Variable mechanical stiffness robotics arm testbed

Matthieu Prigent

March 10, 2025

Abstract

This document aims to summarize what was done in the development and building of testbed 2, and what was learned from it, for further development with testbed 3.

Contents

1 Introduction

This document aims to summarize what was done in the development and building of testbed 2, and what was learned from it, for further development with testbed 3. All of the documents related with this report, which include the software for running the arm and the can be found on this git repo and this drive. The Onshape model can be found here.

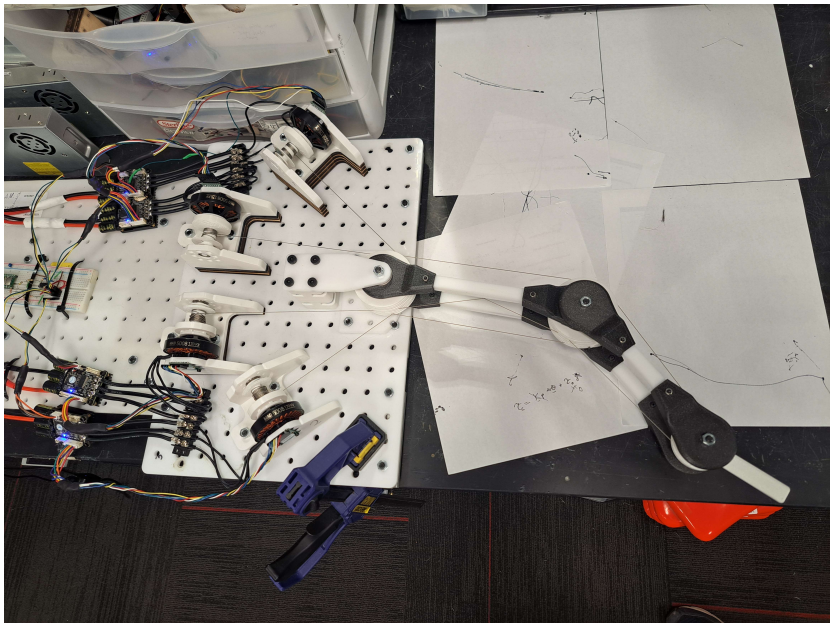

Figure 1: Overview of the testbed.

On the previous figure 1, we can see the testbed, with the main micro controller (partially cut on it) on the far left, then the 4 motor drivers linked throught the CAN bus, the 4 motor pods, and then the arm with its 3 joints.

2 Mechanical design

2.1 cable routing

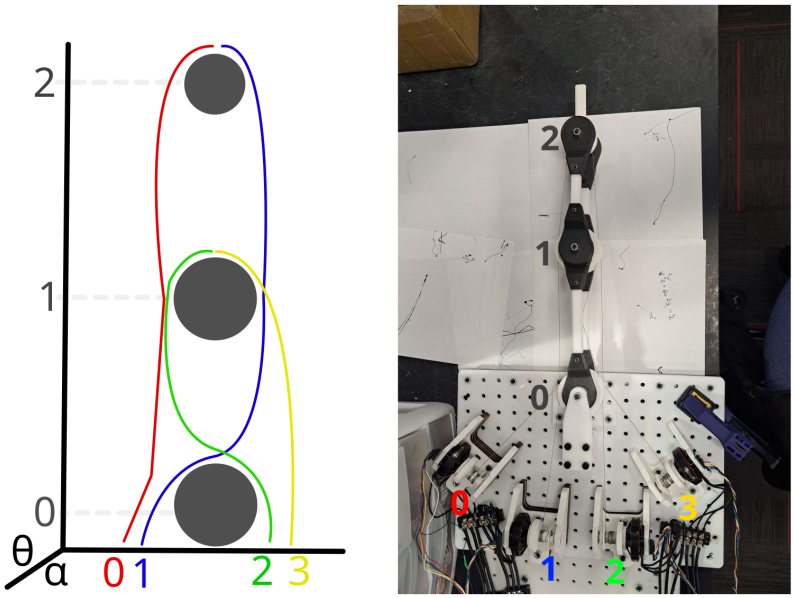

Figure 2: Routing of the cables in the arm.

The routing of the cables is done as shown on 2. This geometry was choose for it's high degree of symmetry and hence easier control implementation features. We call the angle of each of the motors that are pulling a cable α, going from 0 to 3. We call the angle of each of the joints from the arm ϑ, going from 0 to 2. On the actual arm, the cable linked to the motor 0 is on the top pulleys, while the cable linked to the motor 3 is on the bottom pulley. Based on 2, we get that the matrix of the direction of the arm, going from $\alpha$to ϑ, is :

$$P_0 = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{pmatrix} \tag{1}$$

Now, taking into account the diameter of the pulleys, calling d_m the diameter of the motor pulleys and D_0, D_1 and D_2 the respective diameter of the 3 joints, we can rewrite the matrix while taking into account the reduction ratio :

$$P = d_m \begin{pmatrix} \frac{1}{D_0} & \frac{1}{D_0} & -\frac{1}{D_0} & -\frac{1}{D_0} \\ \frac{1}{D_1} & -\frac{1}{D_1} & \frac{1}{D_1} & -\frac{1}{D_1} \\ \frac{1}{D_2} & -\frac{1}{D_2} & 0 & 0 \end{pmatrix} \tag{2}$$

2.2 sizing of the arm

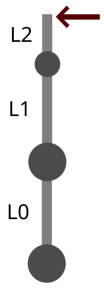

Figure 3: Size of the different links of the arm, and direction of the desired force capability of the arm.

To get a certain area for the arm the be able to reach, we arbitrary choose the length of the links so that the arm is about 45cm long. We will name the length of the links L_0, L_1 and L_2. get a certain force capability F at the output of the arm, we need to size the pulleys accordingly, knowing the defined length. We will name the torque capabilities of each joint τ_0, τ_1 and τ_2. Finally, the torque of the motors will be named τ_{mi}.
First, we can write relation between τ_{mi} and the τ_i.

$$\frac{1}{d_m}\begin{pmatrix} D_0 & D_0 & -D_0 & -D_0 \\ 0 & 0 & D_1 & -D_1 \\ D_2 & -D_2 & 0 & 0 \end{pmatrix}\begin{pmatrix} \tau_{m0} \\ \tau_{m1} \\ \tau_{m2} \\ \tau_{m3} \end{pmatrix} = \begin{pmatrix} \tau_0 \\ \tau_1 \\ \tau_2 \end{pmatrix} \tag{3}$$

We can see a major difference between this matrix and the kinematics matrix seen above, which is that the motors 0 and 1 do moves when the second joint is rotated, but do not contribute to the torque on the second joint. This result is quiet unintuitive, but can be verified experimentally on the testbed. To get the maximum torque of each joint, we use this relation and specify that the motors pulling in the desired direction are at τ_{mmax}, and the ones from the other side are at the base co-activation τ_{mmin}.

$$\frac{2D_0}{d_m}(\tau_{mmax} - \tau_{mmin}) = \tau_0 \tag{4}$$
$$\frac{D_1}{d_m}(\tau_{mmax} - \tau_{mmin}) = \tau_1 \tag{5}$$
$$\frac{D_2}{d_m}(\tau_{mmax} - \tau_{mmin}) = \tau_2 \tag{6}$$

Which, making the hypothesis that the co-activation is small compared to τ_{mmax} get us to :

$$\frac{2D_0\tau_{mmax}}{d_m} = \tau_0 \tag{7}$$
$$\frac{D_1\tau_{mmax}}{d_m} = \tau_1 \tag{8}$$
$$\frac{D_2\tau_{mmax}}{d_m} = \tau_2 \tag{9}$$

We need, if we desire to keep the arm in an equilibrium, to verify

$$F = \frac{\tau_2}{L_2} = \frac{\tau_1}{L_2 + L_1} = \frac{\tau_0}{L_2 + L_1 + L_0} \tag{10}$$

Combining these equations and the precedent reasoning on the arm, we finally get that the diameters of the pulleys must verify :

$$D_2 = \frac{Fd_m L_2}{\tau_{mmax}} \tag{11}$$

$$D_1 = \frac{Fd_m (L_2 + L_1)}{\tau_{mmax}} \tag{12}$$

$$D_0 = \frac{Fd_m (L_2 + L_1 + L_0)}{2\tau_{mmax}} \tag{13}$$

And we see that all of the 3 pulleys are proportional to the diameter of the motor pulleys. The motor pulleys diameter can be choose arbitrarily, but its final value will change the tension of the cables. Bigger motor pulleys will mean relieving the cables from some stress, but also that the value of the D_i will be bigger, which can be inconvenient for the size of the arm. The final choice was to made these pulley quiet small.

Now we know that τ_{mmax} is about 1N.m for a motor, that $L_0 = 21cm$, $L_1 = 16cm$, $L_2 = 9cm$ and that $d_m = 1.4cm$. The target force that we want is about 16N of force. We could go for a bigger force and hence reduction ratio, but that would lead to a high refleted inertia of the motor to the arm). With all of that, we get that $D_2 = 2cm$, $D_1 = 5.6cm$ and $D_0 = 5.2cm$. These value were in the end rounded to $D_2 = 3cm$, $D_1 = 6cm$ and $D_0 = 6cm$, ensuring that we get the desired force capability.

2.3 Building of the testbed

The testbed was designed entirely on Onshape, the project can be found here. The testbed was build using mostly 3D printed PLA, and lasercutting of Delrin and Plexiglas for some parts (using the laser cutter from lab64). This was because of the convenience of using already mastered tools for me, instead of going for more rigid aluminum. I would say it's fine for what we are doing with this testbed right now.

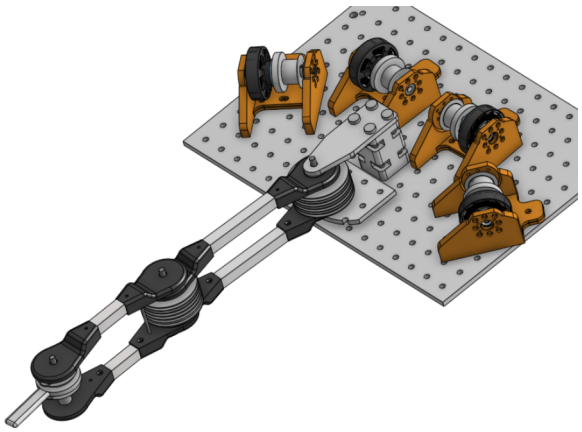

Figure 4: Onshape model.

2.4 What to improve mechanically for testbed 3

- Design a structural link between the top and the bottom part of the arm to improve it's rigidity by a lot (see the blue lines on 5).
- Switch to aluminum for parts that handle a lot of load and can bend due to it (You can use sendcutsend or JLCCNC for machining them if necessary, it is not that expensive if the part is designed properly). Some of the parts that were bending a lot were especially on the motor side, like the main plate that was made out of flexible Plexiglas, and the motor pulley as well. For the motor pulley, putting a Dowel pin in it may do the trick for making it more rigid.

- Switch to a spiral drum and spiral pulleys to prevent cables from ripping against each other. The current smooth pulleys and drums lead to some undesired behaviors, long term tear off of the cable and awful noises. This may be a not so good idea if the arm pulleys ends up being taller, making the arm too tall compared to its width.
- switch from steel cables to dyneema cable.
- We can merge two of the pulleys from the second joint (the ones circled in red on 5) into one, as their movement are already linked anyway. This reduce the part count and complexity of the arm by a bit. What to get from this is that one pulley per cable is not necessary if adjacent pulleys ends up having a linked rotation in the same direction anyway.

The next mechanical challenges to solve on testbed 3 nest in how to run the cables in a non-planar arm.

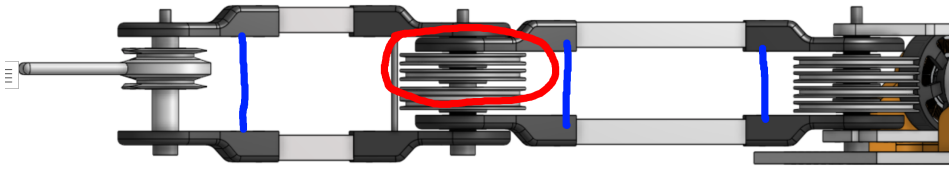

Figure 5: Sideview of the arm on Onshape.

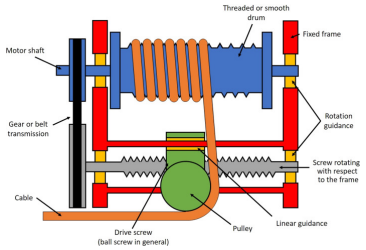

Figure 6: An example of an interesting way to make a spiral drum. Doing it this way can also enable a permanent pretensioning of the cables by adding a spring in the system.

3 Electrical design

3.1 General architecture

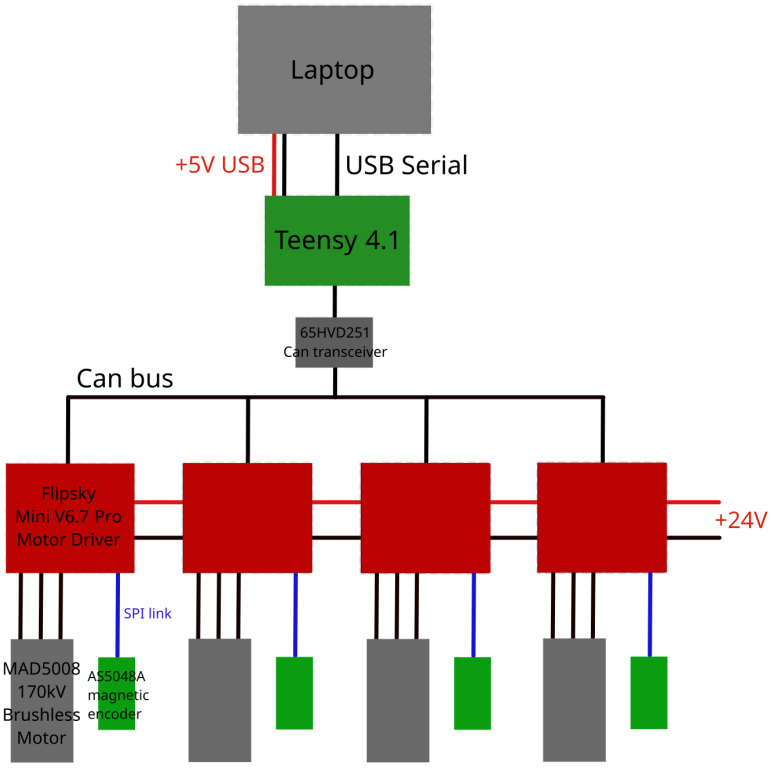

Figure 7: Architecture of the electronics of the arm.

To power on the testbed, you need to plug in the two 24V power supplies that power the motor drivers, and then plug in the usb cable coming from the Teensy to your computer.

3.2 references of the components

- Teensy 4.1 (a great and powerful microcontroller)
- SN65HVD251 Can tranceiver
- FLIPSKY Mini FSESC6.7 PRO motor driver
- 5008 EEE 170kV MAD brushless motor
- AS5048A magnetic encoder

3.3 The motor drivers

One of the main wrong choice that was made was to use Flipsky motor drivers (that are originally meant for powering skateboard motors). They indeed can drive brushless motors with field-oriented control, but this choice led to needing to change the resistors on the motor drivers, and unsoldering them is a painful process to do (and is risky for the driver, you can damage it doing so). For better current capabilities, it was needed to switch from 0.005Ω resistors to 0.05Ω ones.

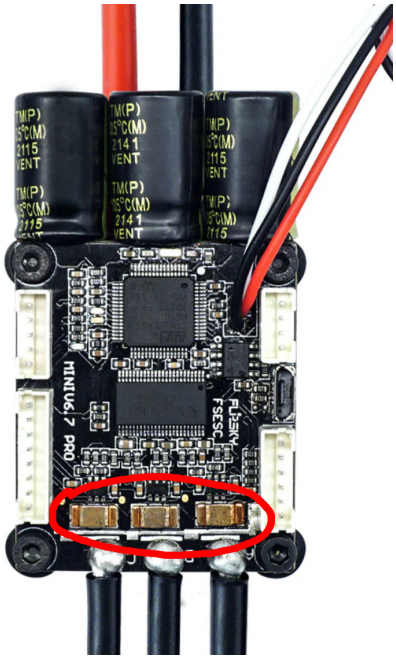

Figure 8: The motor driver, with the resistors circled in red.

3.4 Electrical tips for testbed 3

- To get a better data rate and less interruption during the control loop due to Serial transmission to the laptop (which is quiet slow), it may be interesting to buy a CAN to USB module to make the laptop communicate directly throught the CAN bus, enabling us to get much more data in real time. It may also introduce more complexity in terms of coding, so I'm not totally sure about the pertinence of doing it for now.
- for discussion about choosing a new motor driver and new motor, see the chapter on Cogging.

4 Software

The framework used for the development this testbed software was PlatformIO on VSCode. I recommend finding a quick tutorial on it to learn about it, how to upload a code, look at the Serial Monitor, ect... You also need Python for the computer interface. The current code is made so that you can extend it as much as you like for high number of degrees of freedom, as long as you take the time to write down the invert kinematics matrix, and change the Controlloopcallback function. This should enable a rapid iteration on the code for the next testbed.

To upload a new code on the Teensy 4.1, you can directly plug it to your computer with a micro-USB cable. For the motor drivers, it's a little trickier, as you need to use an external JTAG uploader. The one that we're using is the one from the white nucleo board on 9. To plug it to a motor driver, just plug the 5-pins cables to the motor driver. You can then upload a new code to the motor driver throught it. One thing to take care of when doing so is to change the CAN address of the driver written in the code to the corresponding one (the number of the motor, from 0 to 3, being the CAN address of the motor driver you are uploading to).

Reading this, you should have understand that the software from the testbed is made of two main parts : the code for the Teensy main micro controller, and the code for all of the drivers.

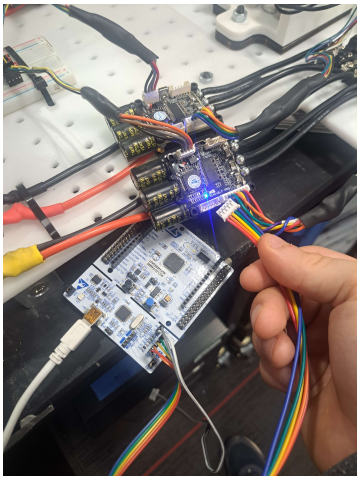

Figure 9: The 5 pins connector to plug and the white nucleo board.

4.1 Teensy code

4.1.1 TaskScheduler

The main code running on the Teensy is heavily based on the TaskScheduler library. This library allows us to specify a frequency for the execution of several functions in a simple and readable way, as long as the micro controller has the processing power to respect the specified timings.

```
//======================Task scheduler============================
// Callback methods
void controlLoopCallback();
void PCuplinkCallback();
void PCdownlinkCallback();

// Tasks declaration (This is where you declare how often you want y
Task controlLoop(1, TASK_FOREVER, &controlLoopCallback);  //1000hz
Task PCuplink(500, TASK_FOREVER, &PCuplinkCallback);      //2hz
Task PCdownlink(20, TASK_FOREVER, &PCdownlinkCallback);   //50hz
Scheduler runner;
```

Figure 10: The task scheduler functions declaration.

We call the functions that we want to call "Callback", and in the Task declaration we specify the delay to happen between each execution of the function. For example, the controlLoop is called once every millisecond, the PCuplink once every 500ms and the PCdownlink once every 20ms (equivalent to 1000Hz for the controlloop, 2Hz for sending back monitoring datas to the PC and 50Hz for processing messages incoming from the PC).

4.1.2 CAN bus and objects

Another main point of this code is how to the CAN bus is managed. It's managed using objects named FOCDriver that are the image of the motors drivers from the Teensy point of view. The Teensy is the master of the CAN bus, starting every communication on it. It can either use a set method to send a value to one of the driver to be set on it, or a get method that asks for an update of a value that is currently stored in the object.
In the driver code, a Controller object is the image of the Teensy, and it handles the reception of these set and update, and in the case of the later send back the requested value.
One the drawback of the way the CAN bus is managed is that you need to call processCANMessages that manage the reading of the incoming message often to ensure that any received new information is handled as soon as possible. In reality it could be improved if you call the function only when you know that there should be a new CAN message (because the Teensy initiate any CAN communication, you're able to predict when a new message should be there). But I didn't did it and instead decided to call calling the processCANMessages often in the different functions. It works so why changing it.

4.2 Driver code

The driver code essentially execute a FOC-oriented control algorithm on the motor driver. The principle of FOC is, using the phase currents and the position of the motor, to enable precise torque control of the motor. For that, we're using the SimpleFOC library that I recommend you to read the documentation.

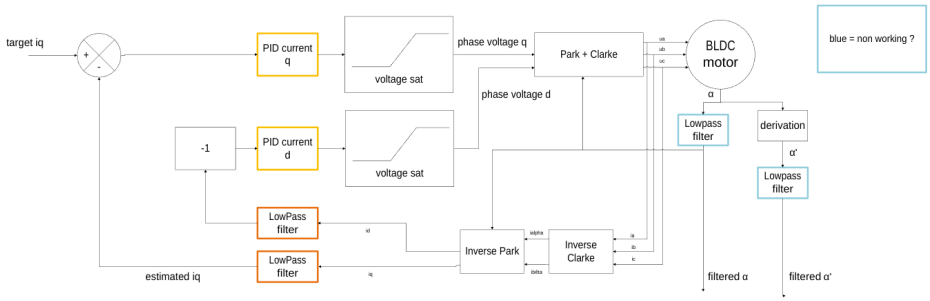

Figure 11: The FOC algorithm executed on the driver.

5 Control

All the θ are 3-dimensional vectors in the joint space, and all of the α are 4-dimensional vectors in the motor space. Iq is 4-dimensional as well, and is actually named targetforce in the code.

5.1 Torque control

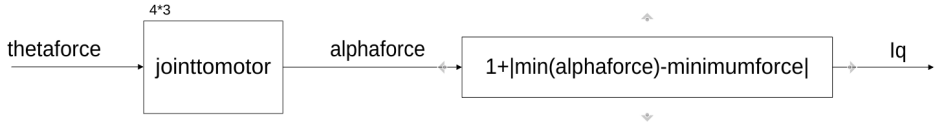

Figure 12: The torque control algorithm executed on the Teensy

The torque control algorithm is quiet straigthforward, with one subtility being the way the coactivation is calculated. The principle of the coactivation is to look which motor has the minimal torque that will be applied to it (which is generally negative) and to increase it until this particular motor has a minimalforce applied it so it keeps the cables tensioned. You then also increase the force applied by all the other motors by the same amount, so that there's no bias in the output torque of the arm.
The current Python interface of the torque control algorithm is a small interface with 4 sliders to command the torque of the 3 arm joints + the coactivation. A cool demo to do with this code is simply to let the torque to be 0 on all of the joints and move the arm freely by hand, while all of the cables stay tensionate, you can quiet feel the motor through the transmission.
The jointtomotor function is the following matrix product :

$$P = d_m \begin{pmatrix} \frac{1}{D_0} & \frac{1}{D_1} & \frac{1}{D_2} \\ \frac{1}{D_0} & -\frac{1}{D_1} & -\frac{1}{D_2} \\ -\frac{1}{D_0} & \frac{1}{D_1} & 0 \\ -\frac{1}{D_0} & -\frac{1}{D_1} & 0 \end{pmatrix} \begin{pmatrix} \tau_0 \\ \tau_1 \\ \tau_2 \end{pmatrix} = \begin{pmatrix} \tau_{m0} \\ \tau_{m1} \\ \tau_{m2} \\ \tau_{m3} \end{pmatrix} \tag{14}$$

5.2 Position control

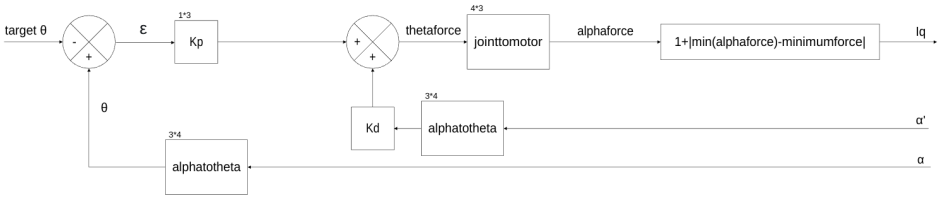

Figure 13: The position control algorithm executed on the Teensy

The position control loop reuse the torque control algorithm and use the position and speed measurements made on each of the motors to implement a PD control loop, in θ space. There is thus 3 Kp and 3 Kd values, one for each joint. Their current values were roughly choose at the end of my stay in the lab and could be finetuned, first by being leveled so that the joint-to-joint transmission feel more natural, and then by being increased or decreased so that the arm is either stiffer or more compliant.

The alphatotheta function is the following matrix product :

$$P = \begin{pmatrix} \frac{d_m}{4D_0} & \frac{d_m}{4D_0} & -\frac{d_m}{4D_0} & -\frac{d_m}{4D_0} \\ 0 & 0 & \frac{d_m}{2D_1} & -\frac{d_m}{2D_1} \\ \frac{d_m}{2D_2} & -\frac{d_m}{2D_2} & -\frac{d_m}{D_1} & \frac{d_m}{D_1} \end{pmatrix} \begin{pmatrix} \tau_{m0} \\ \tau_{m1} \\ \tau_{m2} \\ \tau_{m3} \end{pmatrix} = \begin{pmatrix} \tau_0 \\ \tau_1 \\ \tau_2 \end{pmatrix} \tag{15}$$

For the Python interface of this mode, there's 2 that are available. The first one is the Jointpositioncontrol one, that simply enable to move the joints with 3 sliders. It is quiet intuitive to use.

The second one is a code that show the reachable surface by the third joint of the arm. You then click on one point of this space, and the arm will reach this new cartesian position from it's precedent position while following a line. This code only uses the two first joint, the third being let free because otherwise there would be an infinite number of solutions for the joints of the arm. Because there is still two possible theta solutions for each of the cartesian position of the arm with two joints, I decided to split the reachable surface of the arm into two halves, each half having one conformation enabled. If you switch from one half to the other, the arm will go back being straight before switching to the other half.

Right now, the dynamics of the arm are not taken into account when driving the arm, which makes the arm move sometimes faster or slower than expected for some joints, and prevents the arm from actually following a line. This could be worked out in the future by a more sophisticated control loop, or by tuning the Kp values.

5.3 Future works to be done

One of the future work to be done with Testbed 2 in terms of control and sensing may be to use the position control mode, with the Kp of the last joint set to 0 and to look at the position of the end effector and use it as a way to "sense" and map the environnement that Testbed 2 is evolving in. Using torque control to press gently and precisely on a defined surface could also be something to try out.

6 Cogging

6.1 My work

Most of the files for this study can be find in the Git repo in the "anticogging project" folder.

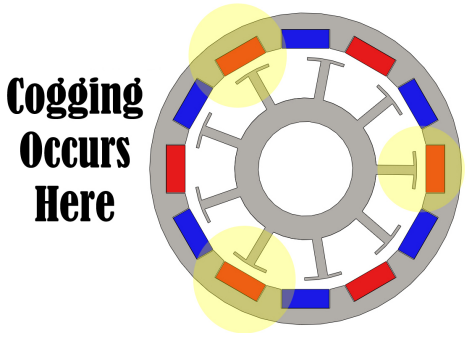

Figure 14:

Cogging is the parasitic torque caused by the interaction between the metallic stator of a motor and the magnets from the rotor. Unfortunately, it is quiet high on our current motors, and create two issues : - It makes us feel small "bumps" and inconsistency in the movement and torque of the arm, due to these variations in torque depending on the position of the motor. -High "deadband" for the motors and hence the arm : the cogging can be assimilated to a sort of high static magnetic friction, that prevent the motor from going to a precise position (high steady state error) when using a PD control loop. (Using integral control solve this issue, but also ruins the compliance of the arm).

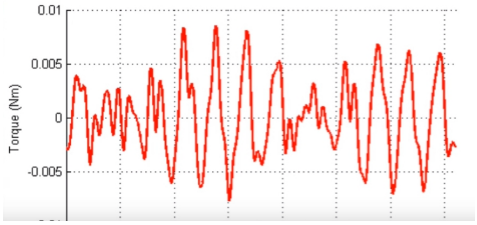

Figure 15: An example of the pattern of the cogging of a motor.

What I first did was to map the cogging using position control, and looking at the current used by the motor to hold a given position, I was able to get the cogging torque at this particular position. A Fourier analysis of the obtained data told me that our motor had two main frequency for cogging : 14Hz and 112Hz.

This can be explained easily if we take a look at the geometry of a brushless motor. A brushless motor has slots, which are the branches of the stator, and poles pairs, which are the pairs of magnets on the rotor. Our motor has 24 slots and 14 pole pairs (28 magnets). The 14Hz is hence easy to understand, and 112Hz = 28*8Hz, Thus an harmonic of the geometry.

Using the obtained datas, I set up a feed-forward in the control loop adding the opposite of the cogging into the control. Unfortunately, I was stupid at the time, thinking that the magnetic encoders were relative encoders, which was an issue for the start-up of the anticogging algorithm (because the cogging is dependant of the position of the motor obviously). These encoders are actually absolute encoder. So I tried to make the anticogging algorithm to start blindly at a local minimum, without really knowing where the motor was in the cogging pattern. This led to deceptive result.

6.2 giving it another shot

I wish I gave anticogging a better shot, several months later. Now that we know that the magnetic encoders are absolute, doing it should be quiet simpler. If someone want to try it again, the best

ressources on it are this paper from Matthew Piccoli and Mark Yim, as well as the SimpleFOC discord, where there is a bunch of people that already tried to implement anticogging with different level of success.

6.3 The future for motor drivers and motors

If you decide on continuing using brushless motors, you can switch to Odrives motor drivers. Firstly, this way you won't need to switch the resistors each time. Secondly, the Odrives comes with a proprietary software (that you still can replace by SimpleFOC is you end up not wanting to use it). This software have an anticogging algorithm implemented, it could be interesting giving it a try before giving up on brushless motors. Finally, we need to discuss about the motor. Switching to even larger outrunner brushless motor with less cogging and higher torque (low Kv number) (we will eventually need more torque if we want a bigger arm anyway). I personally would have loved to try the Eaglepower LA8308 based on their online reputation in robotics.
Otherwise, following Ken advice and switch to brushed motors may be the wisest choice, if the next person handling the project feels confident doing the switch to this type of motors.

7 Conclusion

To get a detailled conclusion on the learnings for each part of Testbed 2, read the end of each chapter, where there is different recommandation based on my experience with it. I don't know what is the next step for Testbed 2, apart for some control/sensing experiments to run on it. If you need any additional information/clarification on Testbed 2, you can reach me at matthieu.prigent@student-cs.fr
.