System optimization and Software design in Grating-Interferometry X-Ray Imaging Laboratories

Eliot Jermann
Supervisor - **M. Stampanoni**
Co-advisor - **M. Polikarpov**

June 2, 2022

Contents

1 Introduction

1.1 Grating-Interferometry X-Ray Imaging

In Switzerland, cancer is the most common cause of mortality in the segment of the population aged of 45 to 84 years [14]. Between 2013 and 2017, breast cancer was the most widespread cause of death due to cancer in the Swiss female population with 18 [%] just before lung cancer with 16 [%]. [9]. All around the world, the implementation of regular screening by mammography has reduced the mortality of breast cancer [8]. In the Netherlands for example, the mortality rate has decreased over 30 [%] for women aged between 55 and 79 years over the last twenty years [16]. Mammography and ultrasound are currently the most efficient ways of screening dense and non-dense breast tissue. However, mammography suffers from the low soft tissue contrast. Current issues of that imaging system are the superposition of relevant anatomical structures and the low values for sensitivity and specificity in dense breast tissue. That is where phase contrast can help, because it provides an additional contrast at the interfaces of areas with different electron density [8, 15, 17]. Phase contrast X-ray imaging is widely used at the synchrotron because of the unique capabilities of the radiation. The sychrotron offers a small but powerful monochromatic radiation. The large distance between source and sample and the good quality beam make it possible to have coherent radiation falling at the sample. Nevertheless, the access to synchrotron radiation can require time and peer-reviewed proposal. For that reason it is not very suitable for clinical application. In contrast, grating interferometry (GI) is the very promising technique which could bring the advantages of the phase-contrast imaging in the laboratory setup. Its concept is to install additional elements, gratings, which allow to recover absorption, phase and dark-field contrast signals from one measurement [13]. The group of M. Stampanoni are one of the leaders of such development. Among various setups and applications of the laboratory, there is one high-resolution GI phase contrast X-ray imaging system, which is able to record tomograms of small breast tissue samples with a resolution of less than 20 [μm]. That particular setup aims to support pathologists and radiologists in ex-vivo assessment of breast specimens. It can perform CT scans with a spatial resolution similar to the achieved one with histology. Compared to histology, it has the advantage of not destroying nor deforming the sample and of avoiding the loss of resolution along the axis normal to the imaged slices. On the other hand, the contrast and the visibility are still better on a histological section.

The project described here was aiming to facilitate development of that an GI imaging setup towards the stand-alone clinical system.

1.2 Setup description

The project has been performed with the imaging system shown below. Figure 1 shows an annotated picture of the whole GI X-ray imaging system. The source is a Sigray MAAST X-ray source with structured anode [4], thus no G0 grating is needed. The Huber tower [5] is the sample holding stage of the setup and is the component of interest in the project. It is composed of two tilting and one rotational stage. G1 grating is a pi-shifting phase grating produced by deep reactive ion etching of silicon. It has a period of 3 [μm] and a height of 25.5 [μm]. G2 is an absorption grating produced by deep reactive ion etching of silicon and gold electroplating. It has a period of 3 [μm] and a height of 35 [μm]. The gratings are essential to restore phase contrast and dark-field signals. Both gratings are hold by 6D towers produced by the SmarAct GmbH [1]. These towers allow all necessary linear

translation and rotation needed for gratings' positioning. The detector is a X-ray sCMOS GSENSE 16.4 MP detector made by Photonic Science [12]. It has a pixel size of 19.85 [μm] and a field of view of 4045x4041 [pixels^2] which is approximately 8x8 [cm^2].

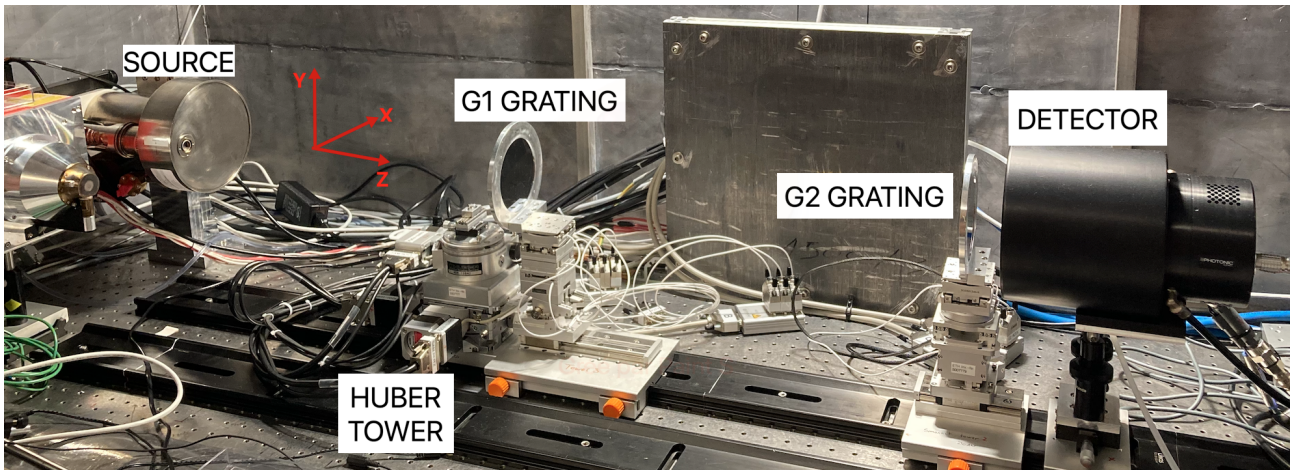


Figure 1: The figure shows the complete setup with its main components annotated.

2 Motivation of the project

2.1 Overall goal

The goal of the project was to make the Laboratory-based X-ray phase contrast systems completely independent of existing hardware controlling infrastructure at the PSI. As a matter of fact, the setup will be moved to ETH Zurich buildings during the current year. In the future, it also has the ultimate goal of becoming clinically compatible. These reasons fully justify the need of making the setup independent. The sample holding tower was the last component of the imaging system which relied on PSI infrastructure. It was controlled by standard motor controllers provided by the PSI and embedded as part of the beamline infrastructure. The infrastructure cannot be moved and relies on the support of the PSI engineers. That system was working correctly and was routinely used for experimental work and tomographic acquisitions in the laboratory. TOMCAT group has bought a specific controlling system to control Huber tower. It has replaced PSI's motor controllers. During the project, *SMC 9300 motor controller* [5] was set. Its hardware and software was integrated to the imaging setup and to the computed tomography (CT) acquisition pipeline. The new controlling system was also tested and his performance optimised. It has been implemented for the real experiment and has allowed the acquisition of new CT scans.

2.2 Objectives

In practice, the project was divided in three different parts. The first part of the project is about controlling system establishment. Huber tower was connected to the controller and adjusted. It was made sure that it worked from the graphical user interface (GUI) displayed on controller's tactile screen. Then the controlling system was integrated into the python environment which is used for setup's utilisation. In the second part of the project, quality assessment and performance tests were run to make sure the tower was working as expected. It was also verified that the tower was adequately integrated to the setup for real

experiments. That was then confirmed by running a CT acquisition of different breast samples. During the last part of the project, I helped to run tomographic experiments by doing several experimental tasks in the laboratory such as setup alignment and image acquisition and processing.

3 Controlling system establishment

3.1 Huber tower

Huber tower [5] is the component of the imaging system in charge of rotating the sample during tomographic acquisition. It is composed of two tilting and one rotational stage. The stages are moved by their respective motors. Each stage has limit switches and encoders. Huber tower is shown and described in figure 2. The tilting stages were needed to correctly align the sample to the setup. Indeed, the vertical axis of rotation of the sample must be known and accurately set. The rotational stage was used to rotate the sample around its vertical axis during a CT scan. The limit switches and the encoders were used for homing procedures.

Huber brand is known for its very precise technology, the tower is supposed to have a precision of 1-2*10^{-3} [deg] [5]. In chapter 4.1 that precision was confirmed. Encoders have a precision of $10^{-}5$ [deg], [5].

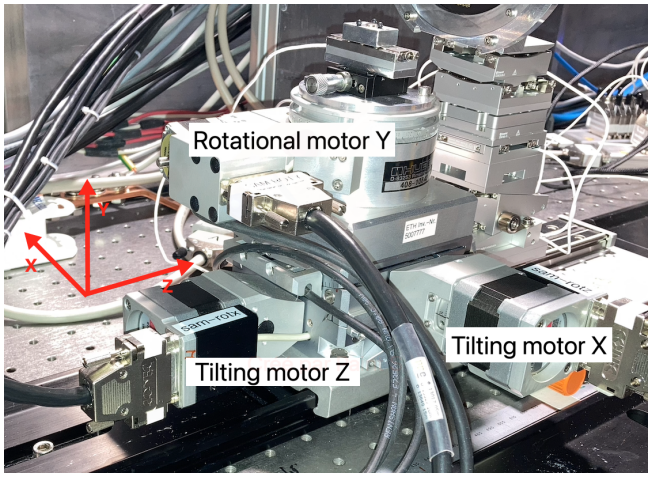
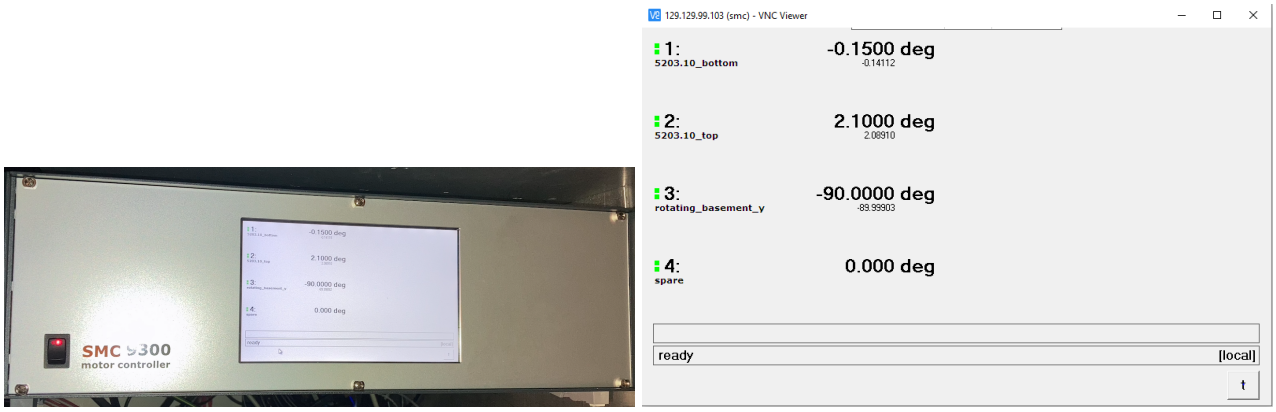


Figure 2: Huber tower with named motors. It is the rotational stage which is used for tomographic acquisition in the setup.

3.2 SMC Controller

SMC 9300 controller was bought to replace PSI controlling system. The new controlling system can handle up to four motors and theirs encoders. It is a piece of hardware running under windows seven operating system. The SMC software [5] installed on it offers a graphical users interface which one can use with the tactile screen. The GUI is user friendly and makes it possible to use the tower without an external input, it is shown in figure 3 (b). The motor's features which are not displayed on the GUI can be send by typing them in a command bar or by inserting a text file into the software. The functioning of the controller is described in details in the users manual [7] (credentials : user, smc). That manual does also describe the language and the function which one can use to operate the motors. For the seek of tomographic acquisition, the tower must be operated from a python code on an

external computer. It is possible since a TCP server [11] is installed by default on the SMC hardware. That server allows the remote control of the tower via a TCP client generated inside a python code.



(a) SMC controller (b) SMC interface

Figure 3: The figure displays (a) the controller on which you can see the GUI on the tactile screen and (b) a screen shot of the GUI.

3.3 Tower connection and calibration

Firstly, the tower and the controller were connected together. The cables were named after the motor they were connected to, for example SAM_ROTY stands for the rotational stage, and wound up in a way to save space inside the setup. The controller has been connected to the device network at TOMCAT such that it could be recognized by other network devices and got an access to the internet. To add a new computer to the server it had to be registered as belonging to the laboratory.

Once the system was connected, it had to be calibrated. Different types of calibration were done. First, the tilting motors were doing a step of half a degree physically for one degree on the controller. Using the configuration variables *gear num* and *gear denum* the step size has been set to be equivalent in reality and inside the controller. The rotational axis did not have that issue.

The encoder resolution was calibrated for the rotational stage. It is a value near one which calibrates the step size to the encoder. The equation below describes the relation between new and old encoders resolution :

$$new\_eres = \frac{old\_eres * should\_be\_position}{encoder\_position} \tag{1}$$

Eres stands for encoder resolution, *should_be_position* is the position in degree at which you commanded the stage to be and *encoder_position* is the value displayed by the encoder. Applying that equation in an iterative manner will calibrate the step size to the encoder.

Another part of calibration was done on homing procedures. The homing commands were not sending the motors to the desired home position which was zero and were even not working for the tilting stage along the x-axis. As you can see on figure 4 a stage has two limit switches. The tilting motors were configured to run in inverted sense. It lead to an issue. The trigger was not activating the first limit switch but the second. It resulted in the stage being locked in between the two switches. The problem has been solved by changing the rotation sense to *normal*. Another issue faced with the limit switch was that

one of the metal pieces meant to trigger the switches was not thick enough to release the safety mechanism. That triggering piece is highlighted on figure 4. The problem was fixed by increasing physically the size of the trigger. Once these issues solved the home position had to be calibrated. It was done with the command *erofs*. That command saves the offset of the stage to the desired homing position and corrects it on the encoders. They are different ways of homing, for that tower two of them were used. The tilting stage are homed with a procedure using the limit switches and the rotational stage does use an in-memory home position. The limit switches do exist on the rotational stage but as for tomographic acquisition the stage must rotate over 360 degrees, triggers were not installed. Even if they are explained in details in the controller's user manual [7], these homing procedures require a minimum of knowledge about the tower, for that reason they were implemented in the API presented in a next chapter.

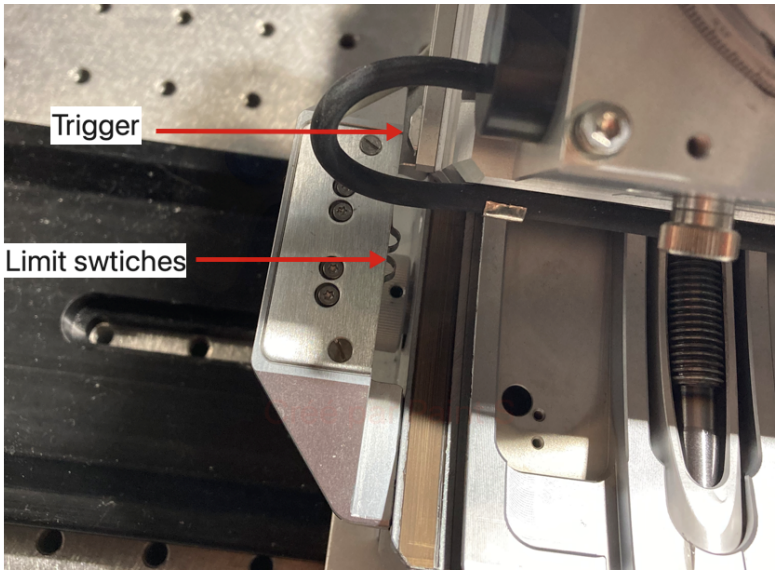


Figure 4: The figure show a situation when Huber tower went out of its range of function. It happened because of a defect limit switch.

3.4 API : MotorClass

The main part of the project was the creation of the API MotorClass. Even so the GUI of the controllers presents all requested features, the API was needed for an user friendly control of Huber tower from a jupyter notebook running in python. That API had also to be compatible with all existing tomographic pipelines. The initial idea was to copy the logic and the code from PSI controller's API. PSI controller was operated using epics.PV library [2]. Epics is a very general library meant to work with a multitude of motors. The implementation of the library was not very straight forward for the very specific task of integration of the Huber tower to the tomographic pipeline. It turned out to be faster to create a new, smaller library, tailored for the operation with the Huber tower.

MotorClass is an API with different types of methods necessary for setup's operation, see Appendix A. An object of that class commands a single motor with its stages, it does not control all stages at once. That choice was made because during tomographic acquisition there is only the need of controlling the rotational stage, the tilting stages are requested for sample alignment. Once the sample aligned to the setup, the tilting motors must to stay static. Thus, there was no need of controlling them during tomographic acquisition. The class offers different types of external methods which are classified in three main categories

: movement, query and configuration methods. The table below summarised the function implemented in the class 1. Each kind of method is described in the next chapters. "Wait" is an internal function playing a crucial role for tomographic scans, for that reason it was described in detail in chapter 3.4.4.

Methods	Description	Category
put()	Puts to absolute position	Movement
putr()	Puts to relative position	Movement
stop()	Stops any running process	Movement
home()	Runs homing procedure	Movement
get()	Returns and prints encoder position	Query
get_status()	Prints the whole status	Query
change_acc_speed()	Modifies the acceleration speed	Configuration
change_slew_speed()	Modifies the slew speed	Configuration
calibrate_eres()	Fine tuning of encoder resolution	Configuration
wait()	Keeps the python cell running	Internal
anycommand()	Send any command to the controller	-

Table 1: The table display a list of relevant methods implemented in the MotorClass with a small description and its category.

The commands are sent to the controller via the network using the existing TCP server and the socket library to generate a client [10]. The creation of a client requires the knowledge of controller's IP address. Since it is connected to TOMCAT server, it can be found by knowing its mac address and using *fing* software [3].

A jupyter notebook showing implementation examples was created, see appendix B. It is meant to help any new user to utilize the newly installed controlling system with MotorClass. It has also a section about recurrent errors and examples of low level communication with SMC controller. Besides from FAQ_MotorClass describing the functioning of the class, every method was commented with its basic function, variables and specific features.

3.4.1 Movement methods

The movement methods are the category of function which are sending commands to Huber tower which will make it move. Two basic methods *put/move()* and *putr()* were implemented in the class. These are respectively for absolute and relative angular positioning of the stage. Figure 5 shows the method *put/move()*. The method is briefly described and offers the option wait. It is an internal function which will be described in a further chapter. Another method considered as leading to a movement is *home()*. As already explained above, the homing procedure is not straight forward and for that reason was implemented. The last method of the list is *stop()*. That function stops any running task on the Huber tower. It can be very relevant in case of home office, when one could not physically access the controller and would require to stop the running command.

```
def put(self, degree = 0.0, wait = False):
    """
        put(degree, wait) : Puts the absolute position and has the option of letting the cell run until the process is done.

        Variables :
            - degree (double) : position value in degrees of a tilt or a rotation.
            - wait (bool) : choose wether or not the wait (True = wait).

        If wait = True, the function get() is called.
    """
    command = "goto" + str(self.axis) + ":" + str(degree) + " \r"
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((self.HOST, self.PORT))
        s.sendall(command.encode())
        time.sleep(0.1)
        data = s.recv(1024)
    if wait :
        time.sleep(0.1)
        self.waiting()
        self.get()
```

Figure 5: The figure shows an example of method implementation in the class MotorClass. It is the put method for absolute angular positioning. All method are described the same way.

3.4.2 Query methods

The query methods return information about Huber tower. The methods *get/get_position()* and *getstatus()* were implemented in order to access the angular position of the stage and its complete status. Both function are printing the information, and for the seek of saving the angular position *get()* is also returning the numerical value. A status information has that form : "0::-90:-90.0006820016057:0:0:0:1:0:0:0:0:0:0:0:0". The legend is displayed in the description of the method. For example the eighth value is motor's readiness. It means that the motor is ready to receive a command. No other query method was implemented as they would not be used frequently.

3.4.3 Configuration method

The configuration methods were written to modify the axis configuration of the motor. SMC's GUI is very user friendly and changing axis configuration on it is convenient. The reason these three methods were implemented was to save time as they were used multiple time during the project and modifying them on the GUI is more time consuming. The methods *change_acc_speed()* and *change_slew_speed()* modify the acceleration and slew speed of the stage. The acceleration stands for the speed at which the motor increases its speed until going to cruise speed. The slew speed sets the maximal cruise speed allowed. These both methods were used during the time optimisation of Huber tower's movements, see chapter 4.3. The other configuration method is *calibrate_eres()*. It is for calibration of encoder resolution, the procedure is described in details in chapter 3.3.

3.4.4 Internal method and *anycommand()*

The internal methods are *wait()* and *find_nth()*. The second method was implemented for the seek of string management inside *get()* method. *wait()* was written in order to keep the cell running as long as the tower's task is not done. That function is crucial for making CT acquisition on that setup. The image acquisition is done step by step and the sample has to stand still during exposure. The waiting method is based on motor's readiness, the function checks constantly the status to know if the tower is ready for its next task.

Writing that function revealed a challenge, in fact when first trying it, it lead to an error. The problem is described with the help of figure 6. The delay between two code line in

python inside the notebook appeared to be smaller than the one between the controller and the Huber tower. When the status is asked, it prints the command inside the controller's TCP server and after a some dead time the tower's status is as well printed in the server. It is that second line that the python cell needs to read. Due to its smaller dead time, the cell inside the notebook would try to read the status on the server before it is even printed. That process lead to errors because the requested information is not yet available. Firstly, the issue was solved by introducing sleep times in between the lines inside a python cell. To completely avoid the occurrence of that error, the sleep time was put to 0.5 [sec]. Subsequently, the problem was overcame by implementing an error management because the produced error was exactly known. The principle was to re-run the code line trying to read on the TPC server until the error disappeared. To avoid an infinite loop in case of a controllers crash, a limit time was implemented. If the limit time is reached, an error message is printed telling the origin of that error. It could come from a connection issue between controller and computer or it could be that the tower's movement has taken longer than the allowed run time. The second error should not appear in normal conditions, the time margins are relatively long, about twice the amount needed for a 360 degrees rotation. The current method is robust and has worked properly during tomographic acquisition.

There is another idea of implementation avoiding the use of error management one could try in the future. That would be to find a way to get information about TCP client's readiness, and once knowing that the server is ready, executing the reading line inside the python code.

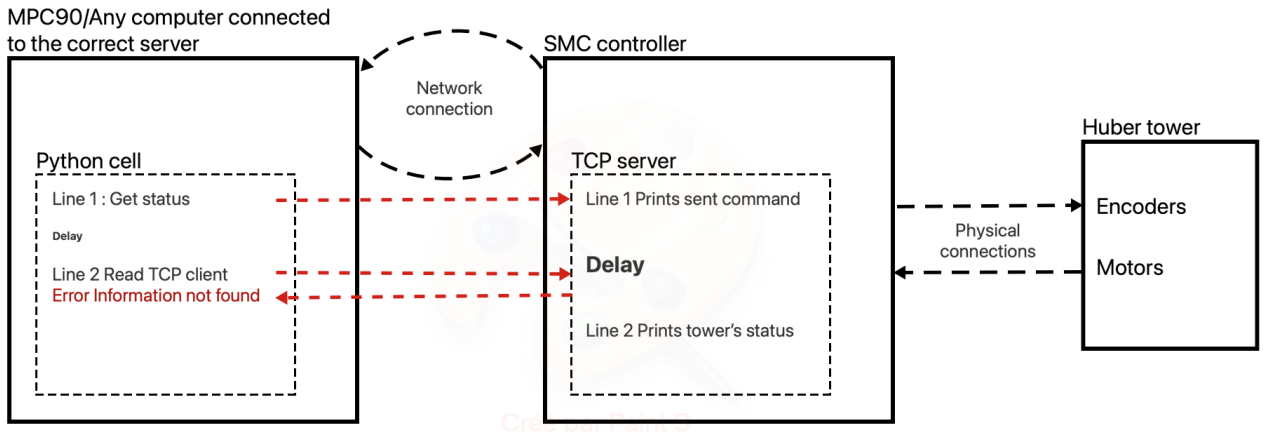


Figure 6: The figure shows a diagram explaining the time delay issue. In summary, the speed at which the python cell requests to read the controller's answer is faster than the process between controller and tower. Asking for currently non existing data does lead to errors.

The command *anycommand()* has been implemented in case a future user has the need of sending a SMC command from the python code to the controller for which no method exists.

4 Controlling system testing and improvement

To assess the precision and the repeatability of the newly established controlling system, test tomographic acquisition were ran. The principle of these scans were to move only the rotational stage of the Huber tower to positions which could be used during real tomography, to save encoder's position at each step and to measure the total running time. A typical test

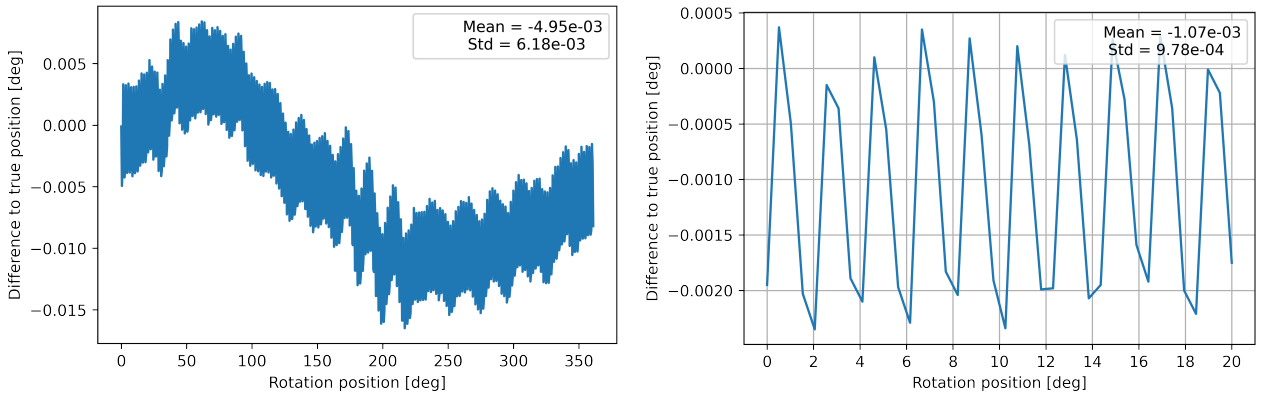
scan was sending the stage to two positions per degree over 360 degrees, it represented 740 angular positions. According to the manufacturer the rotational stage should be precise to (1-2)*10^{-3} [deg] [5]. The precision and repeatability tests were also performed with PSI controlling system because it was not previously done. As the new system must work at least as good the previous one, it was important to know how PSI motor controller performed.

Subsequently, the tower's speed was optimised. The motivation for it was the significantly high overhead time during real tomographic acquisition. The overhead time was of approximately 3.5 to 6 seconds and is due to only three components, the detector, G2 translation stage and the Huber tower. As part of the quality assessment, the speed of the rotational stage has been optimised. The time for a movement of 0.5 degree has decreased form circa 1.1 to 0.4 seconds.

4.1 Quality assessment of precision

To test the precision of the rotational stage test scans with two projection per degree over 360 degrees were run about forty times per configuration. The encoder position was then compared to the true position. The true position is the position at which the tower was meant to be.

After acquisition, two types of systematic positioning errors were uncovered. A high and a low frequency error which you can see on figure 7. The high frequency error has a sinusoidal shape with a period of exactly two degrees, it was the same over all scans. The reason for that error is due to the intrinsic functioning of stepping motors and regarding its standard deviation of $9.78 * 10^{-4}$ [deg] it was in the range of the tolerance which is $(1-2) * 10^{-3}$ [deg] [5] set by the manufacturer.



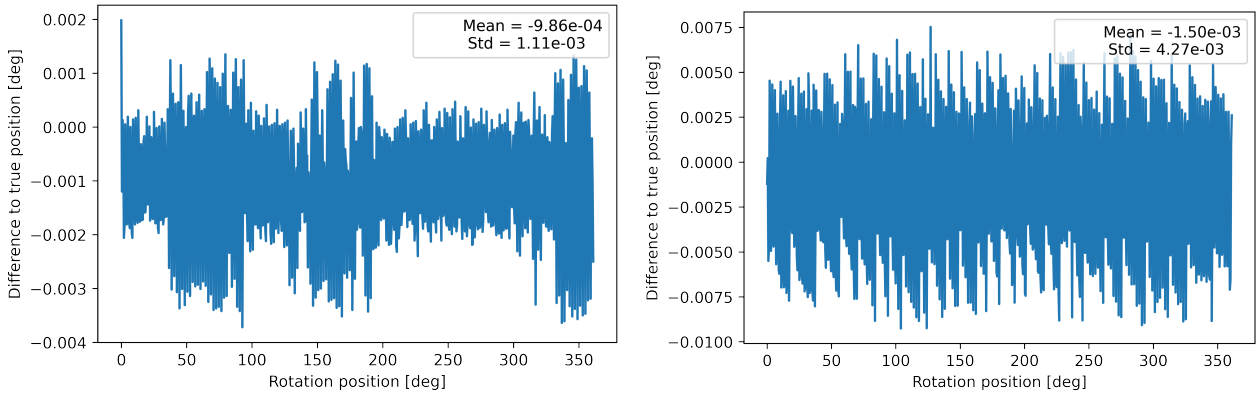
(a) Low frequency systematic error

(b) High frequency systematic error

Figure 7: The low and high frequency error are measured by comparing the true position with the encoder's position at which the tower is. The high frequency error plot is a zoom in of the low frequency one.

The other error had a low frequency and was higher than tolerated. It appeared to be due to disabled closed loop. Closed loop is a feature which allows the motor to check its position after the first guess and to correct its position if the error is bigger than the limit internally set. When it is disabled it would stop in the position it was first sent. After activation of closed loop, the low frequency systematic error disappeared. Figure 8 shows the error over the whole scan with both controllers. It is basically the high frequency error which translated over the complete scan. The standard deviation has improved over a factor three from the

previous to the new controlling system. There is still an unexplained point, the pattern seen on the SMC controller in figure 8 (a) is systematic. It is not an issue as the error is in the tolerances but seems like it could be improved.



(a) SMC controller

(b) PSI controller

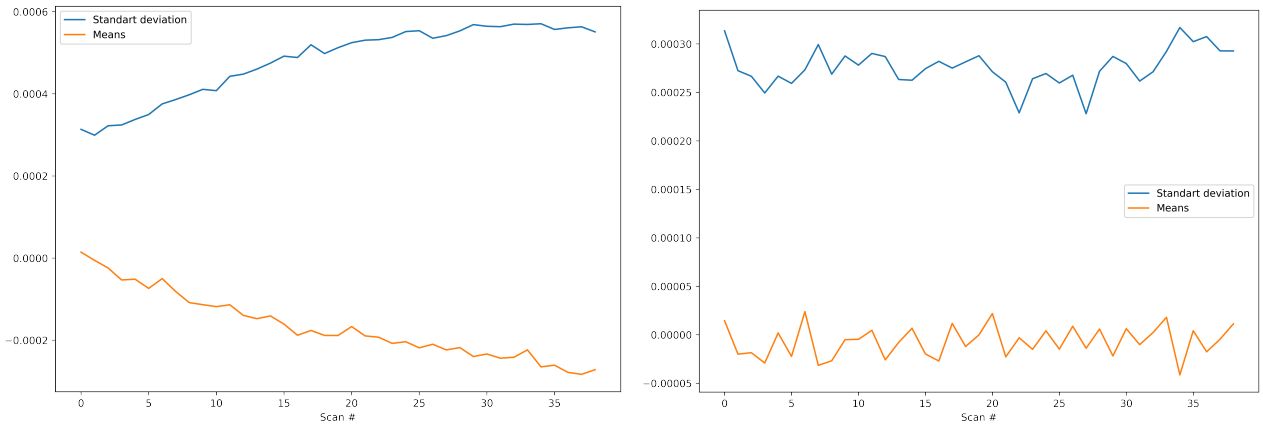
Figure 8: Theses plots are the difference between true and encoder's position over 360 degrees.

In conclusion, the errors were in the tolerated range and thus assessed that the precision of the tower is suitable for the tomographic measurement.

4.2 Quality assessment of repeatability

For repeatability the same data as acquired in the for precision assessment chapter has been used. In order to evaluate their change over time the scans were compared one to each other. In figure 9 (a), the encoder positions of the first scan were subtracted from the encoder positions of the 2^{nd}, 3^{rd}, and etc scans respectively. Then the mean and the standard deviation of these values were plot per scan. A little shift of the mean was observed over time, as a consequence the standard deviation is increasing. Regarding the size of the shift which is in the order of 10^{-4} degrees one can tell that it is completely insignificant for tomography, the standard deviation between scan is smaller than the error on positioning for a single scan. It could be that the tower's precision is in reality smaller than what was found chapter 4.1.

In figure 9 (b), each scan was compared to its previous one in the same way each scan was compared to the first one in figure 9 (a). On the second plot, no shift in mean neither change in standard deviation is visible. Again, the error is in the order of 10^{-4} degrees and thus insignificant for tomographic acquisition.



(a) Comparison of scan N to the scan 1

(b) Comparison to of the scan n to the scan n-1

Figure 9: In the plots, in order to evaluate their change over time the scans were compared one to each other. The encoder positions of a scan were subtracted from the encoder positions of the other scan. The scan's choice depended on the desired comparison.

In conclusion, the repeatability between scans is very accurate. In fact the standard deviation between scans is smaller than the standard deviation observed on a single scan, in that respect the repeatability is assessed.

4.3 Time optimisation

The motivation for time optimisation has been mentioned in the introduction of chapter 4. To enter a bit more into details the mean overhead time for one image during a real tomography with initial SMC configuration is shown in figure 10. The three sources of delay are the detectors delay, the G2 translation stage movement and delay and the movement and delay of Huber tower's rotational stage. The detectors delay has neither been tested nor optimised but was in the range of 2 to 3 seconds for a 15 second exposure time image. The fact, that overhead time increased with exposure time, was certainly due to the increasing amount of data transferred by the detector. There was still a significant amount of overhead time due to the towers' movement. To optimize Huber tower's rotational stage movement time, test tomographic scans have been run. These scans were made up of two steps per angle over 185 degrees. The total test scan time with initial axis' configuration was of about 5 minutes, these values were used to estimate the relative time gain. As a measure of precision, the standard deviation between true position and encoder position has been used. For statistical power theses measurements were taken three times and their mean was used, therefore one can see error bars in figure 11

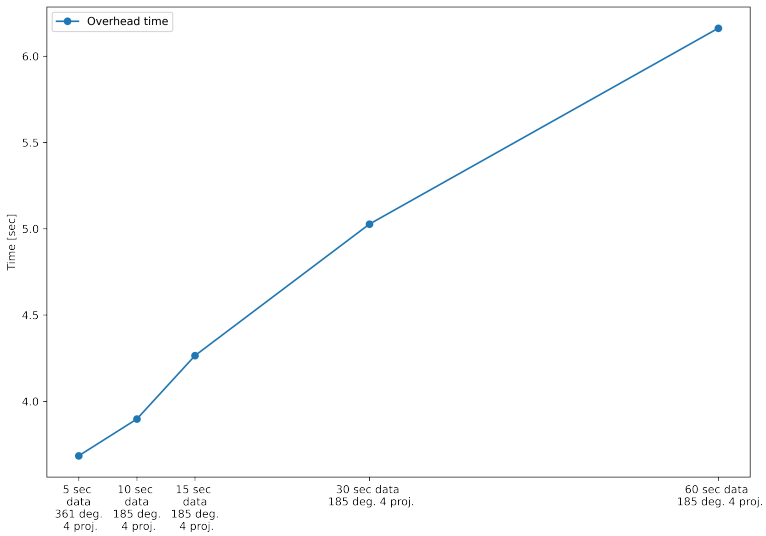


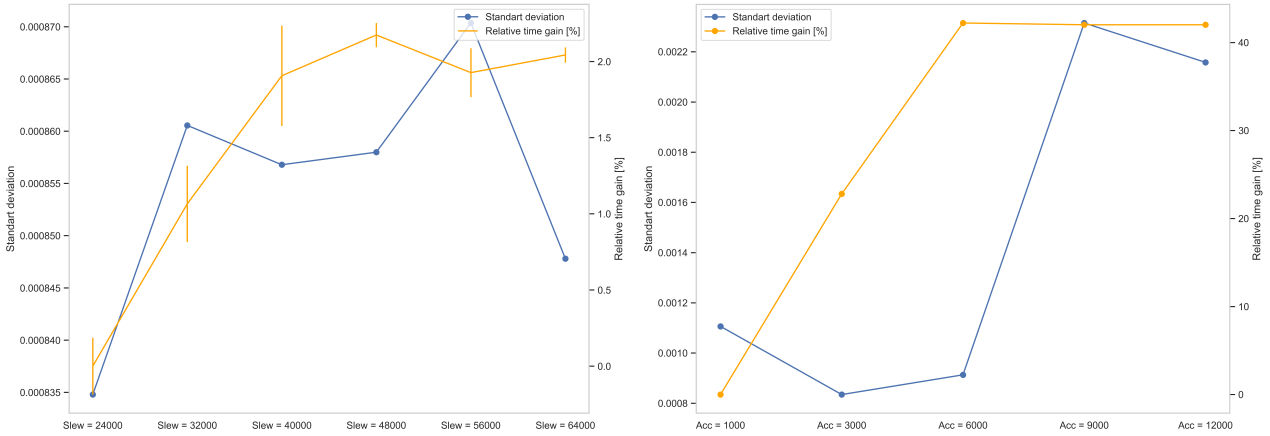
Figure 10: Plot of the overhead time with first SMC configuration.

Firstly, the sleeping times implemented inside MotorClass were optimised. For conservative reasons sleep times of 0.5 seconds were initially set at several spots to avoid time delay issues. When reducing these sleep time to 0.1 seconds a relative time gain of over 30 [%] was observed. Under 0.1 seconds no significant relative time gain was noticed.

Two other parameters were modified to reduce overhead time and these were slew and acceleration speed of the motors. The idea was to compare the actual speed configuration to the PSI's configuration but the values describing that feature were completely different and no relation has been found. The method used to assess the best speed values was to fix one of the speeds and increase the other until a plateau in time gain is reached or a significant loss in precision is observed.

Slew speed increase described in figure 11 (a) did not have a significant impact on relative time gain, only about 2 [%]. According to the order of magnitude of the standard deviation between true position and encoder position, the precision was not affected by slew speed increase. The value of 35000 has been hold back, one of the reasons was quite subjective, over that 35000 number, the motors seemed to have very high revolution for an insignificant time gain.

Acceleration speed, figure 11 (b), did have a significant impact on relative time gain and on precision as well. One can clearly see that the standard deviation between true and encoder position has exceeded the one assessed in chapter 4.1. Regarding the curves displayed on the graph the optimal acceleration speed selected was 6000. Indeed, it has already reached the plateau and loss of precision was not yet to observe.



(a) Slew speed

(b) Acceleration speed

Figure 11: The figure shows two plots in which one sees the standard deviation of scans position to true position in blue. The yellow curve is the relative time gain, there is an error bar because an average over tree measure was taken. The x-axis display the speed values which has been modified.

In summary, a significant amount of time was saved by performing that time optimisation. One can observe a total relative time gain of over 70 [%]. It was one of the reasons of tomographic overhead time and it has allowed to save over 0.5 seconds per image. It resulted in approximately 360 seconds saved time per tomographic scans as they typically consist of 740 images.

4.4 Unsolved issues

An issue which has not been solved is the warmth of the motors. When touching the motors one could tell that they are much warmer than with the new controlling system than they were with PSI's one. The fact motors are warm is normal as current passes through. It is needed to prevent the motors to move. But there is no reason for it to be higher with the new system. A way of reducing the current does exist. It require to modify parameters not available from SMC software and was not explored by now. According to the seller, the controller is correctly configured for the Huber tower motors.

Another remaining room for optimisation is the way of dealing with the time delay issue. One could take the time to understand the working of communication between hardware units and define a hierarchy avoiding the emergence of such type of issue.

5 Experimental work on at the X-ray imaging system

The setup with the recently replaced controlling system had to be tested under real conditions. There was one experiment with breast sample during my stay. Because the system was not used for some time, it had to be assembled and aligned. The source needed to be started and its proper functioning ensured. Gratings with the proper period and highest visibility needed to be picked up from the PSI collection. Then, the whole geometry of the setup had to be set. It meant the distance between the towers, the alignment of the gratings, the cone beam axis and to mount and set the sample at the centre of rotation. Moreover, the image quality during image reconstruction had to be optimized.

I participated in all aforementioned parts of the experiment and would like to highlight a few parts where my input was the most.

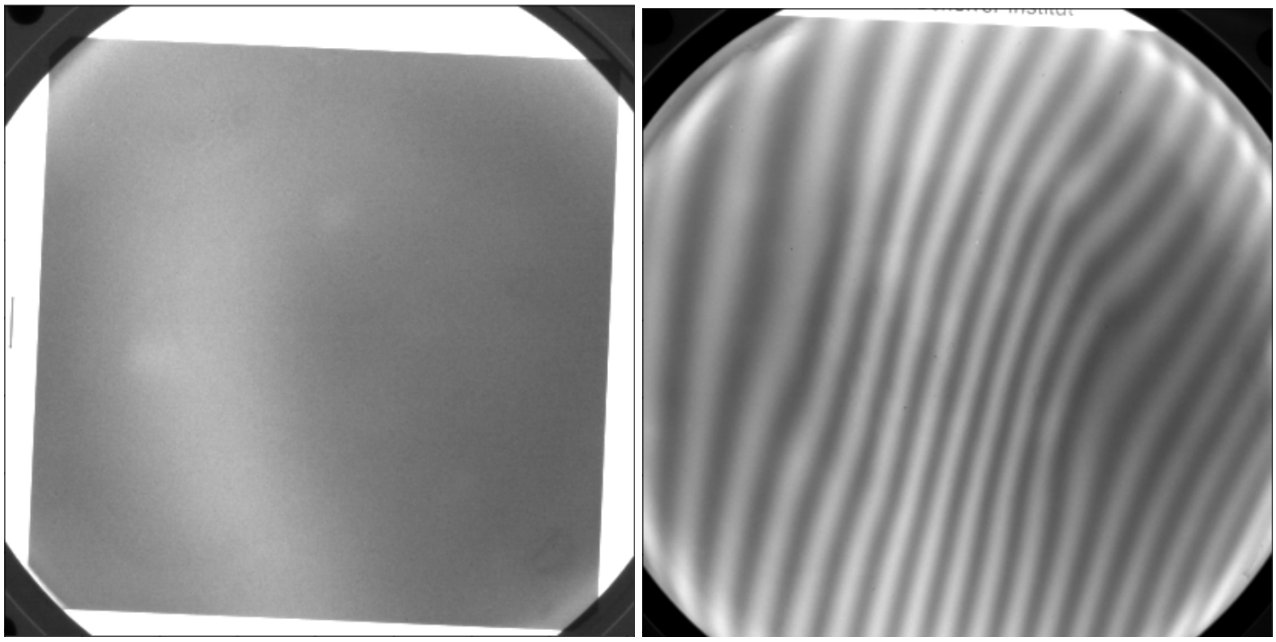
5.1 Grating alignment

The alignment was done in two steps. Firstly, the macro-alignment was done by approximately fixing the towers and the detector at the desired distances with the help of a ruler. Then the gratings were approximately aligned in all required directions by eye. Subsequently, the micro-alignment was done with the help of images taken by the detector. The gratings were placed on the 6D towers which allowed all necessary linear translation and rotation requested for the alignment.

The macro-alignment in terms of distances along the source-detector axis was set according to previous calculation. These distances were computed in order to have the desired Talbot-Lau effect. The gratings were first placed parallel to each other by eye. Then for the angular alignment a first rough calibration was made with visible light. A laser and a piece of paper with horizontal line drawn on were used. The laser does diffract and reflect on the grating. It showed an interference pattern which was aligned to the horizontal lines drawn on the paper on which light was projected. That last macro-steps was redone each time the angular alignment was completely lost. As long as the gratings are not correctly displaced along the source-detector axis, the Moire pattern can be seen on detector's images. That effect due distances' misalignment was convenient for aligning the grating along all their direction. And it was also the reason why distance micro-alignment was the last step of the process.

The micro-alignment was used the fine tune the position of the gratings. The grating needed to be aligned one to the other, thus only G2 was moved during that process. The grating was tilted along all three axis until the Moire effect seen on the image was perfectly vertical with no unexpected shape. The strips were expected to have a rectangular shape, and a tilt along the either the x or the y axis would deform them to a rather triangular shape. The rotation along the y axis was responsible for rotating the strips. Once the gratings aligned along theirs three axis, the setup had to aligned in terms of distance between towers. For that the translation stage along the source-detector was used. The grating was moved until the strips disappeared. Once the gratings aligned, none should be able to tell that gratings are present in the system with a detector's image.

A concrete example of these optical effects can be seen in figure 12. In the left picture an almost perfectly aligned setup's image is shown, it looks as if nothing was in the field of view. In right one, an image with clearly visible Moire pattern is shown. That pattern was seen during alignment process, one would need to tilt the G2 grating in the counterclockwise sense to make the strips vertical.



(a) Aligned setup (b) Non aligned setup

Figure 12: The figure displays detector's images taken during grating alignment.

Grating alignment was done with four different G2 gratings and one G1. For G2 gratings selection, visibility measurements on the phase contrast image were taken. In order to retrieve phase contrast images and to assess the visibility, stepping scans were performed. The visibility values displayed in table 2 were used as relative measures in order to select the best G2 grating. Regarding at the visibility measurements, only one grating was excluded. The choice between the three other was done with other criteria. Au2503_AuBi454 had a too small field of view for the sample's size. Finally, TOMCAT_180711_p3000_5 was selected over the last one because the image looked more uniform.

Grating	Visibility [%]	Remarks
191204_p30003_Si_p3um_h45um_AuBi465	13	-
TOMCAT_180711_p3000_5	12	-
Au2503_AuBi454	11	Small can work as G0
180105_p3000 3	7	Visible strip defects

Table 2: The table display a relative visibility measurement with different G2 gratings tested during setup alignment. The nomenclature was inspired by what was written on the grating.

At some point during grating selection, the visibility dropped. Therefore, the electron beam of the source had to be scanned around its target to find the best area with highest visibility. After the source adjustment, the visibility has raised to the expected value. The histogram of that visibility is shown in figure 13.

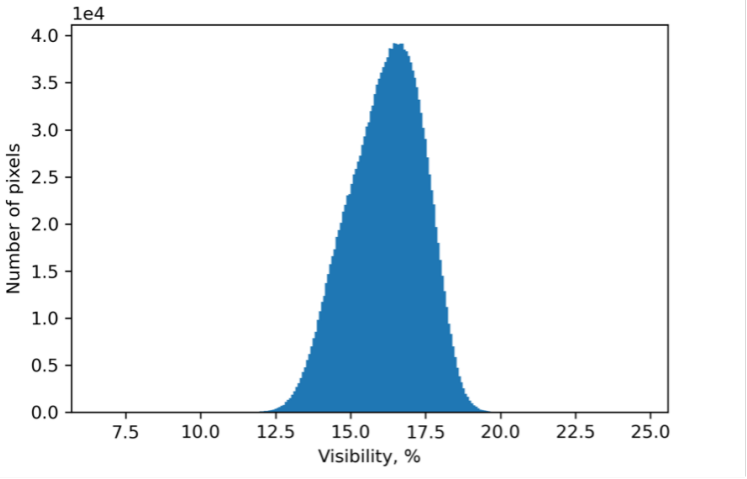


Figure 13: For the histogram, the region of interest corresponds to the position of the sample was used. The mean visibility of 16[%] and peak visibility of 25[%] were observed at the uniform visibility distribution.

In conclusion, all gratings have been aligned and compared. The visibility measures and other criteria allowed the selection of the grating which worked best in the GI setup.

5.2 Retrieval and alignment of the rotation axis

During tomography two rotation axis were used. The vertical rotation axis was the axis around which the sample was moving. The other rotation axis was the geometrical centre of the coordinates used during cone beam images reconstruction. For the cone beam image reconstruction, the pixel coordinate of the central ray was needed as well.

The cone beam rotation axis was aligned with the help of a pin. The idea was to have the horizontal plan of the sample being at all time of the tomography represented in the same row of pixels in the detectors image. That condition must be fulfilled to enable image reconstruction. The procedure was to set a pin on the Huber tower, to take an image at -90 degree, at +90 degrees and to tilt the stage until the tip of the pin stayed in the exact same pixel row in both position. Once that side was aligned, the same process was done to align it in the perpendicular direction. Figure 14 shows and example of pin being aligned along the source-detector axis. In practice *imagej* [6] was used and not a plot from a jupyter notebook.

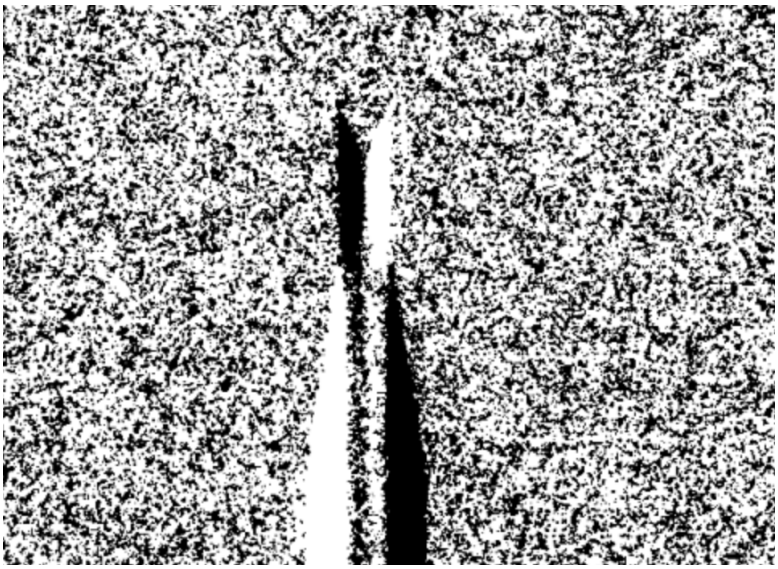


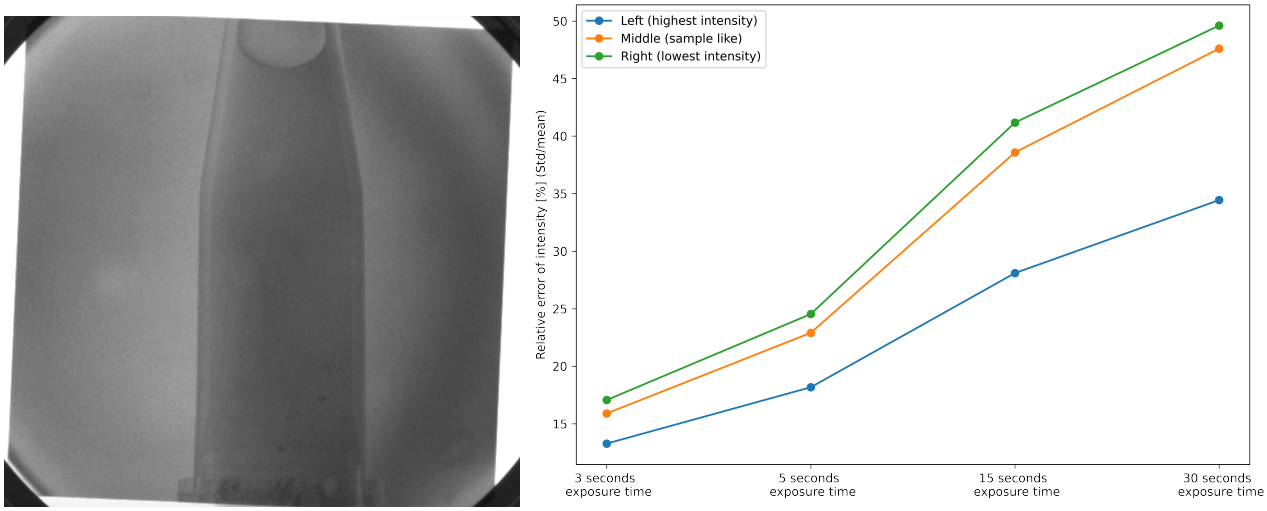
Figure 14: The figure show a picture taken during the process of aligning the pin. In the end the top of the pin should stay in the exact same pixel row.

In reality, for that setup the pin was aligned with a precision of 2-3 pixels. For the seek of comparing reconstruction, it is important to keep the sample cone beam rotation axis during different acquisition. For example in that experiment the tomographic scan with different exposure times were compared.

In order to retrieve the vertical axis, it was necessary to run tomographic acquisition. As a matter of fact, that axis retrieval was only done once arrived to the image reconstruction. Indeed, the vertical rotation axis was determined during tomographic reconstruction. The axis value providing the sharpest reconstruction image with no visible artefact was selected manually by using *imagej* [6]. On that software, the reconstruction images were opened and stacked. That method was convenient for the eyes to compare the sharpness between to very similar images.

5.3 Measure of the source stability

The motivation of characterising source's instability was its known intensity variation over time. The metric used to measure source's intensity fluctuation was the relative standard deviation of the intensity in different regions of an image. The process was done with the same exposure times as during real tomographic acquisitions. One of the images used during the measurements is shown in figure 15 (a). The sample was not moved during the complete experiment. In reality, there was no need of having a real sample on the image, the idea was to have different intensity regions. For each exposure time 100 images where taken in a row, then different region of interests were selected. Three regions were chosen; one of high intensity on the left of the picture, one inside the sample and one of low intensity on the right of the picture. Mean and standard deviation were calculated over all samples in the regions of interest. These measurements were then used for the calculation of relative standard deviation which was then plot according to exposure time in figure 15 (b).



(a) Image used for calculation

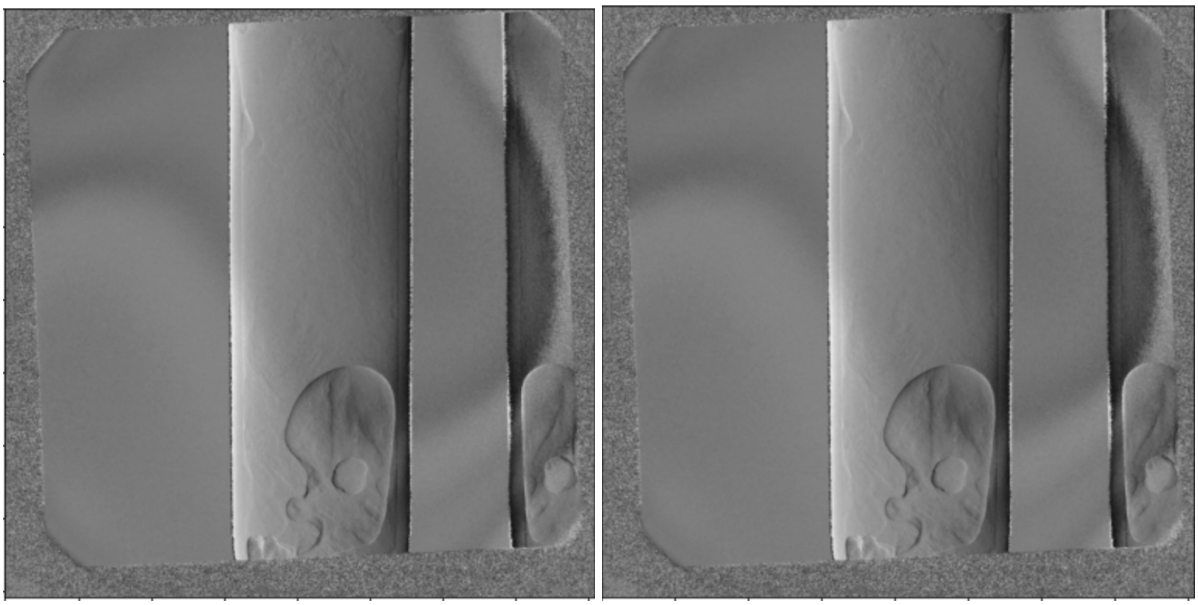
(b) Relative standard deviation plot

Figure 15: The figure shows the relative intensity error for three different regions of the image. The regions are meant to be of different intensity. The standard deviation divided by the mean intensity if the value of comparison between image exposure time. The measurements were done with a hundred images looking like the one in (a).

Even though the deviation of the intensity at one single projection can be high, it gets compensated by randomness of such error and the relatively large amount of projection used during the tomographic reconstruction. About 740 projections were typically taken during a tomographic acquisition. These two facts about tomographic scans helped to obtain reasonable contrast in the reconstructed images. The source instability still leaded to additional artifacts in the reconstruction domain, nevertheless they were compensated by strip removal algorithms. In all cases, that issue will be solved with the procurement of the new model of the source in 2022.

5.4 Phase retrieval

Phase retrieval is a necessary step for phase contrast imaging. At TOMCAT's laboratory two different phase retrieval algorithms are used. The difference between the two algorithms is the way they approximate the period of the grating. One of them uses a Fourier transform method and the other a least square method. The phase contrast image was successfully retrieved with both algorithms. An artifact was visible in both images, see figure 16, and the contrast of that artifact was the same in both cases. According to that measure, it was not possible to distinguish which algorithm worked better.



(a) Least square algorithm (b) Fourier algorithm

Figure 16: The figure shows the retrieved phase image with two different algorithms.

In conclusion, the choice of the algorithm in phase retrieval did not make a significant improvement on the visible artifact.

5.5 X-ray tomography with the new controller

The new controlling system, which is faster and preciser than the previous one, allowed the acquisition of a real tomographic experiment. Figure 17 shows the tomographic reconstruction of a cancerous breast sample which has been taken with the setup. That ultimate assessment of the proper functioning of Huber tower will lead to publication.

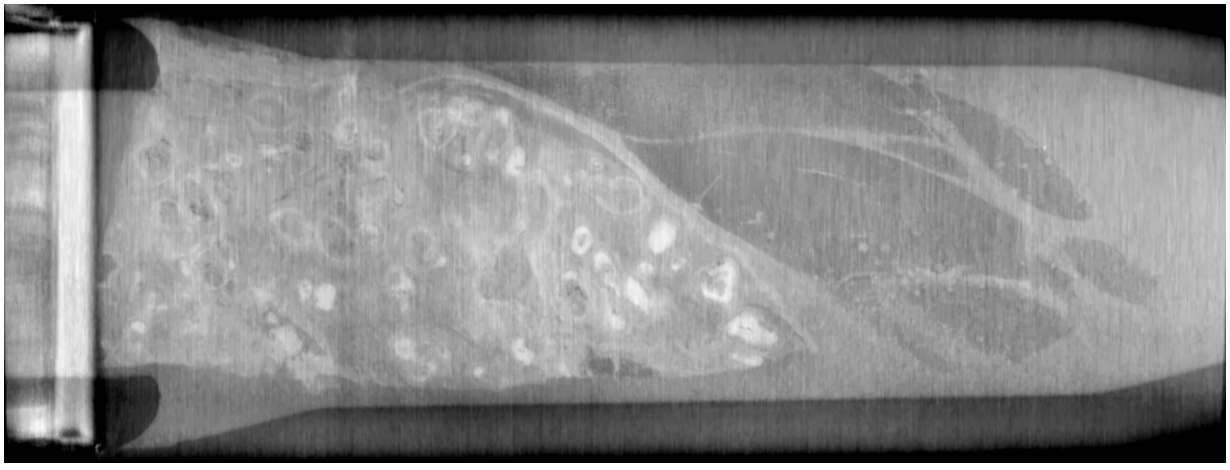


Figure 17: The figure shows a plane of a tomographic acquisition of a cancerous breast sample. The tomographic scan was performed with the tower's new controlling system.

6 Conclusion

The main objective of that project was successfully reached. The imaging system was made completely independent from PSI facility. The integration of the new API 3.4 to the existing pipeline, and the integration of the new controller-motor system to the setup made tomographic acquisition faster than previously. The ultimate proof of its working was the successful reconstruction of the cancerous breast sample displayed in figure 17.

In the second part of the project, the quality of tower's precision and its repeatability has been assessed. The running time of a tower's movement was optimised by more than a factor two. As a matter of fact, a tomographic acquisition which the new controller was 5 minutes faster than with the previous one. As part of a future optimization, one could try to implement the idea exposed in chapter 4.4 about time delay management. It may further improve relative time gain over a tomographic acquisition.

During the experimental work performed on the X-ray imaging system. I helped with aligning imaging setup, adjusting source parameters, selecting the best G2 grating and running a tomographic experiment. Subsequently, I helped with image analysis development, which was about assessing the position of the rotation axis and about the comparison between phase retrieval algorithms. That part of the project was successful as well. As a matter of fact, the aligned setup allowed to reach a visibility of 16 [%]. Meanwhile, the algorithms were both performing successful phase retrieval and none was better than the other according to contrast.

One can conclude that the project was successful because the new controlling system for Huber tower operated correctly and allowed the acquisition of real tomographic scans. In addition, the project allowed to strengthen my skills in the planning, execution and analysis of the tomographic experiments. It allowed me as well to learn about the implementation of GI and phase contrast.

7 References

References

[1] Contact - SmarAct, <https://www.smaract.com/contact>.

[2] Epics, motors records, <https://epics.anl.gov/bcda/synapps/motor/index.html>.

[3] Fing, <https://www.fing.com>.

[4] High-resolution multicontrast tomography with an x-ray microarray anodeâstructured target source, <https://www.pnas.org/doi/10.1073/pnas.2103126118>.

[5] Huber diffraction and positioning equipment, <https://www.xhuber.com/en/>.

[6] Imagej, <https://imagej.nih.gov/ij/download.html>.

[7] Pp-electronics smc software and controller's online documentation, <http://smc.pp-electronic.de/>.

[8] Recent advances in X-ray imaging of breast tissue: From two- to three-dimensional imaging | Elsevier Enhanced Reader.

[9] Rise in number of cancer patients coincides with decline in mortality - Swiss Cancer Report 2021 | Press release, <https://www.bfs.admin.ch/asset/en/19204988>.

[10] Socket, low level networking interface â python 3.10.4 documentation, <https://docs.python.org/3/library/socket.html>.

[11] Tcp client class, <https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcpclient>.

[12] X-ray sCMOS 16MP Detector.

[13] M. E. Kagias. Direct self-imaging methods for x-ray differential phase and scattering imaging. 1:258–261, Apr. 2017.

[14] F. S. Office. Cancer, <https://www.bfs.admin.ch/bfs/en/home/statistiken/gesundheit/gesundheitszustand/krankheiten/krebs.html>.

[15] F. Pfeiffer, T. Weitkamp, O. Bunk, and C. David. Phase retrieval and differential phase-contrast imaging with low-brilliance X-ray sources. *Nature Physics*, 2(4):258–261, Apr. 2006.

[16] V. D. Sankatsing, N. T. van Ravesteyn, E. A. Heijnsdijk, C. W. Looman, P. A. van Luijt, J. Fracheboud, G. J. den Heeten, M. J. Broeders, and H. J. de Koning. The effect of population-based mammography screening in Dutch municipalities on breast cancer mortality: 20 years of follow-up: The effect of population-based mammography screening on breast cancer mortality. *International Journal of Cancer*, 141(4):671–677, Aug. 2017.

[17] U. Veronesi, P. Boyle, A. Goldhirsch, R. Orecchia, and G. Viale. Breast cancer. *The Lancet*, 365(9472):1727–1741, May 2005.

8 Appendix A; API : MotorClass

The code displayed below is the script of MotorClass.

```
import socket
import sys
import time

class MotorClass:
    """
    The class MotorClass defines methods which can be used with the uber tower and SMC control system. It is designed to initialize one motor per object.

    Functions redudancy :
        - get_position = get
        - move = put
    """

    def __init__(self, axis = 3, homing = False):
        """
        Variables :
            - self.axis (int) : Put 1 for SAM_ROTX (tilt), 2 for SAM_ROTZ (tilt) and 3 for SAM_ROTY (rotation)
            - self.homing (bool) : True = homing procedure at initialisation (waiting is on)
            - self.HOST/PORT = server's hostname/IP address and port (may change)
        """
        self.axis = axis #1,2,3
        self.homing = homing
        self.HOST = "129.129.99.103" # Mac Address : 00:40:f2:39:07:5a/b
        self.PORT = 1234
        if self.homing : self.home()

    def waiting(self):
        """
            waiting() : Keeps the cell running as long as the motor is active.

            - Time out of 60 seconds
            - Returns and prints the error type (too long movement / no response from the controller)
        """
        command = "?status"
        command = command + str(self.axis) + " \r"
        start_time = time.time()
        stop = 0
        timeout = 60
        key = 'a'
        while (stop != int(1)) and ((time.time()-start_time) < timeout) and (stop != int(2)) :
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((self.HOST, self.PORT))
                s.sendall(command.encode())
                time.sleep(0.1)
                while (time.time()-start_time) < (timeout+2):
                    try :
                        data = s.recv(1024)
                        msg_test = data.decode()
```

```
                        pos_ = self.find_nth(msg_test,':', 8)
                        stop = int(msg_test[pos_+1])
                        break
                    except ValueError :
                        time.sleep(0.1)
                        if (time.time()-start_time) < (timeout) :
                            key = 'b'
            if ((time.time()-start_time) > timeout) :
                choices = {'a': "Waiting time limit for tower rotation has been reached.",
                           'b': "Waiting time limit for controller's response has been reached."}
                errorType = choices.get(key, 'default')
                print(errorType)
                return errorType
        return False

    def get(self):
        """
            get() : Returns and prints the encoder position of the motor.
        """
        command = "?e"
        command = command + str(self.axis) + " \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.05)
            data = s.recv(1024)
        msg = data.decode()

        pos_start = self.find_nth(msg,':',1)
        pos_end = self.find_nth(msg,';',1)
        msg[pos_start+1:pos_end]
        print("Position = " + msg[pos_start+1:pos_end])
        return msg[pos_start+1:pos_end]

    def change_acc_speed(self, number=6000):
        """
            change_acc_speed(number) : Modifies the start/stop speed of the motors.

            Variables :
                - number (int) : acceleration value optimized for the rotating axis (3)
        """
        command = "frun" + str(self.axis) + ":" + str(number) + " \r"

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            data = s.recv(1024)

    def change_slew_speed(self, number=35000):
        """
            change_dlew_speed(number) : Modifies the slew speed of the motors.

            Variables :
```

```
                - number (int) : acceleration value optimized for the rotating axis (3)
        """

        command = "ffast" + str(self.axis) + ":" + str(number) + " \r"

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            data = s.recv(1024)

    def getstatus(self):
        """
            getstatus() : Returns and prints the complete motor's status.

            The index is for the nth number (ex: 1:2:3:4:[...]:16:17)

            - Legend :
                1 axis number       Axis
                2 error number        ErrN
                3 error message       ErrM
                4 position        Pos
                5 encoder position      EPos
                6 limit switch status     LIMIT
                7 home position status      HOME
                8 encoder home/reference position status  EREF
                9 axis ready        Rdy
                10  oscillation in progress     [0|1]
                11  oscillation error     [0|1]
                12  continuous motion in progress   [0|1]
                13  program is running      [0|1]
                14  current configuration     [0|1]
                15  soft-limit status     SOFT LIMIT
                16  controller is blocked     [0|1]
                17  ext. stop through IN-port enabled [0|1]
        """
        command = "?status"
        command = command + str(self.axis) + " \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.05)
            data = s.recv(1024)
        msg = data.decode()

        pos_start = self.find_nth(msg,':',1)
        pos_end = self.find_nth(msg,';',1)
        msg[pos_start+1:pos_end]
        print("Status = " + msg[pos_start+1:pos_end])
        return msg[pos_start+1:pos_end]

    def put(self, degree = 0.0, wait = False):
        """
            put(degree, wait) : Puts the absolute position and has the option of letting the cell run until the process is done.

            Variables :
                - degree (double) : position value in degrees of a tilt or a rotation.
```

```
                - wait (bool) : choose wether or not the wait (True = wait).

            If wait = True, the function get() is called.
        """
        command = "goto" + str(self.axis) + ":" + str(degree) + " \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.1)
            data = s.recv(1024)
        if wait :
            time.sleep(0.1)
            self.waiting()
            self.get()


    def putr(self, degree, wait = False):
        """
            putr(degree, wait) : Puts the relative position and has the option of letting the cell run until the process is done.

            Variables :
                - degree (double) : position value in degrees of a tilt or a rotation.
                - wait (bool) : choose wether or not the wait (True = wait).

            If wait = True, the function get() is called.
        """
        command = "move" + str(self.axis) + ":" + str(degree) + " \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.1)
            data = s.recv(1024)
        if wait :
            time.sleep(0.1)
            self.waiting()
            self.get()

    def home(self, wait = True):
        """
            home(wait) : The function enables motor's homing and has the option of letting the cell run until the process is done. It will print the encoder's position.

            Variables :
                - wait (bool) : choose wether or not the wait (True = wait).
        """
        if int(self.axis) == 3:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((self.HOST, self.PORT))
                s.sendall("clr \r eref3:- \r nl \r end \r start\r".encode())
                data = s.recv(1024)
            if wait :
                time.sleep(0.5)
                self.waiting()
            self.putr(0.717, wait=True)
        else :
```

```
            #requires LIMIT+ and encoder ECZ signal will not work for axis 3 ...
            command = "clr \r home" + str(self.axis) + ":he;jg15000 \r nl \r end \r start \r"
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((self.HOST, self.PORT))
                s.sendall(command.encode())
                data = s.recv(1024)
            if wait :
                time.sleep(0.5)
                self.waiting()
            self.get()


    def find_nth(self, haystack, needle, n):
        """
            find_nth(haystack, needle, n) : Returns the occurence # 'n' of the caracter 'needle' in the string 'haysack'.

            Variables :
                - haystack (string) : variable in which it looks for the nth occurence of needle.
                - needle (cacarter) : reaserched caracter.
                - n (int) : the occurence in which it is interested.
        """
        start = haystack.find(needle)
        while start >= 0 and n > 1:
            start = haystack.find(needle, start+len(needle))
            n -= 1
        return start

    def anycommand(self, command = "?e1"):
        """
            anycommand(command) : allows to send any command from a python code and print the full message which is or not returned by the controller.

            Variables :
                - command (string) : takes the command, by default it asks for encoder position.
        """
        command = command + " \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.2)
            data = s.recv(1024)
            msg = data.decode()
        print(msg)

    def stop(self):
        """
            stop() : stops whatever the tower is doing.
        """
        command = "stop \r"
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((self.HOST, self.PORT))
            s.sendall(command.encode())
            time.sleep(0.1)
```

```
264             data = s.recv(1024)
265
266     def calibrate_eres(self, old_eres, should_be_position, encoder_position)
    :
267         """
268             calibrate_eres() : changes and prints the new value of eres.
    When step size needs to be calibrated, one would modify the value of eres
    . Over 360 degrees you expect an error of approx. 0.001 degrees. It works
     best when using positions which are quite far away. For that the
    old_eres is needed, the encoder position and the true position.
269
270             Variables :
271                 - old_eres = actual encoder resolution which you can in the
    axis configuration.
272                 - should_be_position = the true position which you want the
    tower to be.
273                 - encoder_position = the value which is displayed by the
    encoder (you can use get())
274         """
275         New_eres = old_eres * should_be_position / encoder_position
276         command = "eres" + str(self.axis) + ":" + str(New_eres)
277         command = command + " \r"
278         with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
279             s.connect((self.HOST, self.PORT))
280             s.sendall(command.encode())
281             time.sleep(0.1)
282             data = s.recv(1024)
283         print(New_eres)
284
285     move = put
286
287     get_position = get
```

9 Appendix B; Jupyter notebook : FAQ_huber_tower

01.05.2022 FAQ huber tower :

Useful scripts and notebook

- /sls/X02DA/Data20/e15889/Maxim_LCT/scripts/experiments
- /sls/X02DA/Data20/Maxim_LCT/anaconda3/lib/python3.6/site-packages/epics
  - motor.py and pv.py

Rotation motor

- https://catalog.orientalmotor.com/item/all-categories-components/all-categories-components-pk-series-no-cables/pkp223u09b#
  - $\pm$ 3 arc minutes ($\pm 0.05^{0}$) for the motor, don't know how it translated to the tower
- https://www.xhuber.com/en/products/1-components/12-rotation/1-circle-goniometers/408/
  - ** step motor, 1000 steps/revolution, 0.002-0.001 [^{o}]

Online documentation

- http://smc.pp-electronic.de/
  - Credentials : user, smc

Useful information :

Step size calibration is done with eres

- You modify it from the user interface on the SMC controller under 'axis configuration' or with method **calibrate_eres()**.
- Need to use that formula :
  - **New_eres = old_eres * should_be_position / displayed_position**
  - Tip : use a big step

Homing and its calibration

- Command for tilt axis : **home[axis]:he;jg15000**
- Command for rotation axis : **eref3:-** (because no limit switch)
- Commands for configuration of a home position offset : **erofs[axis]:[value]**

If SMC crashes

- It may happen for unknown reason
- If it becomes recurent, chek sleep times, wait and home function in huber.py, these are the main sources of errors
- a) Turn off and on the controller
- b) Run a homing procedure or set the zero by hand on the user interface

MAC address

- As you change the location, you may have to change Host and Port values (MAC ADDRESS : 00:40:f2:39:07:5a/b)

Source of errors

- There are several smc commands which do not work on Huber tower, be not surprised if it crasher while trying out.

In [2]:
```
import numpy as np
import os
import matplotlib.pyplot as plt
import socket
import sys
import time

# if you use a beamline console
sys.path.insert(1, '/sls/X02DA/Data20/e15889/Maxim_LCT/Eliot_Semester_2022/')

# if you use some local machine
sys.path.insert(1, '/mnt/Data20/Data20/Maxim_LCT/Eliot_Semester_2022/')

#from data_processing import *
import huber
```

In [2]:
```
help (huber.MotorClass)
```

```
Help on class MotorClass in module huber:
```

```
class MotorClass(builtins.object)
 |  MotorClass(axis=3, homing=False)
 |
 |  The class MotorClass defines methods which can be used with the uber tower and SMC control system. It is desi
gned to initialize one motor per object.
 |
 |  Functions redudancy :
 |      - get_position = get
 |      - move = put
 |
 |  Methods defined here:
 |
 |  __init__(self, axis=3, homing=False)
 |      Variables :
 |          - self.axis (int) : Put 1 for SAM_ROTX (tilt), 2 for SAM_ROTZ (tilt) and 3 for SAM_ROTY (rotation)
 |          - self.homing (bool) : True = homing procedure at initialisation (waiting is on)
 |          - self.HOST/PORT = server's hostname/IP address and port (may change)
 |
 |  change_acc_speed(self, number=6000)
 |      change_acc_speed(number) : Modifies the start/stop speed of the motors.
 |
 |      Variables :
 |          - number (int) : acceleration value optimized for the rotating axis (3)
 |
 |  change_slew_speed(self, number=35000)
 |      change_dlew_speed(number) : Modifies the slew speed of the motors.
 |
 |      Variables :
 |          - number (int) : acceleration value optimized for the rotating axis (3)
 |
 |  find_nth(self, haystack, needle, n)
 |      find_nth(haystack, needle, n) : Returns the occurence # 'n' of the caracter 'needle' in the string 'haysa
ck'.
 |
 |      Variables :
 |          - haystack (string) : variable in which it looks for the nth occurence of needle.
 |          - needle (cacarter) : reaserched caracter.
 |          - n (int) : the occurence in which it is interested.
 |
 |  get(self)
 |      get() : Returns and prints the encoder position of the motor.
 |
 |  get_position = get(self)
 |
 |  getstatus(self)
 |      getstatus() : Returns and prints the complete motor's status.
 |
 |      - Legend :
 |          1       axis number                           Axis
 |          2       error number                          ErrN
 |          3       error message                         ErrM
 |          4       position                              Pos
 |          5       encoder position                      EPos
 |          6       limit switch status                   LIMIT
 |          7       home position status                  HOME
 |          8       encoder home/reference position status  EREF
 |          9       axis ready                            Rdy
 |          10      oscillation in progress               [0|1]
 |          11      oscillation error                     [0|1]
 |          12      continuous motion in progress         [0|1]
 |          13      program is running                    [0|1]
 |          14      current configuration                 [0|1]
 |          15      soft-limit status                     SOFT LIMIT
 |          16      controller is blocked                 [0|1]
 |          17      ext. stop through IN-port enabled     [0|1]
 |
 |  home(self, wait=True)
 |      home(wait) : The function enables motor's homing and has the option of letting the cell run until the pro
cess is done. It will print the encoder's position.
 |
 |      Variables :
 |          - wait (bool) : choose wether or not the wait (True = wait).
 |
 |  move = put(self, degree=0.0, wait=False)
 |
 |  put(self, degree=0.0, wait=False)
 |      put(degree, wait) : Puts the absolute position and has the option of letting the cell run until the proce
ss is done.
 |
 |      Variables :
 |          - degree (double) : position value in degrees of a tilt or a rotation.
 |          - wait (bool) : choose wether or not the wait (True = wait).
 |
 |      If wait = True, the function get() is called.
 |
 |  putr(self, degree, wait=False)
 |      putr(degree, wait) : Puts the relative position and has the option of letting the cell run until the proc
ess is done.
```

```
|
|      Variables :
|          - degree (double) : position value in degrees of a tilt or a rotation.
|          - wait (bool) : choose wether or not the wait (True = wait).
|
|      If wait = True, the function get() is called.
|
|  waiting(self)
|      waiting() : Keeps the cell running as long as the motor is active.
|
|      - Time out of 60 seconds
|      - Returns and prints the error type (too long movement / no response from the controller)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

HowTo MotorClass (SMC controller) :

Waiting function

- Allow to let the cell run until tower's task is done unless reaching time limit error.
  - There are two different time limit errors :
    - Movement took to long.
    - Computer was not able to recieve controllers answer in time.

Initialisation

One can send commands to huber tower once that step is done.

- The homing procedure during initialisation calls waiting function by default, cell will run until its done or time limit reached.

In [8]:
```
# Tower initialisation :
SAM_ROTX = huber.MotorClass(axis = 1)
SAM_ROTY = huber.MotorClass()
SAM_ROTZ = huber.MotorClass(axis = 2)
```

Positionning methods and STOP

It is how you move the tower.

- put() && move() : **Absolute** movement
- putr() : **Relative** movement
- stop() : **Stops any operation**

- wait = False, by default
- wait = True, returns and prints position by calling .get() method

In [ ]:
```
# Absolute movement :
SAM_ROTX.put(0,True)
SAM_ROTY.put(degree=-90,wait=True)
SAM_ROTZ.put(0,True)
```

In [ ]:
```
# Relative movement :
SAM_ROTX.putr(0,True)
SAM_ROTY.putr(degree=5,wait=True)
SAM_ROTZ.putr(0,True)
```

In [ ]:
```
# Stops any tower's operation
SAM_ROTX.stop()
SAM_ROTY.stop()
SAM_ROTZ.stop()
```

Query methods

It is how you get information about the tower.

- get() && get_position() : returns and prints **encoder position**
- getstatus() : returns **complete status information** (legend in documentation)

In [7]:
```
SAM_ROTX.get()
SAM_ROTY.get()
SAM_ROTZ.get()
```

```
Position = -0.14112
Position = -90.00068
Position = 1.98867
```

Out[7]:
```
'1.98867'
```

In [3]:
```
help (huber.MotorClass.getstatus)
```

```
Help on function getstatus in module huber:

getstatus(self)
    getstatus() : Returns and prints the complete motor's status.

    - Legend :
        1       axis number                             Axis
        2       error number                            ErrN
        3       error message                           ErrM
        4       position                                Pos
        5       encoder position                        EPos
        6       limit switch status                     LIMIT
        7       home position status                    HOME
        8       encoder home/reference position status  EREF
        9       axis ready                              Rdy
        10      oscillation in progress                 [0|1]
        11      oscillation error                       [0|1]
        12      continuous motion in progress           [0|1]
        13      program is running                      [0|1]
        14      current configuration                   [0|1]
        15      soft-limit status                       SOFT LIMIT
        16      controller is blocked                   [0|1]
        17      ext. stop through IN-port enabled       [0|1]
```

In [8]:
```
SAM_ROTX.getstatus()
SAM_ROTY.getstatus()
SAM_ROTZ.getstatus()
```

```
Status = 0::-0.15:-0.14111995:0:0:0:1:0:0:0:0:0:0:0:0

Status = 0::-90:-90.0006820016057:0:0:0:1:0:0:0:0:0:0:0:0

Status = 0::2:1.988672:0:0:0:1:0:0:0:0:0:0:0:0
```

Out[8]:
```
'0::2:1.988672:0:0:0:1:0:0:0:0:0:0:0:0\r\n\r'
```

Configuration methods

These method allow to change motor's configuration.

- change_acc_speed() : **modifies the acceleration speed** of the motors.
- change_slew_speed() : **modifies the maximum slew speed** of the motors.
- calibrate_eres() : **modifies the encoder resolution**, fine tunes the step size.

- Default values are 6000 and 35000, optimized for the rotation axis SAM_ROTY, check ??? notebook.
- Read 'Useful information' for more details of eres

In [ ]:
```
SAM_ROTY.change_acc_speed(number=6000)
```

In [ ]:
```
SAM_ROTY.change_slew_speed(number=35000)
```

In [ ]:
```
SAM_ROTY.calibrate_eres(old_eres = 10.4904, should_be_position = 15.0,
                        encoder_position = SAM_ROTY.get())
```

You can get axis configuration with that command : (for knowing encoder resolution)

In [14]:
```
SAM_ROTY.anycommand("?conf3")
```

```
smc 1.2.1129
?conf3
# configuration settings of axis 3
alias3:3~rotating_basement_y
type3:0
unit3:deg
gnum3:2000
gden3:1
mres3:0.0005
dcpl3:4
rofs3:0
blp3:0
bln3:0
mdir3:0
lsa3:0
lsnf3:0
ofs3:100000
rfs3:10000
hsdm3:0
sdm3:1
sdpw3:0
slen3:0
slneg3:0
slpos3:0
frun3:6000
ffast3:35000
acc3:10
dec3:10
macc3:10
mdec3:10
emips3:-1
cf3:1
corr3:0
ecl3:1
eclst3:1
ecp3:0
ect3:1
edev3:0.01
edir3:0
ehst3:0.5
eias3:1
emode3:0
erofs3:-0.284
eres3:1.09864247883122E-04
esh3:1
esm3:4
est3:1
mpr3:250
prst3:0.02
prt3:0.5
twen3:0.0
biss_if3:0
biss_id3:
biss_ch3:0
biss_e3:0
biss_i3:0
biss_t3:360
```

Homing procedure

It is how one can run homing for the Huber tower.

- By default, waiting function is activated.
- It **homes and puts to position 0.00**

In [ ]:
```
SAM_ROTX.home(wait = True)
SAM_ROTY.home(wait = True)
SAM_ROTZ.home(wait = True)
```

Sending any command

Allows to send whatever command you would like to.

- The ' \r' is alreay implemented
- It returns the whole messages printed by the controller (not robust at all)

In [11]:
```
SAM_ROTX.anycommand("?status") # status with no selected axis returns everything
SAM_ROTY.anycommand("?status")
SAM_ROTZ.anycommand("?status")
```

```
smc 1.2.1129
?status
1:0::-0.15:-0.14114988:0:0:0:1:0:0:0:0:0:0:0:0:0
2:0::2.1:2.0890992:0:0:0:1:0:0:0:0:0:0:0:0:0
3:0::123.3995:123.40018240781:0:0:0:1:0:0:0:0:0:0:0:0:0
4:0::0:0:0:0:0:1:0:0:0:0:0:0:0:0:0


smc 1.2.1129
?status
1:0::-0.15:-0.14114988:0:0:0:1:0:0:0:0:0:0:0:0:0
2:0::2.1:2.0890992:0:0:0:1:0:0:0:0:0:0:0:0:0
3:0::123.3995:123.40018240781:0:0:0:1:0:0:0:0:0:0:0:0:0
4:0::0:0:0:0:0:1:0:0:0:0:0:0:0:0:0


smc 1.2.1129
?status
1:0::-0.15:-0.14111995:0:0:0:1:0:0:0:0:0:0:0:0:0
2:0::2.1:2.0890992:0:0:0:1:0:0:0:0:0:0:0:0:0
3:0::123.3995:123.40018240781:0:0:0:1:0:0:0:0:0:0:0:0:0
4:0::0:0:0:0:0:1:0:0:0:0:0:0:0:0:0
```

In [6]:
```
# absolute position
SAM_ROTY.anycommand("goto3:10")
```

```
smc 1.2.1129
goto3:10
```

In [15]:
```
SAM_ROTY.anycommand("?ip")
```

```
smc 1.2.1129
?ip
129.129.99.103
```

Implementation in LabStetup

In [ ]:
```

```

epics.PV or epics.Motor (PSI controllers)

If for some reason one need to plug it back to PSI controllers, that is how you control them.

epics.Motor

- Use m = epics.Motor('X02DA-LAB-XIFM:SAM_ROTY')
- Use **readback=1** to get encoder's position !

epics.PV

- SAM_ROTY = epics.PV('X02DA-LAB-XIFM:SAM_ROTY')

In [10]:
```
import epics
```

epics.Motor

In [5]:
```
# Record postion with encoders from PSI controllers :
X = epics.Motor('X02DA-LAB-XIFM:SAM_ROTX')
Y = epics.Motor('X02DA-LAB-XIFM:SAM_ROTY')
Z = epics.Motor('X02DA-LAB-XIFM:SAM_ROTZ')

print('Encoder position of axis X:',X.get_position(readback=1))
print('Encoder position of axis Y:',Y.get_position(readback=1))
print('Encoder position of axis Z:',Z.get_position(readback=1))
```

```
Encoder position of axis X: -0.140865
Encoder position of axis Y: -0.00352576
Encoder position of axis Z: 0.7987744999999999
```

In [ ]:
```
# absolute position
Y.move(10, wait = True)

# relative position
Y.move(10, relative = True)
```

epics.PV

- Did not find a way to get encoder position...

In [ ]:
```
X = epics.PV('X02DA-LAB-XIFM:SAM_ROTX')
Y = epics.PV('X02DA-LAB-XIFM:SAM_ROTY')
Z = epics.PV('X02DA-LAB-XIFM:SAM_ROTZ')

print('Desired position of axis X:',X.get())
print('Desired position of axis Y:',Y.get())
print('Desired position of axis Z:',Z.get())
```

In [ ]:
```
# absolute position
Y.put(10, wait = True)

# relative position
# did not find how to ...
```

Low level control :

The names are inspired by the user manual

All these commands can also be written into the user interface of the SMC controller

- If done manually, no need od \r, it works as 'enter'

In [ ]:
```
import socket
```

In [ ]:
```
HOST = "129.129.99.103"  # The server's hostname or IP address
PORT = 1234  # The port used by the server
```

Direct command :

"move[axis]:[distance] \r" for relative movement

"goto[axis]:[position] \r" for absolute movement

"eref:[axis][direction] \r" brings back to reference position, **a way of homing SAM_ROTY**

Positive rotation sense :

- Axis X : pos = horaire
- Axis Z : pos = horaire
- Axis Y (top view): pos = anti-horaire

In [ ]:
```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall("move3:7 \r".encode())
    data = s.recv(1024)
```

Program commands :

- A smc program line usually consists of several command lines, terminated by the command line nl which indicates the end of a program line
- By clicking start, it reruns the last program entered
- **Example :** 3:10.0s[start-stop speed][max slew speed]
- 3:10 \r 2:1.0s500r2500a10 \r

In [ ]:
```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall("clr \r 3:10 \r 2:1.0s500r2500a10 \r delay5 \r nl \r end \r start \r".encode())
    data = s.recv(1024)
```

Configuration commands :

- Shows homing :
  - **It will not work for rotation axis because LIMIT SWITCH is needed**
  - home[axis]:he;jg15000

In [ ]:
```
# homing procedure for SAM_ROTX (WORKS ALSO FOR SAM_ROTZ WITH 2 INSTEAD OF 1)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall("clr \r home1:he;jg15000 \r nl \r end \r start \r".encode())
    data = s.recv(1024)
```

In [ ]:
```
# Homing procedure for SAM_ROTY (NO LIMIT SWITCH)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((HOST, PORT))
                s.sendall("clr \r eref3:- \r nl \r end \r start\r".encode())
                data = s.recv(1024)
```

Query commands :

- ?e return encoder position of all motors
- ?status returns the status of all motors

In [ ]:
```
command = "?e3  \r"
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(command.encode())
    time.sleep(0.1)
    data = s.recv(1024)
msg = data.decode()

print("?e3 = " + msg)
```