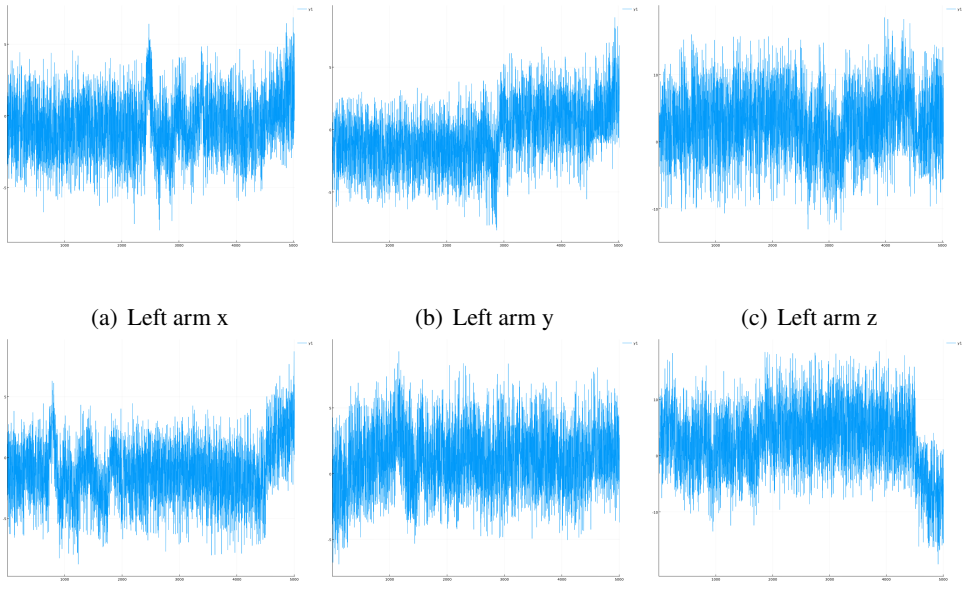


Figure 1 Two different choices of $C_\tau(\dot{q})$, eq. (3) in blue, eq. (4) in orange.



(a) Left arm x (b) Left arm y (c) Left arm z

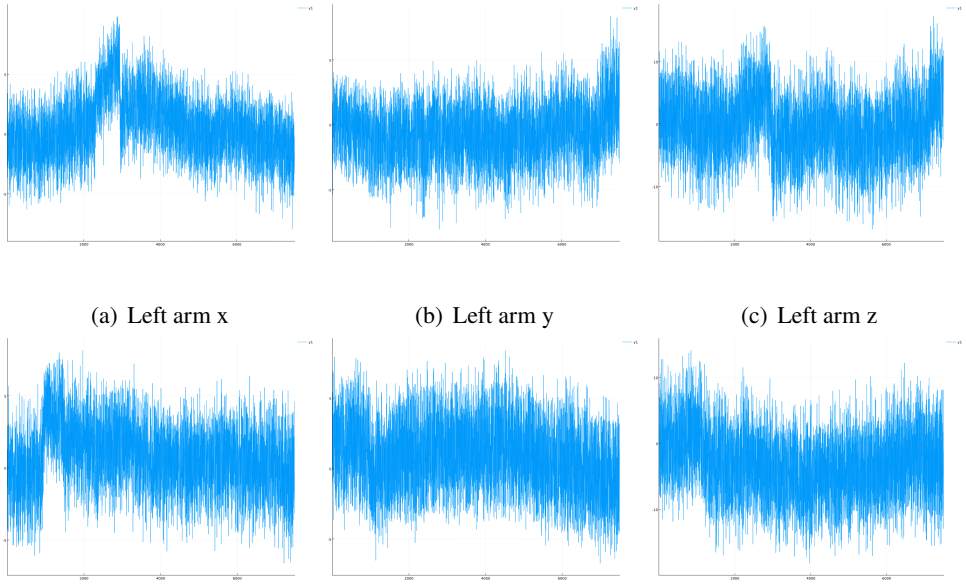
(d) Right arm x (e) Right arm y (f) Right arm z

Figure 2 Directional coordinates of force wrench for tip manipulation experiment.

5.2 Qualitative evaluation

Two initial experiments are reported as an indication of the performance of the force estimation algorithms on the platform. In the first experiment the tip of the right hand is pressed by the operator in the X, Y and Z robot base directions followed by a repetition for the left hand. The robot arms are in home pose. The three directional coordinates of the estimated force wrench are plotted for the left and right arm in fig. 2. The pressure is distinctly detected in directions less affected by weak joints close to the wrist (2a and 2d). It is less distinctly detected in directions involving weak joints (2c and 2f). Fig. 2b is probably showing a hysteresis effect stemming from friction. The last portion of the figures, most prominently displayed in 2d and 2f, is after shutdown of the running RAPID program.

In the second experiment, approximately 500g of pressure was applied in the x and z directions, see fig. 3. Arms are still in home pose. The X direction (not involving weak joints) report force directional magnitude approximately on the same order as the applied pressure.



(a) Left arm x (b) Left arm y (c) Left arm z

(d) Right arm x (e) Right arm y (f) Right arm z

Figure 3 500g pressure experiment.

The Z direction is less distinct.

6. YuMi

This section provides instructions on how to use the framework to control ABB robots through the EGMRI interface.

6.1 Registering controller callback function

The controller is implemented in a function which is registered in the bridge program as a callback. This function is called repeatedly with relevant measurement signals as input. The control signal is sent using a call to `labcomm_send` from within this function. This function also implements logging and updating of state variables etc. See algorithm 3 for an example.

6.2 Control using EGMRI

Implement your controllers in the controller functions in the client files, located in `Robotlab/src/connections`. The controller function receives three vectors of size `n_joints`, those vectors are copies and you are thus free to modify them. The controller function should return three vectors, `pos_o, vel_o, trq_o`. It's okay to give only a position reference and set the others to zero, if you want to comman velocity references, you have to supply an integrated version as position reference. Type `?@integrate` to see how the integrator macro works. You may also find the vector `pos0` useful, this is the position of the robot when the client program started. When you're done with your controller, do the following:

1. Set robot in auto mode with motors off (pressing the three bar button on the teach pendant a couple of times)
2. Run the EGMRI RAPID program (PP to main first).
3. Run bridge program located in `/labcomm2egmri/bridge/bridge`

Algorithm 3 Example of code that opens the labcomm connection, registers a controller callback function and starts the labcomm server.

```
function handler(tm::Cint, pos_p::Ptr{Float64}, vel_p::Ptr{Float64},
↪  trq_p::Ptr{Float64})::Cint
    start_time = time()
    pos        = SVector(ntuple(i->unsafe_load(pos_p,i), n_joints))
    vel        = SVector(ntuple(i->unsafe_load(vel_p,i), n_joints))
    trq        = SVector(ntuple(i->unsafe_load(trq_p,i), n_joints))

    pos_o, vel_o, trq_o = controller(pos,vel,trq,t)
    push!(log, pos,vel,trq, pos_o, vel_o, trq_o,start_time, tm)
    ccall((:labcomm_send,labcommbridgepath),Void,
    ↪  (Cint,Cint,Ptr{Float64},Ptr{Float64},Ptr{Float64}), tm, pos_o, vel_o,
    ↪  trq_o)

    (global last_time = start_time)::Float64
    (global t += h)::Float64
    (global seq_num += 1)::Int
    gc()
    zero(Cint)
end

precompile(handler, typetuple(handler))
const handler_c = cfunction(handler,Cint,typetuple(handler))
ccall((:labcomm_register_retrieve_callback_v1,labcommbridgepath), Void,
↪  (Ptr{Void},), handler_c)
gc_enable(false); gc()

const hostname = "localhost"
const portno = 7511 # 7511: left arm, 7512: right arm
ccall((:labcomm_start,labcommbridgepath), Void, (Ptr{UInt8},Cint), hostname,
↪  portno)
```

4. Run Julia client program, i.e., `julia_left_clent.jl`. Wait until it says `C connection is active`
5. Trigger `EGMRI_TRIGGER` signal under digital inputs on the teach pendant.
6. When you are done, kill the bridge program (ctrl-C), the Julia program will now save the log with the filename `logleft` or `logright`
7. The Julia promt that is left is loaded with all you need, the data can be accessed through the log object. type `log.` followed by TAB to see which signals are available.

6.3 Log during leadthrough

1. Set robot in manual mode with motors off (pressing the two bar button on the teach pendant a couple of times does the trick)
2. Run the EGMRI RAPID program (PP to main first).
3. Run bridge program located in `/labcomm2egmri/bridge/bridge`
4. Trigger `EGMRI_TRIGGER` signal under digital inputs on the teach pendant. You have to be in signal simulation mode for this to work. If the robot craps out when you trigger the signal, start over. It's probably due to the robot not being in the home position.

5. Press stop on the teach pendant
6. Enable leadthrough on the teach pendant.
7. Run Julia client program, i.e., `julia -qi julia_left_clent.jl` (`q` for quiet (no startup banner), `i` for interactive, so that the program ends with a Julia prompt).
8. Perform lead-through.
9. When done, kill the bridge program (ctrl-C), the Julia program will now save the log with the filename `logleft` or `logright`
10. The Julia prompt that is left is loaded with all you need, the data can be accessed through the `log` object. type `log.` followed by TAB to see which signals are available.

When restarting the procedure, it's important to kill the program on the teach pendant entierly, i.e., go to motors off. Otherwise the robot will complain loudly (!)

See https://git.cs.lth.se/mathias/labcomm2egmri for more info regarding connecting to the robot and finding the RAPID programs needed etc.

References

Bagge Carlson, F. (2015). *Robotlib.jl*. Dept. Automatic Control. URL: https://github.com/baggepinnen/Robotlib.jl.

Bagge Carlson, F., R. Johansson, and A. Robertsson (2015a). "Six DOF eye-to-hand calibration from 2D measurements using planar constraints". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. URL: http://lup.lub.lu.se/record/7613773.

Bagge Carlson, F. and M. Karlsson (2016). *Dynamicmovementprimitives.jl*. Dept. Automatic Control. URL: https://github.com/baggepinnen/DynamicMovementPrimitives.jl.

Bagge Carlson, F., A. Robertsson, and R. Johansson (2015b). "Modeling and identification of position and temperature dependent friction phenomena without temperature sensing". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. URL: http://lup.lub.lu.se/record/7613758.

Bagge Carlson, F., A. Robertsson, and R. Johansson (2017). "Linear parameter-varying spectral decomposition". In: *2017 American Control Conference (ACC)*. URL: http://lup.lub.lu.se/record/ac32368e-e199-44ff-b76a-36668ac7d595. Accepted.

Chen, I.-M., G. Yang, C. T. Tan, and S. H. Yeo (2001). "Local poe model for robot kinematic calibration". *Mechanism and Machine Theory* **36**:11, pp. 1215–1239.

Karlsson, M., F. Bagge Carlson, A. Robertsson, and R. Johansson (2017). "Two-degree-of-freedom control for trajectory tracking and perturbation recovery during execution of dynamical movement primitives". In: *20th IFAC World Congress*. Accepted.

Linderoth, M., A. Stolt, A. Robertsson, and R. Johansson (2013). "Robotic force estimation using motor torques and modeling of low velocity friction disturbances". In: *Intelligent Robots and Systems (IROS), International Conference on*. IEEE/RSJ. URL: http://lup.lub.lu.se/record/4191947.

Wahrburg, A., A. Robertsson, B. Matthias, F. Dai, and H. Ding (2016). "Improving contact force estimation accuracy by optimal redundancy resolution". In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. URL: www.researchgate.net/publication/310951509.