NTNU
Kunnskap for en bedre verden

TMR4345 - Marine Computer Science Lab

Dynamic Positioning

Author:
Petter Hangerhagen

May 2022

Summary

This rapport summarizes a dynamic positioning project in TMR4345 - Marine Computer Science Lab. The rapport contains how the lab was setup, the theory used, the implementation process, the results of a complete program and a discussion of the result and errors.

The vessel operates in one dimension with a PID-controller used to control it. A important process when working with a PID-controller is tuning. The program is written in python code, which made it quite simple to make a PID-algorithm. However to increase the complexity of the program some more part where added. A live plot function was implemented, which live plots the position of the vessel compared to the reference position. A graphical user interface was also implemented to make the program easier use.

The resulting PID-values was $K_P = 220$, $K_i = 95$ and $K_d = 140$. These values resulted in a little overshoot, but the controller found the reference position after roughly 7 seconds. The system was also tested in many other ways, to see how stable the system was with these parameters.

Table of Contents

List of Figures

List of Tables

1 Introduction

The control systems on today's ships are complex systems. A lot of vessels has dynamic position (DP) systems which makes a ship stand still on a given position. A real DP-program has to take six degrees of freedom into account. However this project addresses a simple DP-program working in only one dimension. The program is written in python, and the implementation of the project started with controlling the Phidget cards, setting the thrust and reading the position. After this a PID-algorithm was made with a proportional part (Kp), integral part (Ki) and a derivative part (Kd). When these values was tuned, a graphical user interface (GUI) was made to make the program easier to use. Finally GLWiew is used to make a digital visualization of the movements of the vessel.

The pool on the lab was quite small which made it important to set the start and end position correctly, therefor a startup program will be running every time the program starts to read these positions. The vessel had a box with a mass dragging the vessel backwards, working as an external force. With only this force working, the boat didn't need any reverse throttle, since the force was dragging the vessel backwards. The Phidget card reading out the position is only reading approximately every 0.25 second which limits the system to only update every 0.25 second.

2 Lab setup

The lab setup consists of a remote control boat in a basin, with a rope in the bow and the stern limiting the movements. The vessel is controlled by setting the speed of the propeller through a Phidget servo motor card. The position is read out from a Phidget interface card [1]. It reads the voltage change of the potentiometer hanging on the right end of the basin.

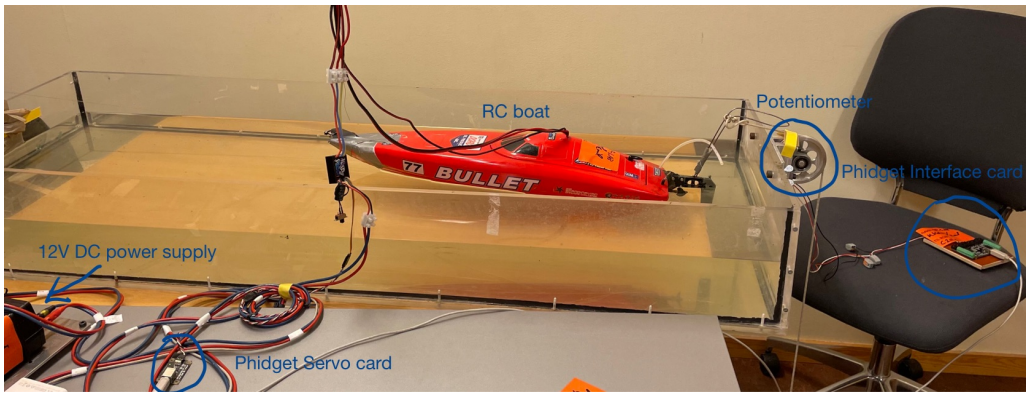


Figure 2.1: Lab setup

2.1 PhidgetAdvancedServo 1-Motor

The PhidgetAdvancedServo 1-Motor card allows to control the position, velocity, and acceleration of the RC servo motor [2]. The motor in the boat is powered by a 12V DC power supply. It's connected to the computer by a USB cable.

2.2 PhidgetInterfaceKit 8/8/8

The inputs to the PhidgetInterfaceKit 8/8/8 card are used to measure continuous voltage outputs generated by various sensors [1]. The sensor in this setup is a potentiometer which is a three-terminal resistor with rotating contact that forms an adjustable voltage divider [3]. In this case it results in a voltage change if the pulley is rotated. This voltage change can be read out from the Phidget card which is connected to the computer by a USB cable as well.

3 Theory

3.1 Dynamic positioning

Dynamic position is a computer control system making it possible for a marine vessel to automatically maintain a specific position using it own propellers and thrusters [4]. The advantage with a DP-system is that it gives good maneuverability and has a quick setup. The old way with using anchors can be difficult in specific cases where the water depth is large or the seabed is occupied by other equipment. Therefor a DP-system on an offshore vessel is highly useful. There exist DP-systems that are based on PID-controllers today, but moderns systems use a mathematical model of the ship to calculate the position and heading. The DP-system addressed in this project is based on a PID algorithm.

3.2 PID algorithm

The theory for this project is based on a simple PID-algorithm. The explanation of the algorithm will be referred to as it works for a DP-system. However there exist many other systems in the world using the same algorithm.

PID stands for proportional, integral and derivative. This is the tuning parameters for the system, and will be thoroughly explained. In figure (3.1) a block diagram for a PID-controller can be seen. If the PID-parameters are tune, the algorithm only needs one input parameter and that is the reference position $r(t)$. The algorithm also needs the output from the system (position of the vessel in the DP-case), but this is connected in a feedback loop. The difference between the the reference and the position is called the error $e(t)$. The error or the deviation is the value which is processed by the PID-terms. After the error is processed, the output of the contributions are summed up and sent to the system that is going to be controlled. In the DP-case this is the speed of the propeller. The vessel then outputs a new position and the loop starts again.

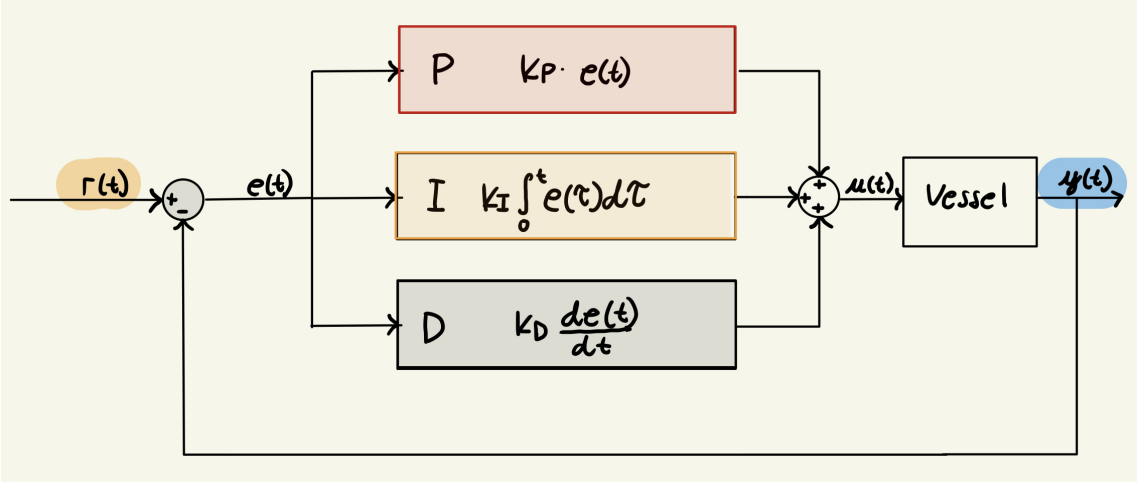


Figure 3.1: Block diagram for PID-controller

3.2.1 Proportional part

The proportional part is simply a tuned value multiplied by the error $e(t)$, see equation (3.1). A controller with the proportional part alone would work well for a system not effected by any external forces. In a case like that the error would linear decrees and the system would stop giving errors when the reference value was reached. However, if a system has an external force applied the controller would stop the vessel before it reaches the reference, a so called steady state error [5], see figure (3.2). For the system tested in this project this is the case, because the box hanging on the

end of the tank is effected by gravity and this will be a constant force working on the vessel. The force applied from the propeller can't be zero because then the vessel would be drag backwards by this external force. Nevertheless the proportional controller alone will get the vessel to a position where the proportional part is equal to the external force, therefor a steady state error [6]. An example of a single P-controller will be showed in section (5.4.1).

$$P = Kp \cdot e(t) \tag{3.1}$$

High proportional gain results in a large change in the output for a given change in the error, and a small gain results in a small output response to a large input error [6]. So it's important to tune this value correctly to get a system that response in a fast way, but not too fast because it can make the system unstable.

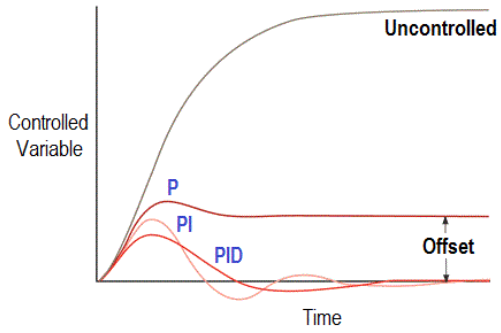


Figure 3.2: PID-values compared [7]

3.2.2 Integral part

The integral part is also a tuned value, but it's not simply multiplied by the error. This part is integrating the error from the start time to the present time, in other words it's summing the errors along the way, see equation (3.2). It will most likely cause an overshoot of the reference, but when it overshoots it will get a negative error which will end up decreasing the total sum, which ends up lowering the contribution from the integral part. This might happen sometimes back and fort til it find the reference position in the end. In figure (3.2) it can be seen that the PI part is oscillating a bit before finding the reference position. The integral part will always contribute a output, and when it reaches the desired position the integral part will be the only one with a contribution. It calculates the amount of output needed to eliminate the steady state error.

$$I = Ki \cdot \int_0^t e(\tau)d\tau \tag{3.2}$$

3.2.3 Derivative part

The proportional and integral part manages to get the vessel to the desired position, but the vessel could use a while to get there, or the vessel might have big oscillations to get there. This depends on how the parameters are tuned. To make it possible to tune the parameters aggressive a derivative part is needed. This will lower the oscillations and the reference position will be reached rather quickly. The derivative of the error finds the rate of change of the error, see equation (3.3). So if the system is approaching the error with a high rate of change the derivative part will contribute a large negative output, damping the total output.

$$D = Kd \cdot \frac{de(t)}{dt} \tag{3.3}$$

The derivative action predicts system behavior and thus improves settling time and stability of the system [6]. However the derivative term is sensitive to external disturbances. To deal with this a low-pass filter or other type of filter could be added. This is not added in the project because it was not needed. The results of the three parts can be seen as the PID part in figure (3.2).

3.3 Tuning

An import part of a setting up a PID-controller is tuning of the parameters. A PID-controller is tuned optimally, if the device minimizes the deviation from the set point, and if it responds to disturbances or reference point changes quickly but with minimal overshoot [8]. There are different ways of doing this, and in most cases it depends on the process that is going to be tuned. Two methods for tuning will be describe where the first one is a tuning method (Ziegler–Nichols method), and the second is manual tuning. Today it's normal for PID-controllers to have an auto-tune system integrated. An auto-tune system lets the controller tune itself depending on the response of the system [8].

3.3.1 Ziegler-Nichols method

Ziegler–Nichols method is a widely known tuning method. The method starts with setting K_i and K_d to zero, and then gradually increase K_p until the system reaches a state where the system is oscillating around the steady state error with constant period T_p, and K_u is set equal to K_p. The PID-values is then set to the values given in table (3.1), depending on which kind of controller is wanted [9].

Control type	Kp	Ki	Kd
P	$0.5K_u$	0	0
PI	$0.45K_u$	$0.54\frac{K_u}{T_u}$	0
PD	$0.8K_u$	0	$0.10K_uT_u$
classic PID	$0.6K_u$	$1.2\frac{K_u}{T_u}$	$0.075K_uT_u$
PID some overshoot	$0.33K_u$	$0.66\frac{K_u}{T_u}$	$0.11K_uT_u$
PID no overshoot	$0.20K_u$	$0.40\frac{K_u}{T_u}$	$0.066K_uT_u$

Table 3.1: Ziegler-Nichols method [9]

3.3.2 Manual tuning

Manual tuning is something everyone can do. The problem with it, is that the system needs to be available for a tuning process. If the person tuning has low experience, the process may take a while, which also can be a problem. However there exists guidelines on how to tune, and one simple method will be explained.

The process should start with increasing K_p until the output is oscillating a bit. Next increase K_i until the steady state error is eliminated and finally increase K_d until the reference is reached in a sufficient amount of time [6]. In table (3.2) it can be seen how the system is effected by increasing the parameters.

Parameter	Rise time	Overshot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Decrease
K_i	Decrease	Increase	Increase	Eliminate	Decrease
K_d	Minor change	Decrease	Decrease	no effect	Depends on the size

Table 3.2: Effect of increasing the tuning parameters [6]

4 Implementation

The project is written in python code in the Microsoft Visual Studio Code editor on a mac OS. The code is run by a main functions which calls code from other files. To how a closer look at how the code is implemented it can be seen in Appendix (A), (B), (C), (D), (E) and (F).

The actual plan for the project was to work in C or C++, but encountered issues with the setup on mac OS. Instead of using much time to fix the setup, it was determined to used python which is a well known language and was already working on the computer.

4.1 main.py

The main script consist of the main function. It starts with checking if the Phidget cards are connected, if so, it initializes them and reads out the initial parameters needed from a text file. Then a start up function will be run which sets the boundaries of the tank. Next the PID functions starts, which is the controller. It works for a time set in initial parameters. After that the result is written to text files, plotted and saved. The results are also written to a VTF-file.

4.2 PID.py

The PID script consist of the PID function. This function needs the following input; Kp, Ki, Kd, timeout, dt, r, rc and ch. Where Kp, Ki and Kd are the tuning parameters, timeout is the duration, dt is the integration step, r is the reference position, rc is the RCServo and ch is the VoltageRatioInput. The function is simply running a while loop for the wanted duration and calculates the PID values continuously. The function also saves all the results to different text files. The voltage can only be read about every 0.25 seconds, so the while loop also has a sleep part added to slow it down to approximately get it to calculate the values every 0.25 seconds.

4.3 utilities.py

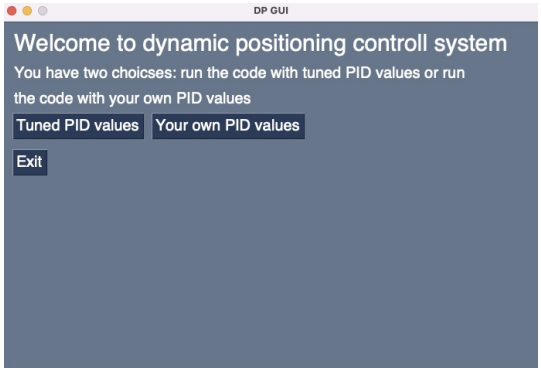
The utilities script consists of all the support functions needed. It includes a start_up function, many functions which treats the text files, a get_pos function, a set_thrust function, a thrust_scaling function, plot functions and a make_vtf_file function. It also includes a load_image function which is a support function to the GUI implementation. The last important function included is subprocess_call, which is used to run a shell script.

4.4 live_plot.py

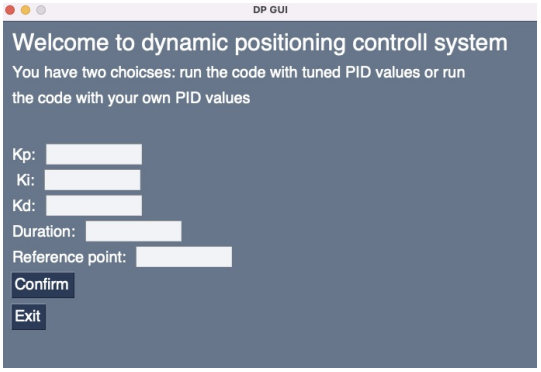
The live plot script consists of only the live plot function. This function reads the position and reference from the text files, simultaneously with them being updated by the PID function. This makes it possible to have a live animation of the DP control system working on the vessel, and has been very useful during manual tuning.

4.5 GUI.py

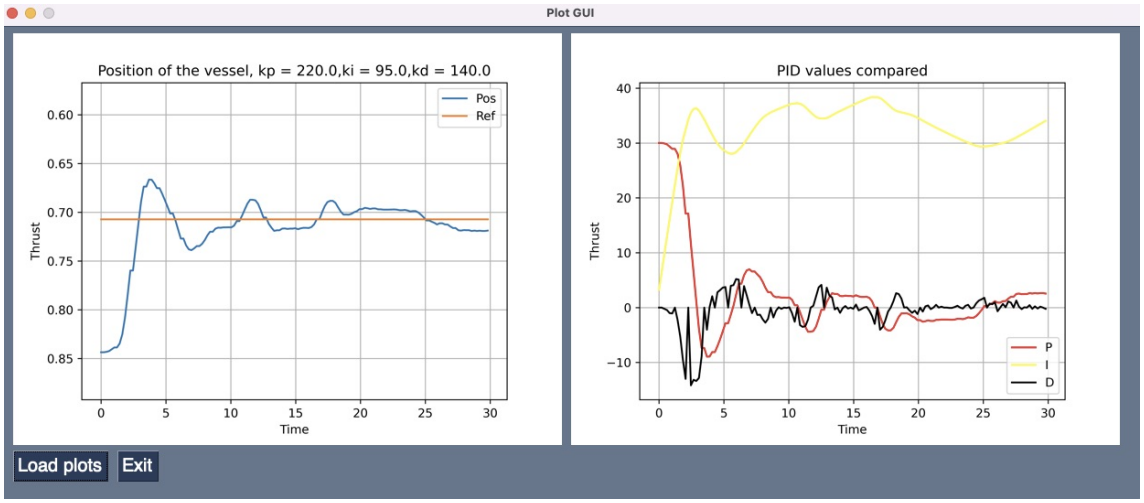
To make the program more user friendly it was decided to make a graphical user interface (GUI). The GUI script is based on the PySimpleGUI-library. It made it fast and simple to setup a all right looking GUI. The GUI gives the possibility to use predetermined values, or to type in the values yourself. When the program is finished it gives the possibility to load the plots saved and have a closer look at the results.



(a) GUI start up window



(b) GUI input window



(c) GUI plot window

Figure 4.1: GUI

4.6 run.sh

The last script of the program is a shell script. This is used to get the possibility of running two scripts at the same time. The need for this was discovered when the live animation inside the PID function caused a delay which disturbed the voltage readings. It runs live_plot.py and main.py separately and simultaneously. At the start the terminal had to be used to run the shell script, but after a while it was discovered that this could be done inside python as well. This is what the subprocess_call function does.

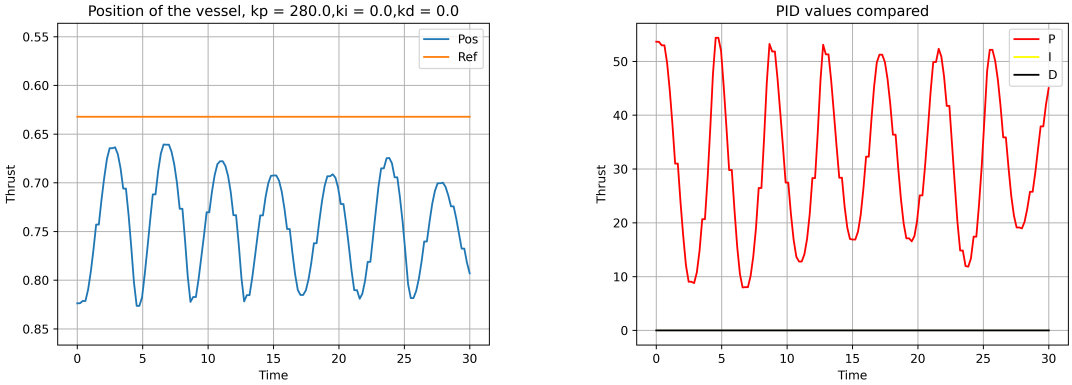
5 Results

The results presented is divided into several parts. The first case is the results when the algorithm is tuned by the Ziegler-Nichols method. The second case is the results of manual tuning, which was the desired way of tuning the parameters. This two cases are tuned with the initial setup of the lab. The third case includes different cases testing the roughness of the tuned parameters. The last case viewed is a visualization of the vtf file in GLview.

5.1 Ziegler-Nichols method

The Ziegler-Nicholas method was actually performed after manual tuning, but the results of the tuning method will be showed before, because that is how it should have been. The setup are following how it's describe in section (3.3.1).

The method started with getting a K_u-value where the vessel oscillates with a roughly constant period T_p. It ended up with $K_u = 280$ and $T_p = 4s$, determined from the test in figure (5.1).



(a) Position and reference

(b) PID values contribution

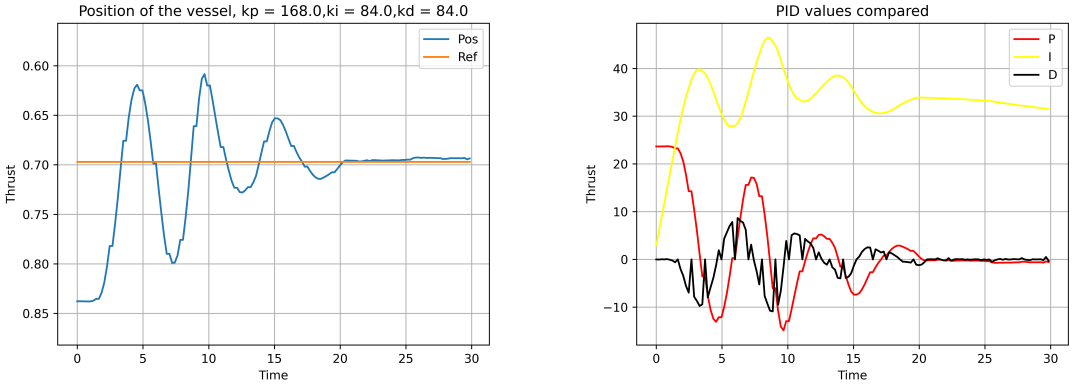
Figure 5.1: Results of K_u and T_p for Ziegler-Nichols method

The final PID-values are determined from table (3.1) where it was chosen to test a classic PID. The calculated values can be seen in table (5.1).

Parameter	value
K_p	168
K_i	84
K_d	84

Table 5.1: Ziegler-Nichols tuned values

The result of the tuned values can be seen in figure (5.2).



(a) Position and reference

(b) PID values contribution

Figure 5.2: Results of Ziegler-Nichols method

As seen the method produce a controller which works all right. It oscillates a bit, but the result was better then expected.

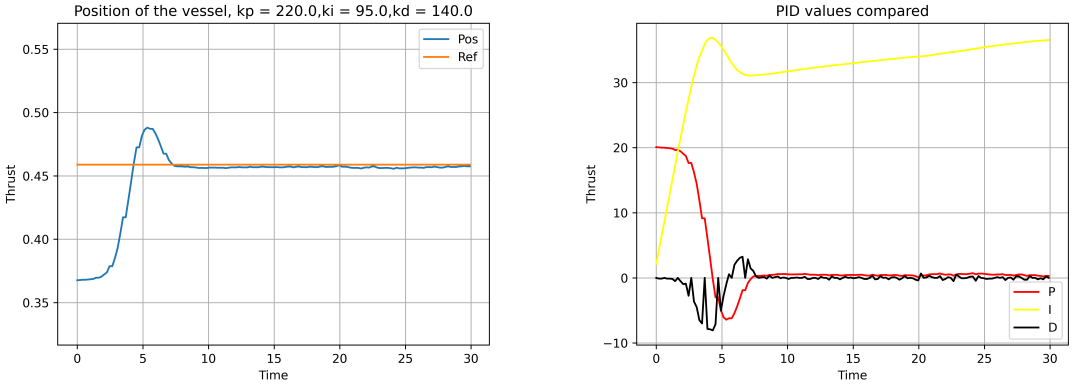
5.2 Manual method

After the code was complete, the tuning process started, and the resulting values can be seen in table (5.2). To have the possibility of tuning K_p and K_i relatively aggressive, the K_d value had to be tuned roughly high. The system never became unstable during testing, even if K_d was high.

Parameter	value
K_p	220
K_i	95
K_d	140

Table 5.2: Manual tuned values

During the tuning process the system sometimes seemed stable, but suddenly a test showed that it wasn't the case. Therefor the values were tested multiple times to be sure these values gave a stable system. The result of a test can be seen in figure (5.3). Note that it says thrust on the y axis of figure (5.3a), but it is actually the position as voltage from the potentiometer. It is like this for all figures in the result section.



(a) Position and reference

(b) PID values contribution

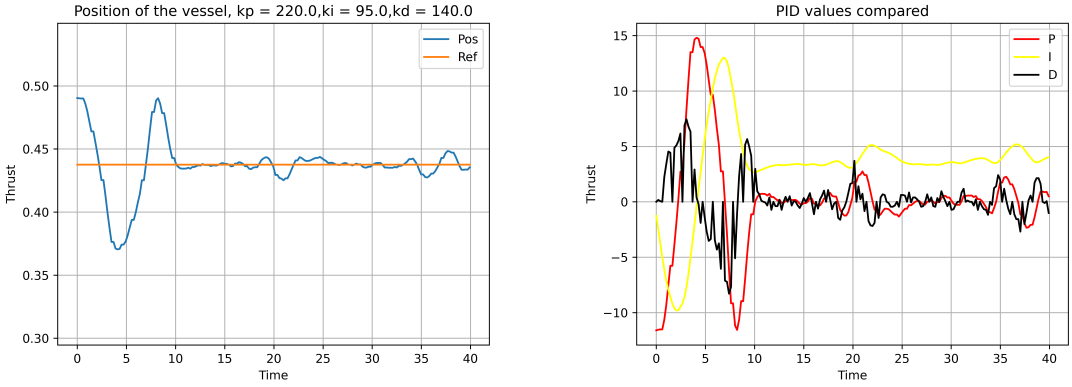
Figure 5.3: Results manual tuning

5.3 Testing the roughness of the controller

To be sure the PID-values chosen was stable with disturbances and changes in the reference position, more tests needed to be taken. This sections displays different tests. The controller is tuned for the case in section (5.2), so it works clearly best for this case, but works all right for the cases presented here.

5.3.1 Lower weight in the box

The first test was to see how the controller reacted to a change in the external load. To carry out this test the weight in the box was set to approximately half of what it was initially. The results can be seen in figure (5.4).



(a) Position and reference

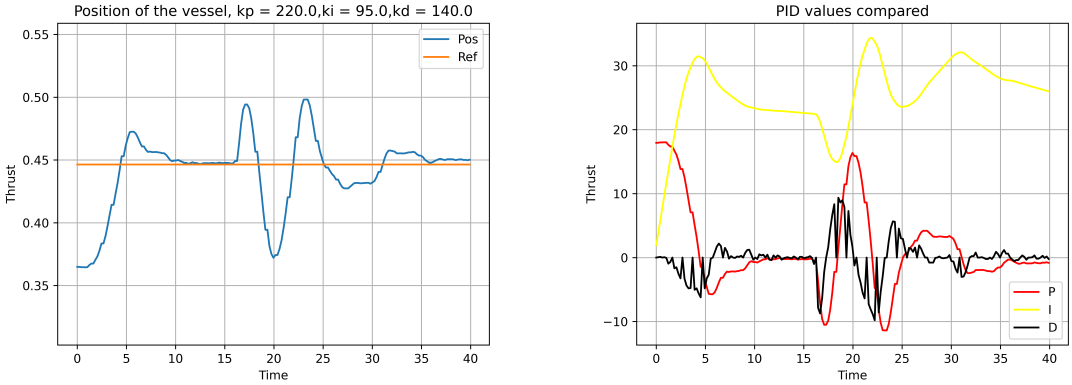
(b) PID values contribution

Figure 5.4: Result with lower weight

With a closer look it can be seen that the vessel starts from a position longer up in the basin. This is because the start_up function had too big value on the reverse throttle and wasn't able to go back to start. However it seemed like a good addition to test the controller even more.

5.3.2 Giving the vessel a push

To test how the controller deals with an external disturbance, a test where the vessel was getting a rough push was performed. The results of this can be seen in figure (5.5).



(a) Position and reference

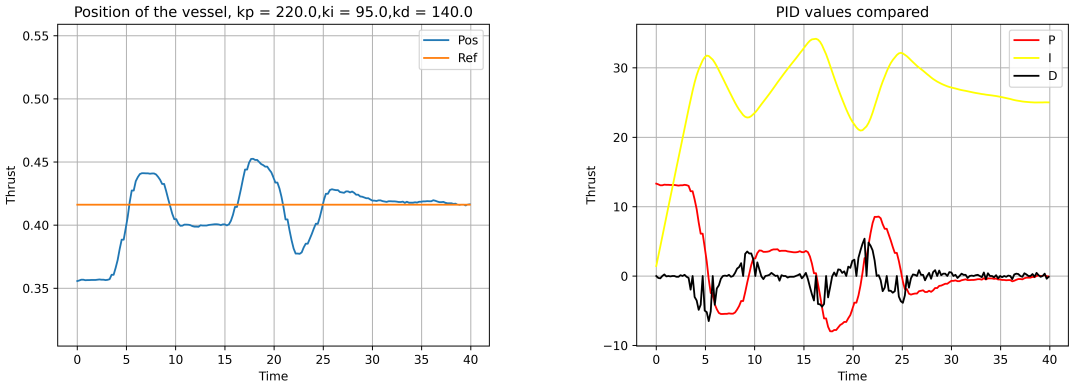
(b) PID values contribution

Figure 5.5: Result with a push given

From the figure it seems like the push happen at roughly 18 seconds, and the controller regulates the vessel back to the position after about 20 seconds.

5.3.3 Different reference

The third test was to try giving the controller a different reference from what it was initially tested with. The result for this can be seen in figure (5.6)



(a) Position and reference

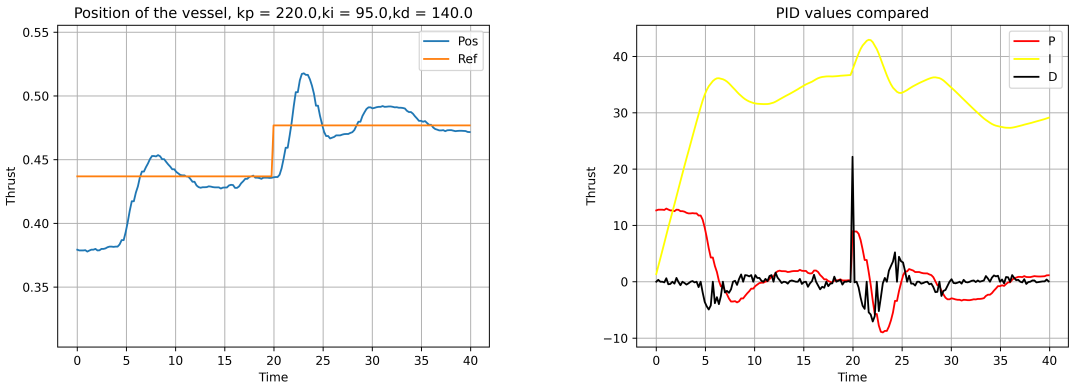
(b) PID values contribution

Figure 5.6: Result with different reference

The controller is struggling a bit more with a different reference, but it finds the desired position and the P- and D-values dies out after roughly 30 seconds.

5.3.4 Change of reference during a test

The next test was to check how the controller responded to a change in reference during try out. The result can be seen in figure (5.7).



(a) Position and reference

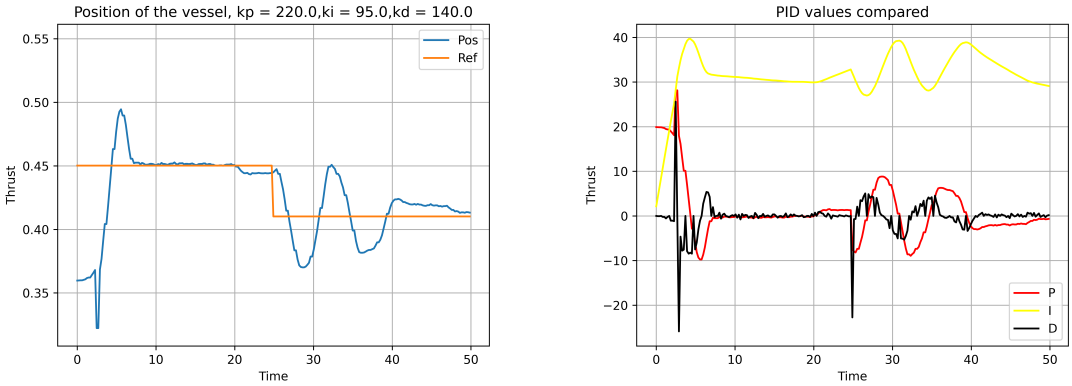
(b) PID values contribution

Figure 5.7: Result with change of reference during try out

As seen the duration wasn't more then 20 seconds on each reference value, but the controller response all right nevertheless.

5.3.5 Change of reference backwards

This test is almost the same as the previous one, except that the reference changes in the other direction. The results can be seen in figure (5.8)



(a) Position and reference

(b) PID values contribution

Figure 5.8: Result with change of reference backwards during try out

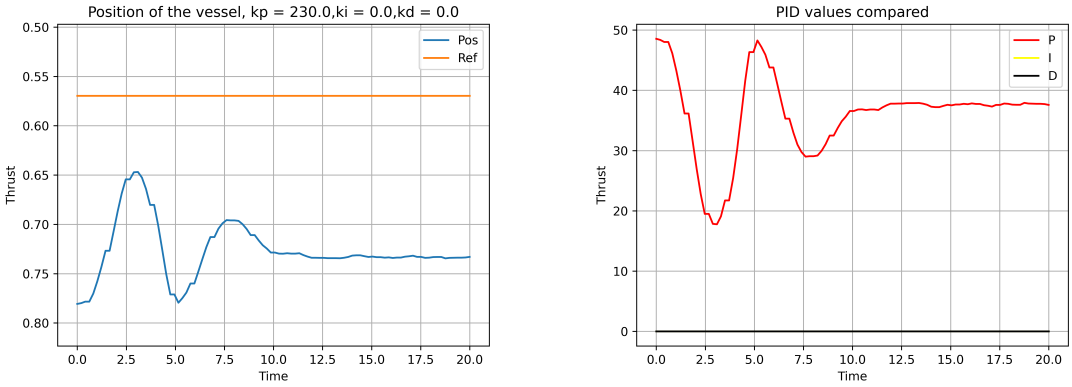
With closer examination it can be seen that the result is quite similar to how it is for previous test. It oscillates a bit when it changes, but finds the correct position rather quick.

5.4 Testing different setup of the controller

To get a better understanding and more experience with how PID-controller works and how to tune them, it was determined that a good idea was to test how the controller works with just the P-parameter and with just the PI-parameters.

5.4.1 Test of P-controller

To show that the system doesn't work with a single P-controller, a test for this was done. The result can be seen in figure (5.9). As seen the system stabilizes with a steady state error. The system also oscillates a bit to get there.



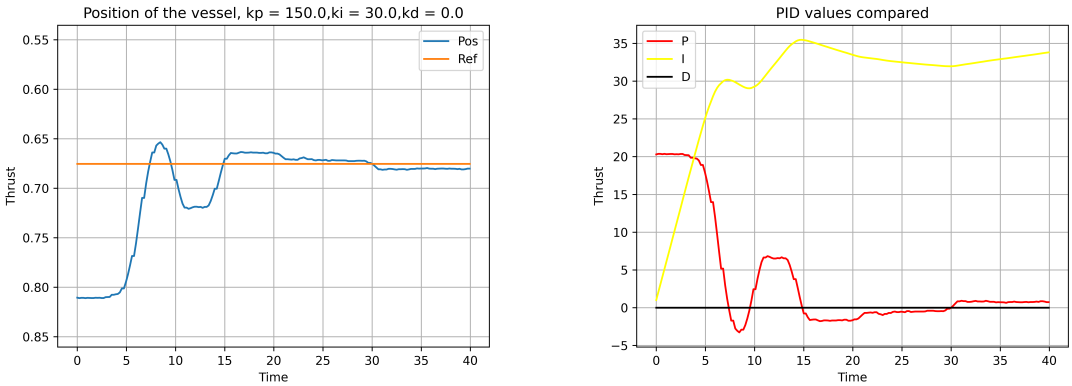
(a) Position and reference

(b) PID values contribution

Figure 5.9: Result for P-controller

5.4.2 Test of PI-controller

A PI-controller could have worked for the system, but is not ideal. See figure (5.10).



(a) Position and reference

(b) PID values contribution

Figure 5.10: Result for PI-controller

The system gets to the reference position after roughly 20 seconds, which is a lot compared to what the PID-controller manage. The system needed to be tuned carefully to avoid big oscillations, and that goes on the expense of the rise time.

5.5 Visualization

The task required a visualization of the vessel. The result was to make a VTF-file and visualize it in GLview. The result of two cases is added to the submission folder, where one is when the controller is tuned and the response is low. The other one is with more movements. To have a quick view on how the result was, a picture of the animation can be seen in figure (5.11). The red rectangle represent the boat, and the green rectangle represent the tank.

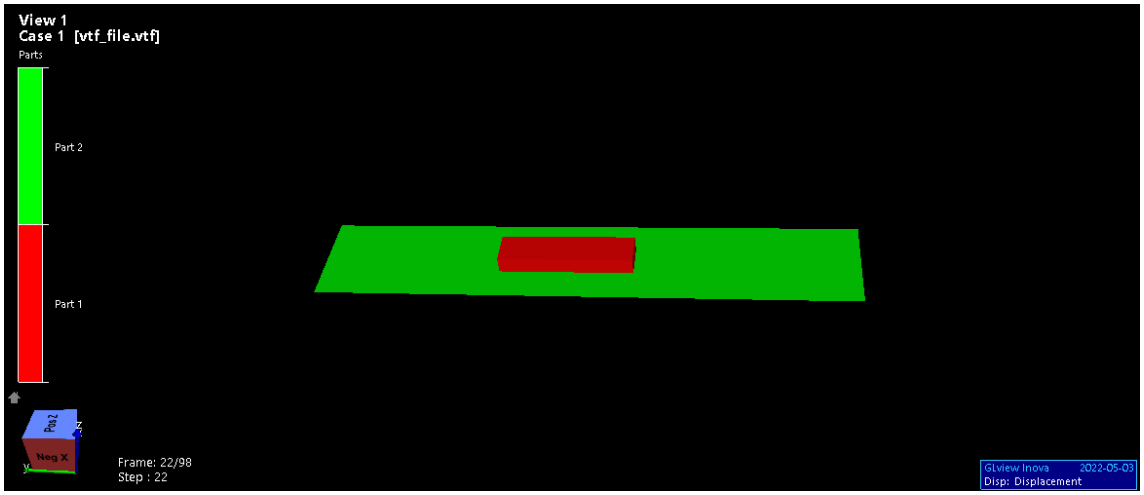


Figure 5.11: Picture of the animation in GLview

6 Discussion

6.1 Discussion of results

The result of Ziegler-Nicholas method was surprisingly good, which indicates that this empirical method is a good indication. The method is in principle an aggressive tuning method, but the manual tuning ends up being even more aggressive. Which means the DP-system tested in this project withstands instability very good. This is something making sense since the lab setup is inside and in a basin, where there are little disturbances. The learning of using this method is that it should have been used before manual tuning, which might have reduced the amount of time used on manual tuning.

The results for the manual tuning became very good in the end. Although the results could have been improved, since the vessel always had an overshoot of the wanted position. To reduce the overshoot, either K_d should be increased or K_p decreased, ref table (5.2). Since K_d already was quite high it seemed stupid to increase this more, because this could make the system unstable. However the results has proven that the system is pretty stable for $K_d = 140$. With more time, it would have be tested with how high K_d can be tune before it effects the stability of the system. The other alternative with reducing K_p, decreases the rise time, which is unwanted. So summed up the overshot of the controller could have been improved, but this might then gone at the expense of the stability or the rise time. Due to little time to do more testing and the result being quite good, it was determined to keep the PID-parameters as they was.

In all the cases it can be seen at the integral part dominates the output thrust when the system has become stable. This indicates that the code implemented is correct. There was at a time a problem with the integration part blowing up and becoming very big. This was dealt with by limiting the integration sum if it became to big or small. For a system working fine, this part shouldn't contribute anything, but was useful for a early tuning process.

6.2 Discussion of error sources

Overall the simple lab setup worked very well. However the setup was a bit unreliable. There were at several times issues with the potentiometer which made it impossible to use the lab, since the potentiometer gave weird outputs. There were many students working in the same lab. Sometimes small changes to the equipment was made, and that resulted in a need of re-calibration of the parameters. But the resulting parameters proved to respond good even if changes were made.

As mention earlier the system is simplified to a one dimensional system working in x-direction. However the model is not perfect and the boat tends to move sideways at times. These sideways movement are suppose to be cancel by the weights hanging in the ropes at each end and dragging the vessel to work in the same line, and that is actually what is happening too. Although these effects influences the position reading a bit, it's concluded with the effect being small and the assumption of a one dimensional system is good. During testing the rotational movements, yaw, roll and pitch, were small and didn't effect the system.

The basin in the lab setup is quite small. That resulted in waves being reflected back. It was seen as an effect that was all right since it gave a more realistic picture with unpredictable waves included. However this effects produces waves that are coming towards the propeller of the boat and if these waves were large enough the propeller got a ventilation problem. Ventilation is when the propeller is taking in air to the propeller blades and loses the grip on the water, which reduces the thrust and increases the rpm. This is something happening very fast and the propeller are quickly getting the grip back, but it effects the controller, and sometimes gives some new oscillations. At some point the basin was filled with more water which reduced this effect since the waves became smaller when there was more water in the tank.

6.3 Further work

The project was really enjoyable and the resulting code became good. However a fun addition to the project would have been to add an additional degree of freedom. This is maybe unrealistic to implement because the lab setup needs to be change, but a bigger tank and maybe a Lidar or cameras to read the position would have worked. The RC boat would need some kinds of side thrusters to make it possible as well.

The graphical user interface implemented is a very simplified version, so to improve the code, it would probably be wise to replace this with a library which would have given a GUI which looked better and had more possibilities. The animation made in GLview is very simple, so it would probably be smart to improve this as well.

7 Conclusion

The conclusion for the project is that the DP-system worked very well, and the program was easy to use with the graphical user interface. The lab setup was easy and clever, although the potentiometer has been a bit unreliable. It has been fun to see the physical output from your own code. A good effort has been made in coding, tuning and testing, and it has been both fun and instructive. The learning outcome was that Python has many good and easy to use libraries. Have also learn the limitations of python, with the code being slow, when live plotting inside the PID-controller. In similar projects in the future, C or C++ would preferably be used.

A big thanks to the teaching assistants, especially Kristoffer Sørfonn, and the subject manager Håvard Holm, without help and guidance the project hadn't been finished in a all right way. A special thanks to the once fixing the DP-lab, when there was issues.

Bibliography

[1] Phidget, *Phidgetinterfacekit 8/8/8*. [Online]. Available: https://www.phidgets.com/?tier=3&catid=2&pcid=1&prodid=1021.

[2] ——, *Phidgetadvancedservo 1-motor*. [Online]. Available: https://www.phidgets.com/?tier=3&catid=21&pcid=18&prodid=1044.

[3] *Potentiometer*. [Online]. Available: https://en.wikipedia.org/wiki/Potentiometer, (accessed: 04.05.2022).

[4] *Dynamic positioning*. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_positioning, (accessed: 05.05.2022).

[5] B. Dougles, *Understanding pid control, part 1: What is pid control?*, https://www.youtube.com/watch?v=wkfEZmsQqiA&ab_channel=MATLAB, May 2018.

[6] *Pid controller*. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller#cite_note-18, (accessed: 04.05.2022).

[7] H. Patel, *P vs pi vs pid controller graph*. [Online]. Available: https://instrumentationtools.com/what-is-pid-controller/, (accessed: 04.05.2022).

[8] *How to tune a pid controller?* [Online]. Available: https://www.omega.co.uk/technical-learning/tuning-a-pid-controller.html, (accessed: 04.05.2022).

[9] *Ziegler–nichols method*. [Online]. Available: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method, (accessed: 04.05.2022).

Appendix

A Code - main.py

```
from Phidget22.Phidget import *
from Phidget22.Devices.RCServo import *
from Phidget22.Devices.VoltageRatioInput import *

from PID import PID
from utilities import start_up, read_parameters_from_file, plot_PID,plot_pos,
    make_vtf_file


def main():

    run = True
    try:    # checks if the Phidget cards are connected
        rc = RCServo()
        ch = VoltageRatioInput()
        ch.setChannel(4)
        ch.openWaitForAttachment(5000)
        rc.openWaitForAttachment(5000)
    except:
        print("Not connected to the phidigets")
        run = False

    if run:
        print("main is running")

        # Reads the chosen parameters in
        parameters_list = read_parameters_from_file("textfiles/parameters.txt")
        kp = parameters_list[0]
        ki = parameters_list[1]
        kd = parameters_list[2]
        t = parameters_list[3]
        ref = parameters_list[4]

        # Runs the start up function which sets the boundaries conditions
        bound = start_up(rc, ch)
        # Calculate the reference to be correct with respect to the varying
    boundaries
        # This needed to be changed after the lab setup was fixed 06.05.2022
        # The potentiometer was changed to read higher voltage at the end then on
    the start
        # It was the opposite way before the change.
        r = bound[1] + (1-ref) * (bound[0] - bound[1])
        dt = 0.25 # integration step
        PID(kp, ki, kd, t, dt, r, rc, ch) # PID runs

        # All the files with the data needed to make plots
        files = ["textfiles/pos.txt","textfiles/ref.txt","textfiles/P.txt","
    textfiles/I.txt","textfiles/D.txt"]

        # Plot for position vs reference
        plot_pos(files[0],files[1],kp,ki,kd,bound)

        # Plot the PID values
        plot_PID(files[2],files[3],files[4])

        # Making a VTF file
        make_vtf_file(bound)

        # close the Phidget cards
        rc.close()
        ch.close()


if __name__ == "__main__":
    main()
```

B Code - PID.py

```
import matplotlib.pyplot as plt
import time
from datetime import datetime

from utilities import clear_file, write_to_file, get_pos, set_thrust, scale_u


# The PID controller
def PID(kp, ki, kd, timeout, dt, r, rc, ch):

    # sets the start deviation
    # This also needed to be changed, when the setup was changed, 06.05.2022
    e_prev = -r + get_pos(ch)  # e = r - y
    e_sum = 0
    e = e_prev

    # All the files i need to save data
    files = ["textfiles/pos.txt","textfiles/ref.txt","textfiles/P.txt","textfiles/I
    .txt","textfiles/D.txt"]

    # Clear all the files
    for file in files:
        clear_file(file)

    sleep = 0.20    # the sleep time of the while loop
    timeout_start = time.time()

    # The loop runs until the time reaches timeout
    while time.time() < timeout_start + timeout:
        # the current time from the loop started
        now_time = time.time()-timeout_start
        # Changes the reference at mid time. Used to test the systems strengh
        # if ((timeout/2) - sleep/2) < now_time < ((timeout/2) + sleep/2):
        #    r -= 0.04

        e_prev = e  # e_prev is set to be the error from the last timed the loop
        # gets a new error
        # This also needed to be changed, when the setup was changed, 06.05.2022
        e = -r + get_pos(ch)

        # Proprotional part
        P = kp*e

        # Integrator part, sums the errors
        e_sum = e_sum + e * dt
        I = ki * e_sum
        # Needs make sure the integration doesn't gets to big or low
        if I > 90:
            e_sum = 90/ki
        elif I < -90:
            e_sum = -90/(ki)


        # Derivative part
        dedt = (e - e_prev)/dt
        D = kd * dedt



        # Needs to add 90 because this is the angle where the thrust is zero
        u = P + I + D + 90
        # Scales the thurst to be within working range, else its zero
        new_u = scale_u(u)
        # Apply thrust
        set_thrust(rc,new_u)


        # Write the positon and reference to file
        write_to_file(files[0],now_time,get_pos(ch))
        write_to_file(files[1],now_time,r)
```

```
70
71
72          # write the PID values to file
73          write_to_file(files[2],now_time,P)
74          write_to_file(files[3],now_time,I)
75          write_to_file(files[4],now_time,D)
76
77
78          # Sleep the loop to make it run approximatly every 0.25 seconds,
79          # becuase it's when the voltage also is updated
80          time.sleep(sleep)
81
82      # Stops the thrust appleied when the loop is finished
83      rc.setEngaged(False)
```

C Code - utilities.py

```
1  from PIL import Image, ImageTk
2  import subprocess
3  import time
4  import matplotlib.pyplot as plt
5  from datetime import datetime
6
7  # Startup function, to get the maximum position and minimum position.
8  # Uses sleep to be sure it's on the right positions
9  def start_up(rc, ch):
10     bound = []
11     startPos = ch.getVoltageRatio()
12     bound.append(startPos)
13     rc.setTargetPosition(140)
14     rc.setEngaged(True)
15     time.sleep(5)
16     endPos = ch.getVoltageRatio()
17     time.sleep(1)
18     bound.append(endPos)
19     rc.setTargetPosition(100)
20     rc.setEngaged(True)
21     time.sleep(8)
22     rc.setEngaged(False)
23     print(f'Start position is {startPos}')
24     print(f'End position is {endPos}')
25     write_to_bound_file(bound)
26     return bound
27
28 # Reads out the position as voltage
29 def get_pos(ch):
30     return ch.getVoltageRatio()
31
32 # Sets the thrust of the vessel
33 def set_thrust(rc,thrust):
34     rc.setTargetPosition(thrust)
35     rc.setEngaged(True)
36
37 # Scales the thrust to operate within reasonable thurst areas
38 def scale_u(u):
39     new_u = 0
40     if u < 90:
41         new_u = 90
42     elif u > 145:
43         new_u = 145
44     else:
45         new_u = u
46     return new_u
47
48 # Here comes all the functions doing operations on text files:
49 # -------------------------------------------------------------
50
51 # Write the boundaries to file with correct syntax
52 def write_to_bound_file(liste):
53     temp = []
54     filename = "textfiles/bound.txt"
```

```
    f = open(filename, "w")
    temp.append(str(liste[0]) + "\n")
    temp.append(str(liste[1]))
    f.writelines(temp)
    f.close()

# Reads out the boundaries and return them as a list
def read_bound():
    file = open("textfiles/bound.txt","r").read()
    lines = file.split("\n")
    return [float(lines[0]),float(lines[1])]

# Clears a file
def clear_file(filename):
    file = open(filename, "w")
    file.write('')
    file.close()

# Reads out parameters from a given file and retun them as a list
def file_to_list(filename):
    file = open(filename,'r').read()
    lines = file.split('\n')
    xs, ys = [],[]
    for line in lines:
        if len(line) > 1:
            x, y = line.split(',')
            xs.append(float(x))
            ys.append(float(y))
    return xs,ys

# Appends two value to a given file
def write_to_file(filename,a,b):
    file = open(filename, "a")
    file.write(f'{str(a)},{str(b)}\n')
    file.close()

# Reads out the parameters from a file, given a filename
def read_parameters_from_file(filename):
    f = open(filename,'r').read()
    lines = f.split('\n')
    temp = []
    for elem in lines:
        temp.append(float(elem.split('=')[1]))
    return temp

# Overwrite the parameters of a given file
def write_parameters_to_file(filename,kp,ki,kd,time,ref):
    f = open(filename,'w')
    f.write(f"kp = {kp}\n")
    f.write(f"ki = {ki}\n")
    f.write(f"kp = {kd}\n")
    f.write(f"t = {time}\n")
    f.write(f"ref = {ref}")

# ------------------------------------------------------------

# a function which loads images in the GUI_plot function
def load_image(path, window, key):
    try:
        image = Image.open(path)
        image.thumbnail((600, 600))
        photo_img = ImageTk.PhotoImage(image)
        window[key].update(data=photo_img)
    except:
        print(f"Not possible to open file from {path}!")

# plot the position and refernce and save them
def plot_pos(file1,file2,kp,ki,kd,bound):
    fig, ax = plt.subplots()
    xs_pos,ys_pos = file_to_list(file1)
    xs_ref,ys_ref = file_to_list(file2)
    ax.plot(xs_pos, ys_pos, label = "Pos")
    ax.plot(xs_ref, ys_ref, label = "Ref")
```

```
    ax.set_ylim(bound[0] + 0.05 ,bound[1] - 0.05)
    ax.legend()
    ax.grid(True)
    ax.set_xlabel("Time")
    ax.set_ylabel("Thrust")
    ax.set_title(f"Position of the vessel, kp = {kp},ki = {ki},kd = {kd}")

    # Saves plot
    save_results_to = 'Results/Results_pos_ref/'
    plt.savefig(save_results_to + f'image{datetime.now().strftime("%d,%m %H:%M:%S")}.png',dpi=600)
    save_results_to = 'Results/GUI_plots/'
    plt.savefig(save_results_to + f'pos.png',dpi=600)

# plot the PID values and save them
def plot_PID(fileP,fileI,fileD):
    fig, ax = plt.subplots()
    colors = ['red','yellow','black']
    files = [fileP,fileI,fileD]
    labels = ["P","I","D"]
    for i in range(3):
        xs,ys = file_to_list(files[i])
        ax.plot(xs,ys,color = colors[i],label = labels[i])
    ax.legend()
    ax.grid(True)
    ax.set_xlabel("Time")
    ax.set_ylabel("Thrust")
    ax.set_title("PID values compared")

    # Saves plot
    save_results_to = 'Results/Results_PID/'
    plt.savefig(save_results_to + f'image{datetime.now().strftime("%d,%m %H:%M:%S")}.png',dpi=600)
    save_results_to = 'Results/GUI_plots/'
    plt.savefig(save_results_to + f'PID.png',dpi=600)

# write the vtf file
def make_vtf_file(bound):
    filename = "textfiles/pos.txt"
    x,y_pos = file_to_list(filename)

    filename2 = 'vtf_file.vtf'
    file = open(filename2,'w')

    # Parameters for vessel
    l = 8
    w = 2
    h = 1

    # hardcoe the vessel
    file.write("*VTF-1.00\n")
    file.write("\n")
    file.write("!   8 noder i en firkant :\n")
    file.write("*NODES              1\n")
    file.writelines([f"{int(-w/2)}. {int(-l/2)}. 0.\n",f"{int(-w/2)}.  {int(l/2)}. 0.\n",f" {int(w/2)}. {int(-l/2)}. 0.\n",f" {int(w/2)}.  {int(l/2)}. 0.\n",f"{int(-w/2)}. {int(-l/2)}. {h}.\n",f"{int(-w/2)}.  {int(l/2)}. {h}.\n",f" {int(w/2)}. {int(-l/2)}. {h}.\n",f" {int(w/2)}.  {int(l/2)}. {h}.\n"])
    file.write("\n")

    # Parameters for tank
    l_t = 32
    w_t = 6

    # hardcode the tank
    file.write("!   4 noder i et plan :\n")
    file.write("*NODES              2\n")
    file.writelines([f"{int(-w_t/2)} {int(-l_t/2)} 0\n",f"{int(w_t/2)} {int(-l_t/2)} 0\n",f"{int(-w_t/2)} {int(l_t/2)} 0\n",f"{int(w_t/2)} {int(l_t/2)} 0\n"])
    file.write("\n")


    file.write("!   8 noder i en firkant :\n")
```

```
    file.write("*ELEMENTS            1\n")
    file.write("%NODES #1\n")
    file.write("%QUADS\n")
    file.writelines(["1 2 4 3\n","5 6 8 7\n","1 2 6 5\n","3 4 8 7\n","1 3 7 5\n","2
     4 8 6\n"])
    file.write("\n")


    file.write("!   8 noder i en firkant :\n")
    file.write("*ELEMENTS            2\n")
    file.write("%NODES #2\n")
    file.write("%QUADS\n")
    file.write("1 2 4 3\n")
    file.write("\n")


    file.write("*GLVIEWGEOMETRY 1\n")
    file.write("%ELEMENTS\n")
    file.write("1 2\n")
    file.write("\n")

    # iterates to write all the position to the file
    scaled_y_pos = []
    for i in range(len(y_pos)):
        scaled_y = (y_pos[i]-bound[0])/(bound[1]-bound[0])
        n_scaled_y = -l_t/2 * (scaled_y - 0.5)
        scaled_y_pos.append(int(n_scaled_y))


    for i in range(0,len(y_pos)):
        file.write(f"*RESULTS   {i+1}\n")
        file.write("%DIMENSION 3\n")
        file.write("%PER_NODE #1\n")
        for j in range(0,8):
            file.write(f"0. {scaled_y_pos[i]}. 0\n")
        file.write("\n")


    file.write("*GLVIEWVECTOR   1\n")
    file.write('%NAME "Displacement"\n')
    for i in range(0,len(y_pos)):
        file.write(f"%STEP   {i+1}\n")
        file.write(f" {i+1}\n")


    file.close()
    print("Finished with writing to file")

# runs the run.sh script which runes both main and live_plot
def subprocess_call():
    subprocess.call(['sh','./run.sh'])
```

D Code - live_plot.py

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

from utilities import file_to_list,read_bound, read_parameters_from_file,
    clear_file

# makes a live animation by using the animation class to matplotlib
def plot_ani(file1,file2):
    fig, ax = plt.subplots()

    clear_file(file1) # clears the files
    clear_file(file2)

    # need to have the duration of the simulation to know when to close the window
    parameters_list = read_parameters_from_file("textfiles/parameters.txt")
    t = parameters_list[3]
```

```
    start_time = time.time()

    # the function which updates the animation with new points added
    def animate(i):
        if time.time() > start_time + t + 20:
            exit()
        x_pos,y_pos = file_to_list(file1)
        x_ref,y_ref = file_to_list(file2)
        bound = read_bound()

        ax.clear()
        ax.plot(x_pos, y_pos,label="pos")
        ax.plot(x_ref,y_ref,label="ref")
        ax.set_ylim(bound[0] + 0.05, bound[1] - 0.05)
        ax.set_title("Live plot of vessel vs reference")
        ax.set_xlabel("Time")
        ax.set_ylabel("Position")
        ax.grid(True)
        ax.legend()

    ani = animation.FuncAnimation(fig, animate, interval=1000)
    plt.show()



if __name__ == "__main__":
    file1 = "textfiles/pos.txt"
    file2 = "textfiles/ref.txt"
    plot_ani(file1,file2)
```

E Code - GUI.py

```
#!/usr/local/bin/python

import PySimpleGUI as sg
from PIL import Image, ImageTk

from main import main
from utilities import read_parameters_from_file, load_image,
    write_parameters_to_file, subprocess_call

#
    ----------------------------------------------------------------------------------------------------------------------------------


# GUI_plots pops up when the simulation is over, and load the saved plots

def GUI_plots():

    path1 = 'Results/GUI_plots/pos.png'
    path2 = 'Results/GUI_plots/PID.png'

    layout = [  [sg.Image(key = "image1"),sg.Image(key = "image2")],
                [sg.Button("Load plots",font=("Helvetica", 20)),sg.Button("Exit",
    font=("Helvetica", 20))]

                    ]
    window = sg.Window("Plot GUI",layout,size=(1250,520))

    while(True):
        event, values = window.read()

        if event == sg.WIN_CLOSED or event == "Exit":
            break
        if event == "Load plots":
            load_image(path1,window,"image1")
            load_image(path2,window,"image2")

    window.close()

#
```

```
    ------------------------------------------------------------------------------------------------------------------------


def GUI_startup():
    # The layout consists of the GUI window with text,buttons and input cells
    layout = [  [sg.Text("Welcome to dynamic positioning controll system",font=("Helvetica", 30))],
                [sg.Text("You have two choicses: run the code with tuned PID values or run ",font=("Helvetica", 20))],
                [sg.Text("the code with your own PID values",font=("Helvetica", 20))],
                [sg.Button("Tuned PID values",key="pd",font=("Helvetica", 20)), sg.Button("Your own PID values",key="yov",font=("Helvetica", 20))],

                [sg.Text("Kp:", key="kp", visible=False, font=("Helvetica", 20)),sg.InputText(key="input_kp",visible=False,font=("Helvetica", 20),size = (10,1))],
                [sg.Text(" Ki:", key="ki", visible=False, font=("Helvetica", 20)), sg.InputText(key="input_ki",visible=False,font=("Helvetica", 20),size = (10,1))],
                [sg.Text("Kd:", key="kd", visible=False, font=("Helvetica", 20)),sg.InputText(key="input_kd",visible=False,font=("Helvetica", 20),size = (10,1))],
                [sg.Text("Duration:", key="time", visible=False, font=("Helvetica", 20)),sg.InputText(key="input_time",visible=False,font=("Helvetica", 20),size = (10,1))],
                [sg.Text("Reference point:", key="ref", visible=False, font=("Helvetica", 20)),sg.InputText(key="input_ref",visible=False,font=("Helvetica", 20),size = (10,1))],
                [sg.Button('Confirm',key="confirm",visible=False,font=("Helvetica", 20)), sg.Text("Try again!",key="error",visible=False,font=("Helvetica", 20))],
                [sg.Button('Run',key="run",visible=False,font=("Helvetica", 20)),sg.Button('Show live plots',key="live_plot",visible=False,font=("Helvetica", 20))],


                [sg.Button('Exit',key="exit",font=("Helvetica", 20))]
                    ]

    # Create the Window
    window = sg.Window('DP GUI', layout,size=(700,450))

    # Event Loop to process "events" and get the "values" of the inputs
    while True:
        # Continuesly reading the window
        event, values = window.read()
        # If user closes window or clicks cancel
        if event == sg.WIN_CLOSED or event == "exit":
            break
        # If the user presses predetermined values
        if event == "pd":
            print("You choose predetermind values")
            parameters = read_parameters_from_file("textfiles/tuningparameters.txt")
            kp = parameters[0] # values are read from tuningparameters
            ki = parameters[1]
            kd = parameters[2]
            t = parameters[3]
            ref = parameters[4]
            window["run"].Update(visible = True) # the run buttom becomes visible
        # If the user presses your own value
        if event == "yov":
            print("You want to choose your own values")
            window["pd"].Update(visible = False)
            window["yov"].Update(visible = False)

            state = True # makes all the input cells visible
            window["kp"].Update(visible = state)
            window["input_kp"].Update(visible = state)
            window["ki"].Update(visible = state)
            window["input_ki"].Update(visible = state)
            window["kd"].Update(visible = state)
            window["input_kd"].Update(visible = state)
            window["time"].Update(visible = state)
            window["input_time"].Update(visible = state)
```

```
            window["ref"].Update(visible = state)
            window["input_ref"].Update(visible = state)
            window["confirm"].Update(visible = state)
        # Have to press confirm to save the input values
        if event  == "confirm":
            kp = (values['input_kp'])
            ki = (values['input_ki'])
            kd = (values['input_kd'])
            t = (values['input_time'])
            ref = (values['input_ref'])
            # Needs to check that the input values are valid
            if len(kp)<1 or len(ki)<1 or len(kd)<1 or len(t)<1 or len(ref)<1:
                print("Error")
                window["error"].Update(visible = True)
            else:
                window["error"].Update(visible = False)
                kp = float(kp)
                ki = float(ki)
                kd = float(kd)
                t = float(t)
                ref = float(ref)
                window["run"].Update(visible = state)

        if event == "run":
            # Need to save the parameters, to make the aviable for main
            file = "textfiles/parameters.txt"
            write_parameters_to_file(file,kp,ki,kd,t,ref)
            # Main and live_plot runs
            subprocess_call()
            # Launch a windows which gives the possiblity to see the results
            GUI_plots()

    window.close()

#
    ------------------------------------------------------------------------------------------------------------------------


if __name__ == "__main__":
    GUI_startup()
```

F Code - run.sh

```
python3 live_plot.py & python3 main.py
```

´