
Coastal Profiling Float Depth Control

M B A R I



Author : **Brian Ha**
Mentor : **Gene Massion**
Last Revised : **2018-09-26**
MBARI Team : **Chemical Sensor Group**

Table of Contents

Introduction	1
Acronyms	3
Constants	3
Variables	3
Section 1: The CPF Model	4
Non-Linear ODEs	4
Linearization.....	5
State Space Model	6
Transfer Function Model	6
Section 2: The Controller	7
Performance Requirements.....	7
Dual Lead Compensator Expected Performance & Limitations.....	7
LTI Continuous-Time Simulation	9
NLTI Continuous-Time Simulation	10
NLTI Discrete-Time Analysis.....	11
Discretization	11
Command Unit Step Response	11
Stability Margins	12
Bode Plots of $S(z)$ and $T(z)$	13
Section 3: The Reference Governor	14
Motivation & Simulation.....	14
C++ Implementation	16
Suggested Future Work	17
Add Integral Control.....	17
Design and Simulate a Pressure Sensor Noise Filter.....	17
Conclusion.....	18
Acknowledgements.....	18
References	19
Appendix	20
A.1 MATLAB Script	20
A.2 Simulink Model	20
A.3 C++ Implementation	20

Introduction

Science-based marine conservation policies require an understanding of primary production rates in the ocean [1]. Although coastal areas represent only 5-10% of global surface area, they are estimated to contribute as much as 10-30% of global primary production [2]. Currently, no observation system exists with the resolution, frequency, and cost-effectiveness required to understand coastal ecosystem production processes. The Chemical Sensors Group at the Monterey Bay Aquarium Research Institute (MBARI) in Moss Landing, CA has been developing a solution: The Coastal Profiling Float (CPF).

The CPF is an oceanographic instrument platform optimized for making subsurface measurements in coastal areas (see Figure 1). The current prototype (#3) measures water column conductivity, temperature, depth, pH, nitrate, oxygen, fluorescence, backscatter, and optical radiation levels. Figure 2 depicts a typical mission profile. There are five phases in each profile: descent, drift, park (or anchor), ascent, and surface data transmission. The CPF ascends by pumping oil from an inner bladder (within its pressure housing) into two external bladders (exposed to the water) on either side. As the external bladders expand, they displace more water. As a result, the buoyant force increases and the CPF rises. Conversely, the CPF descends by pumping oil back into its inner bladder. The CPFs are designed with a battery life of 2-3 years and MBARI envisions each one completing (on average) six profiles per day.

There are two challenges unique to CPFs that must be addressed. First, surface currents may push the float onto shore. Second, the CPF will operate in relatively shallow areas (up to 500 meters in depth) where there is a risk of unintentionally hitting the seafloor. Both challenges motivate the need for depth control. To counteract surface drift, the CPF must descend to and maintain a depth where seaward currents dominate. To avoid striking the seafloor, the depth control system must have minimal overshoot. On top of these challenges, the limited battery life of the CPF necessitates the use of a low-energy pump with a relatively small displacement and slow top pumping speed. Consequently, the depth control system must also be energy efficient and respect pump speed constraints.

The author designed and simulated a depth control system for the CPF as part of his 2018 summer internship with MBARI. What follows is an explanation of the design process and the theoretical results. Section 1 presents the mathematical model of the CPF. Section 2 details the design of a discrete time, dual lead compensator. Lastly, Section 3 describes why and how a reference governor was augmented to the control system. It is the author's hope that this work will help MBARI move the CPF project from the development and prototyping phase into science data acquisition and nominal operations.

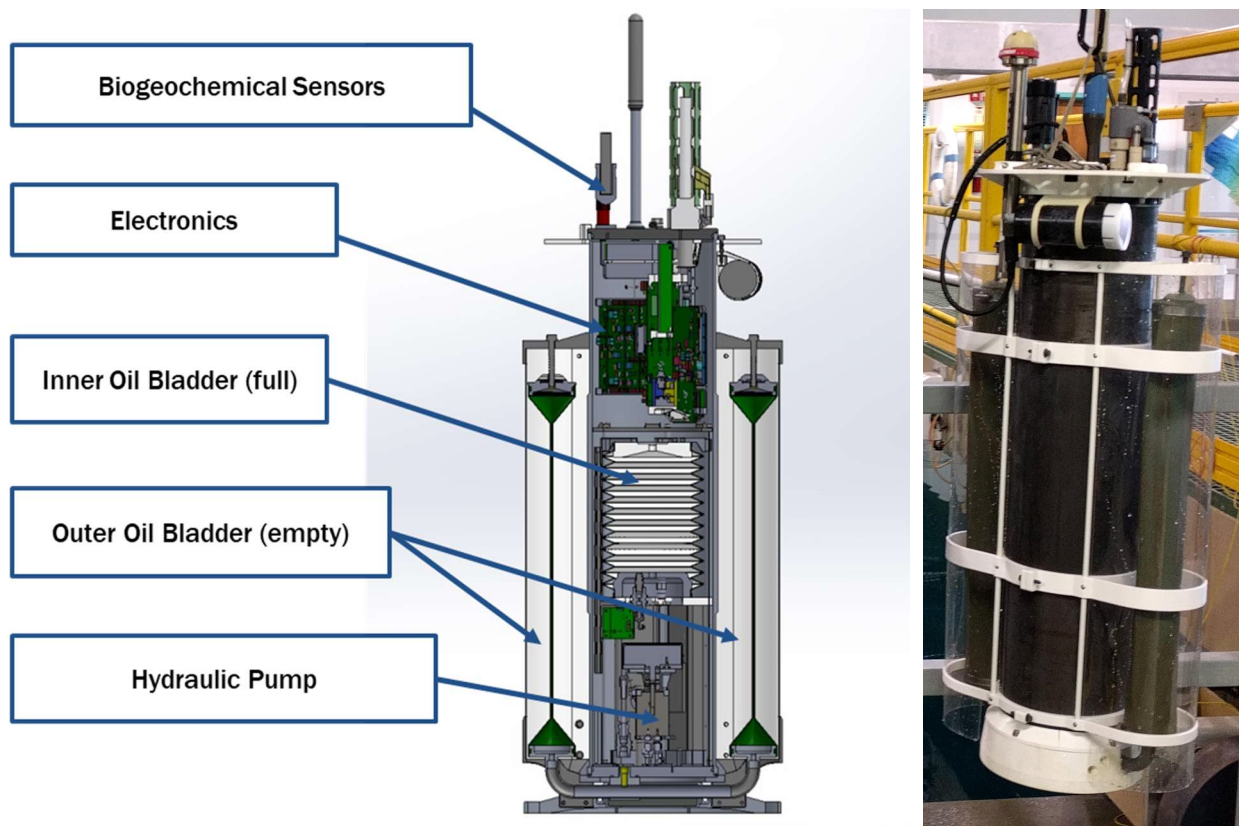


Figure 1: The Coastal Profiling Float (Prototype 3)

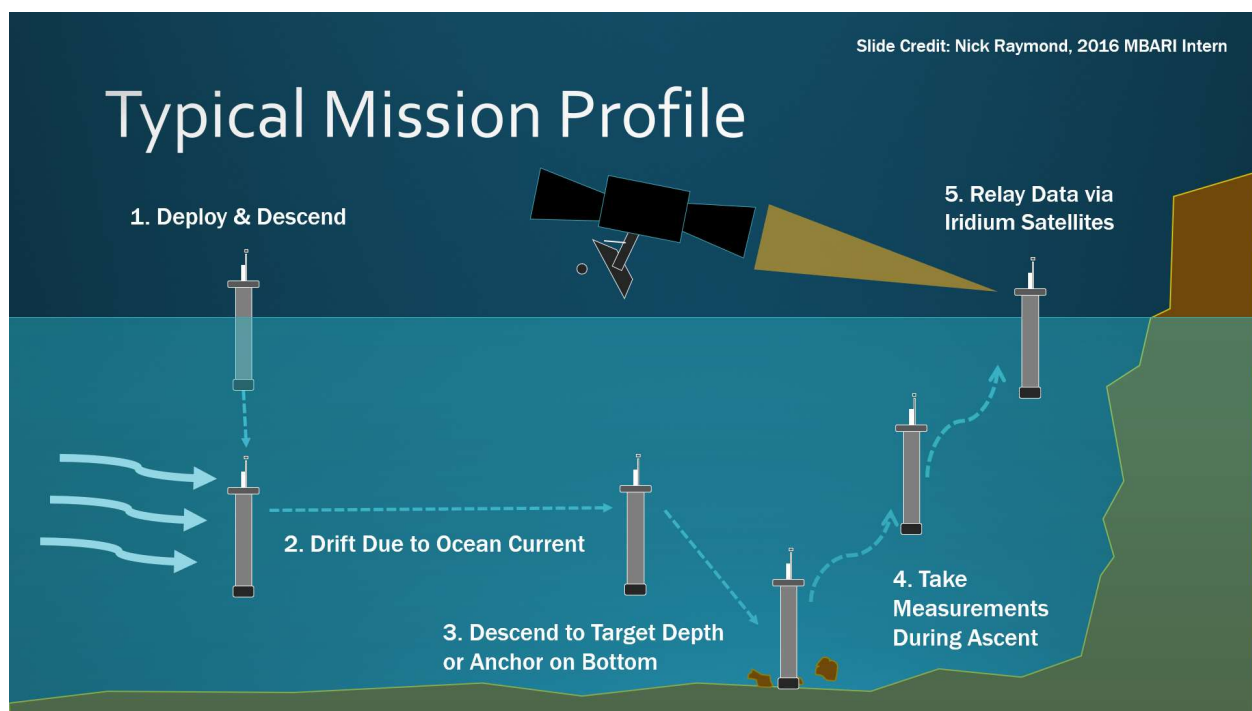


Figure 2: A Typical Mission Profile

Acronyms

CPF	Coastal Profiling Float	NLTI	Non-Linear Time-Invariant
ODE	Ordinary Differential Equation	SISO	Single Input, Single Output
LTI	Linear Time-Invariant	C#	C-Sharp (programming language)

Constants

Term	Definition	Value	Units
ρ	Density of Sea Water	1025	kg/m ³
g	Gravitational Acceleration	9.807	m/s ²
A	Frontal Area	0.0324	m ²
C_D	Drag Coefficient (3-dimesional)	0.9	-
m	Mass = CPF Mass + Virtual Mass	47.5	kg
ε	Estimated Pump Efficiency	0.6	-
δ	Pump Displacement	1.56e-7	m ³ /rev

Variables

Term	Definition	Units
F_B	Buoyant Force	N
F_W	Weight Force	N
F_D	Drag Force	N
ΔV	Change in Volume of the External Bladders	m ³
d	Depth	m
$\dot{d}$	Velocity	m/s
$\ddot{d}$	Acceleration	m/s ²
ω	Pump Speed	rev/s
$x(t)$	State Vector	-
$u(t)$	Plant Input (a.k.a. the control signal)	rev/s
$y(t)$	Plant Output	m
A	State Matrix	-
B	Input Matrix	-
C	Output Matrix	-
D	Feedforward Matrix	-
s	Laplace Operator	-
I_n	n-dimensional Identity Matrix	-
$P(s)$	Continuous-Time Plant Transfer Function	-
$C(s)$	Continuous-Time Controller Transfer Function	-
$C(z)$	Discrete-Time Controller Transfer Function	-
$d_i(t)$	Plant Input Disturbance	rev/s
$d_o(t)$	Plant Output (Depth) Disturbance	m
$n(t)$	Sensor Noise	-

Section 1: The CPF Model

Non-Linear ODEs

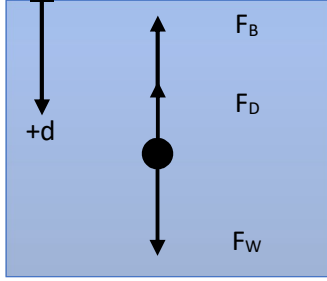


Figure 3: CPF Free Body Diagram

The CPF was modelled as a point mass with purely vertical motion. Descending motion was given a positive sign because pressure increases with depth. 1 decibar of pressure was assumed to equal 1 m of depth. Three forces were modelled: buoyancy (F_B), weight (F_W), and drag (F_D).

$$\Sigma F = -F_B + F_W - F_D \quad (1)$$

The buoyant force can be thought of as being comprised of a constant term and a variable term.

$$F_B = F_{B_{\text{constant}}} + F_{B_{\text{variable}}} \quad (2)$$

By assuming the constant term is equal to the weight of the CPF, the force model simplifies to:

$$\Sigma F = -F_{B_{\text{variable}}} - F_D \quad (3)$$

where

$$F_{B_{\text{variable}}} = \rho g \Delta V \quad (4)$$

$$F_D = \frac{1}{2} \rho A C_D \dot{d}^2 \quad (5)$$

ΔV denotes the change in volume of the CPF's external bladders away from the volume necessary for neutral buoyancy. In other words, if $\Delta V = 0$, the CPF is neutrally buoyant. If $\Delta V > 0$, the CPF will ascend and if $\Delta V < 0$, the CPF will descend. Let d denote depth, $\dot{d}$ denote velocity, and $\ddot{d}$ denote acceleration. Then, per Newton's 2nd Law ($\Sigma F = m\ddot{d}$):

$$-\rho g \Delta V - \frac{1}{2} \rho A C_D \dot{d}^2 = m \ddot{d} \quad (6)$$

The mass includes an additional virtual mass factor of 25%. Solving equation (6) for $\ddot{d}$ yields:

$$\ddot{d} = \frac{1}{m} \left(-\frac{1}{2} \rho A C_D \dot{d}^2 - \rho g \Delta V \right) \quad (7)$$

The rate of change of the volume of the CPF's external bladders (denoted as $\Delta \dot{V}$) is given by:

$$\Delta \dot{V} = \epsilon \delta \omega \quad (8)$$

where ϵ denotes estimated pump efficiency, δ is pump displacement, and ω is the pump speed.

Together, the non-linear, ordinary differential equation (ODE) (7) and the linear ODE (8) relate the plant input (pump speed) to the plant output (a change in depth).

Linearization

To take advantage of linear control design and analysis strategies, the model must be linearized. Define the state vector $x(t)$ and the input $u(t)$ as:

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} d(t) \\ \dot{d}(t) \\ \Delta V(t) \end{bmatrix} \quad u(t) = \omega(t) \quad (9) - (10)$$

The CPF model was linearized about an equilibrium point (x_{eq}, u_{eq}) at which equations (7) and (8) equal zero. This occurs at any depth when the CPF is motionless, neutrally buoyant, and with zero pump speed. Restated equivalently, this occurs when $\dot{d} = \Delta V = \omega = 0$. Without loss of generality, the equilibrium depth d_{eq} was simply chosen as zero. Define deviations about the equilibrium point as:

$$\delta x = x - x_{eq} \quad (11)$$

$$\delta u = u - u_{eq} \quad (12)$$

By performing a Taylor Series expansion about (x_{eq}, u_{eq}) and discarding the non-linear terms (greater than first order), the following LTI state space model is obtained:

$$\delta \dot{x}(t) = A \delta x(t) + B \delta u(t) \quad (13)$$

where A and B are Jacobian matrices:

$$A = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} \quad B = \begin{bmatrix} \frac{\partial f_1}{\partial u} \\ \frac{\partial f_2}{\partial u} \\ \frac{\partial f_3}{\partial u} \end{bmatrix} \quad (14) - (15)$$

where $f_1 = \dot{d}$, $f_2 = \text{Equation (7)}$, and $f_3 = \text{Equation (8)}$.

This linearization was performed in the `cpf_depth_control.m` MATLAB file (see Appendix) with the author-defined “`symLin`” function and verified by a hand derivation.

Going forward, the delta (δ) notation is dropped for simplicity, but the reader is asked to remember that any state or input value is relative to $x_{eq} = [d_{eq} \ 0 \ 0]^T$ and $u_{eq} = 0$.

State Space Model

The linearization yielded the following LTI SISO state space model:

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (16)$$

$$y(t) = Cx(t) + Du(t) \quad (17)$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -211.6 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 9.36 * 10^{-8} \end{bmatrix}, \quad C = [1 \quad 0 \quad 0], \quad D = 0 \quad (18) - (21)$$

The A matrix is a triple integrator and is thus unstable. This makes sense because linearizing the CPF about a motionless equilibrium point negates the stabilizing effect of drag. In other words, this linearized state space model is unstable because drag is not modelled. This is important to note because a real-world test or Simulink simulation of a stabilizing controller with non-linear drag included will have less overshoot than that predicted by this model.

The standard Kalman Rank Condition tests were used to determine that the model is both controllable and observable.

Transfer Function Model

The state space model can be converted into the following plant transfer function $P(s)$:

$$P(s) = C(sI_3 - A)^{-1}B + D = \frac{-1.9808 * 10^{-5}}{s^3} \quad (22)$$

where s denotes the Laplace operator and I_3 is a 3x3 identity matrix. This conversion was implemented in the `cpf_depth_control.m` MATLAB file (see Appendix) via the “`zpk`” (zero-pole-gain) command.

Section 2: The Controller

Performance Requirements

There were three performance requirements for this project:

1. No more than 25% overshoot for a 1 m step command.
2. 150 second max settling time to stay within 5% of a 1 m step command.
3. Per the oil pump manufacturer, pump speed magnitude must never exceed 50 rev/s.

Dual Lead Compensator Expected Performance & Limitations

The author was asked to develop a dual lead compensator based on the recommendation of Prof. Stephen Rock at Stanford University. Lead compensators have the benefit of being relatively easy to implement in code, but they may not be robust against model parameter uncertainty due to their lack of integral control. Additionally, they do not explicitly factor in constraints. A dual lead compensator has the following format:

$$C(s) = \frac{k(s + z_1)(s + z_2)}{(s + p_1)(s + p_2)} \quad (23)$$

where k denotes the gain, z_i denotes a zero, p_i denotes a pole, and $z_i < p_i$.

Figure 4 shows the architecture of the feedback system. This deterministic analysis accounted for the presence of plant input disturbances $d_i(t)$ and plant output disturbances $d_o(t)$, but not sensor noise $n(t)$.

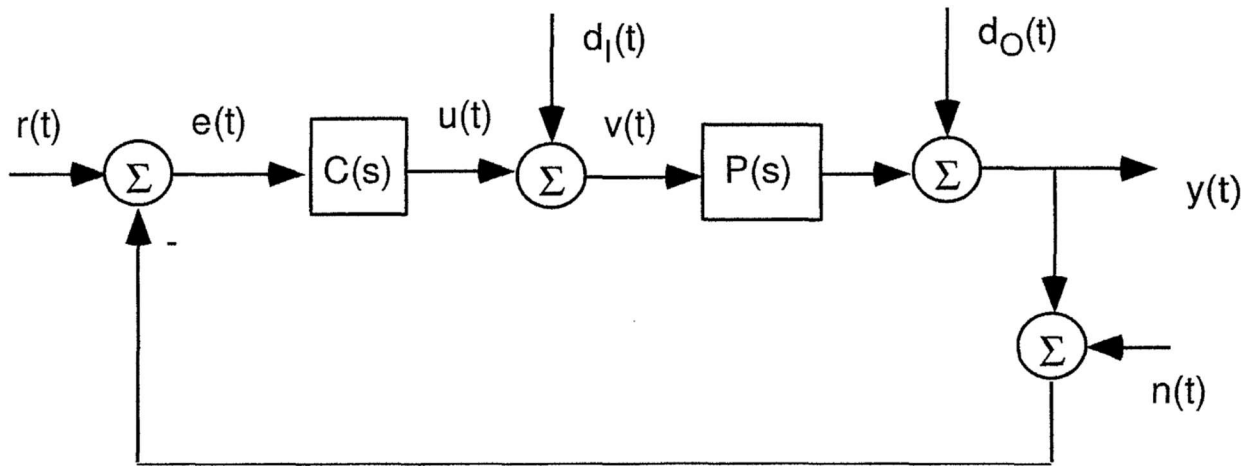


Figure 4: Feedback System Block Diagram. Image Source: [3]

Define the open loop transfer function as:

$$L(s) = P(s)C(s) \quad (24)$$

When dealing with a SISO LTI system, there are four closed loop transfer functions that must be stable.

1. The Plant Output Disturbance Sensitivity Function, $S(s)$

$$S(s) = \frac{1}{1 + L(s)} \quad (25)$$

This describes the effect of a depth disturbance (e.g. obstacles, internal waves), denoted in Figure 4 as $d_o(t)$, on the depth of the CPF. The controller should completely reject these.

2. The Command and Noise Complementary Sensitivity Function, $T(s)$

$$T(s) = \frac{L(s)}{1 + L(s)} = 1 - S(s) \quad (26)$$

This describes how the CPF's depth responds to a reference command. However, this also describes how the CPF's depth responds to pressure sensor noise. Note that good reference command tracking performance implies vulnerability to noise.

3. The Control Signal Sensitivity Function, $CS(s)$

$$CS(s) = C(s) * S(s) \quad (27)$$

This describes how the pump speed signal, $u(t)$, responds to reference commands, output disturbances, and noise. The magnitude of its output must never exceed 50 revs/s.

4. The Plant Input Disturbance Sensitivity Function, $SP(s)$

$$SP(s) = S(s) * P(s) \quad (28)$$

This describes the effect of an input disturbance (e.g. electrical noise), denoted in Figure 4 as $d_i(t)$, on the depth of the CPF. The CPF is unlikely to experience this type of disturbance. Thus, this transfer function need only be stable (asymptotic stability is not required).

Since $L(s)$ contains three integrators, it is a "Type 3" system. Some key insights can now be made about the theoretical performance and limitations of this feedback system.

1. The step response of $T(s)$ will have zero steady state error.

See [3], Theorem 1.9 on page 34 for proof. Assumes the system remains close to linear.

2. The step response of $T(s)$ must necessarily exhibit overshoot.

See [3], Theorem 6.3 (a) on page 188-189 for proof.

3. The step response of $S(s)$ will have a steady state value of zero (desirable).

See [3], Corollary 1.10 on page 35. Assumes the system remains close to linear.

4. The step response of $SP(s)$ will have a non-zero steady state value (undesirable).

See [3], Theorem 1.11 on page 35 for proof. This can be fixed by adding an integrator to $C(s)$.

5. **The magnitude of $S(s)$ must necessarily be greater than 1 over some frequency interval.**

See [3], Theorem 6.15 on page 206. This means that depth disturbances over some frequency interval will be amplified. The peak of $S(s)$ must be kept acceptably small. This also implies that the Nyquist plot must necessarily penetrate the unit circle centered at the critical point $(-1,0)$.

With the performance expectations and limitations in mind, the following controller was designed:

$$C(s) = \frac{-45(s + 0.015)(s + 0.001)}{(s + 0.17)(s + 0.15)} \quad (29)$$

LTI Continuous-Time Simulation

Figure 5 shows the unit step response of each of these transfer functions. The top left plot shows that the 150 second settling time specification is just barely violated, but the overshoot is unacceptable (40%). The reader is reminded that overshoot will be reduced once drag is included in the simulation. The top right plot confirms that the pump speed never exceeds ± 50 rev/s. The bottom left plot demonstrates the successful rejection of a 1 m depth disturbance. The bottom right plot shows that a 1 rev/s bias in pump speed is NOT rejected and can seriously affect depth in the long term. However, this disturbance is unlikely to occur and was deemed not concerning. The main takeaways here are that all four transfer functions are indeed stable and performance expectations #1-4 were observed.

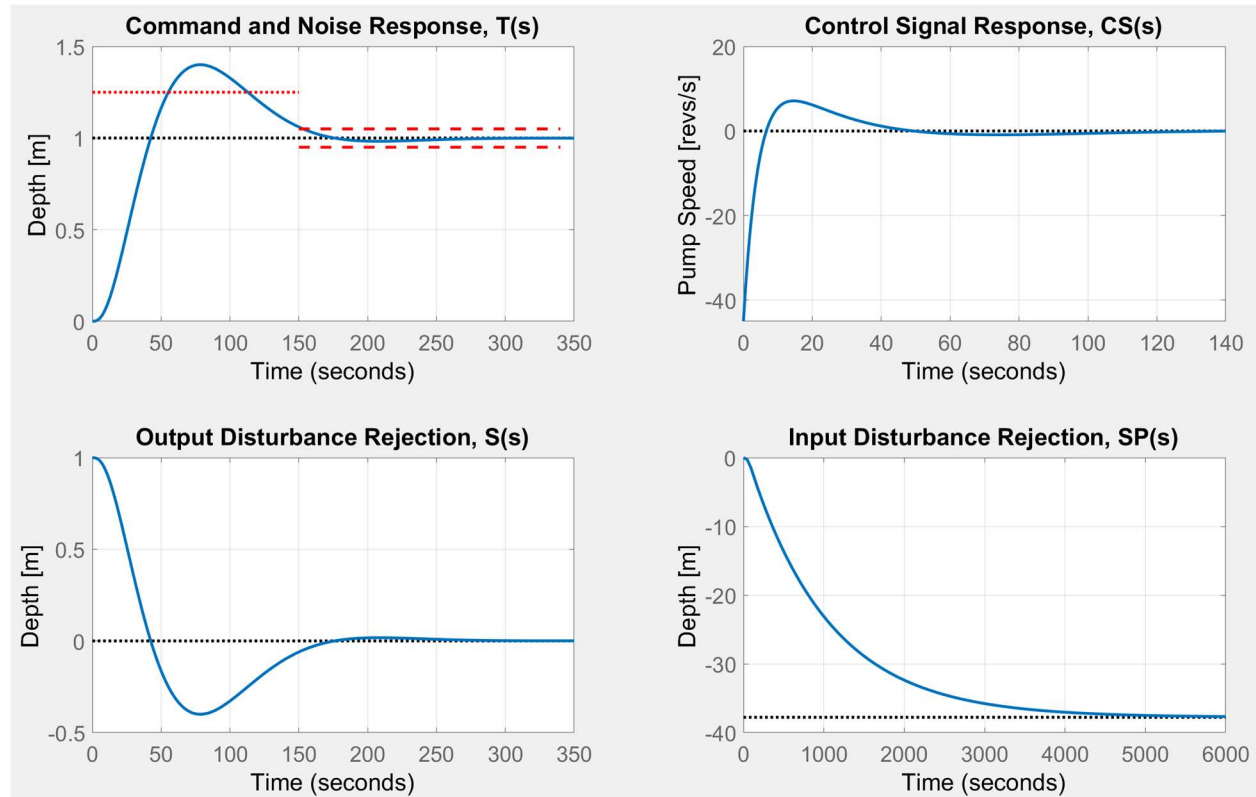


Figure 5: Unit Step Responses of the Four Closed Loop Transfer Functions (drag not simulated)

NLTI Continuous-Time Simulation

A Simulink model (cpf_model_continuous_controller.slx) was created to simulate system performance with non-linear (quadratic) drag. The block diagram can be found in the Appendix. A comparison of the command unit step responses and pump speeds is provided below in Figure 6. Observe, the overshoot and settling time requirements are now satisfied while the pump speed commands are nearly identical.

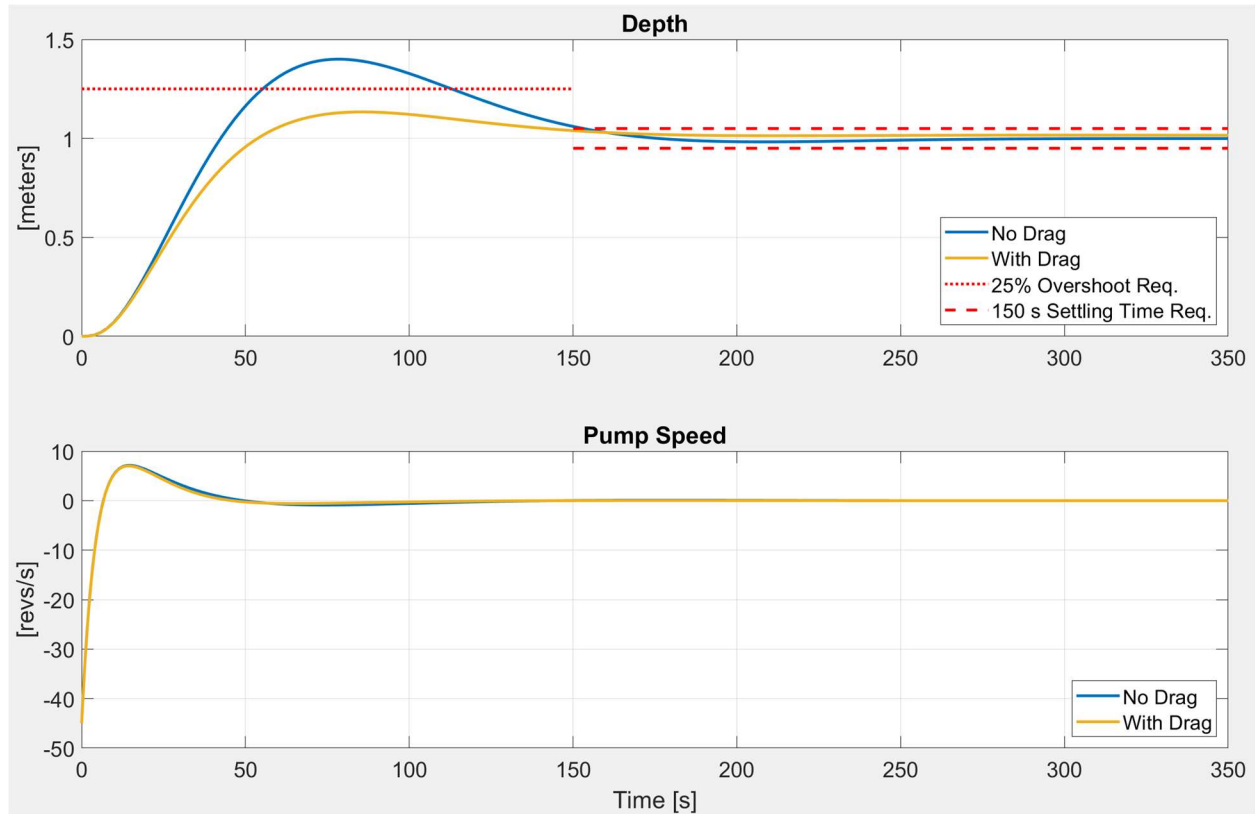


Figure 6: Comparison of Responses With and Without Drag Simulated

NLTI Discrete-Time Analysis

Discretization

The CPF measures water pressure once every 3 seconds. Due to this fact, the performance and stability of a discretized version of the controller needed to be verified. The discretization was done via the MATLAB c2d command with the “matched” method. The discrete controller transfer function was:

$$C(z) = \frac{U(z)}{E(z)} = \frac{-29.07 z^2 + 56.78 z - 27.71}{z^2 - 1.238 z + 0.3829} \quad (30)$$

Equation (30) was converted into the following difference equation for implementation in C#.

$$u_k = -29.07e_k + 56.78e_{k-1} - 27.71e_{k-2} + 1.238u_{k-1} - 0.3829u_{k-2} \quad (31)$$

where:

- u_k is the next pump speed command
- u_{k-1} is the previous pump speed command (initially 0)
- u_{k-2} is the previous, previous pump speed command (initially 0)
- e_k is the current depth error
- e_{k-1} is the previous depth error (initially 0)
- e_{k-2} is the previous, previous depth error (initially 0)

Command Unit Step Response

A Simulink model (cpf_model_with_discrete_controller.slx) was created to simulate system performance with non-linear (quadratic) drag. The block diagram of the difference equation can be found in the Appendix. A comparison of the command unit step responses and pump speeds is provided below in Figure 7. Observe, the continuous and discrete cases are nearly identical.

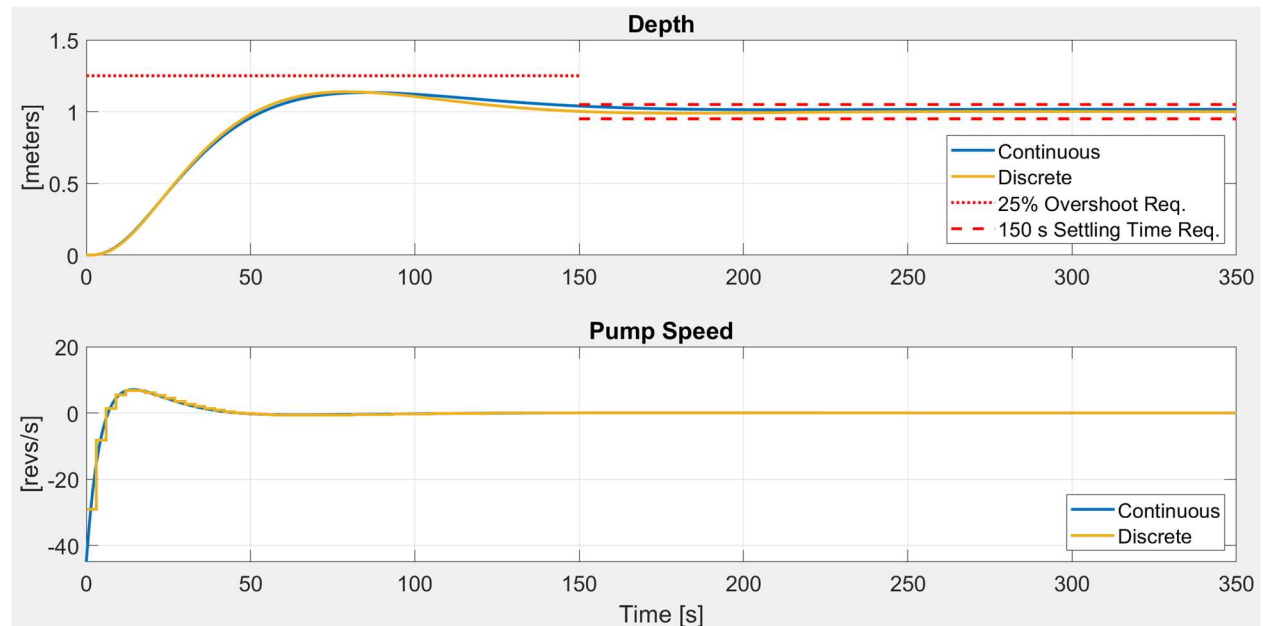


Figure 7: Continuous vs. Discrete Controller Responses

Stability Margins

Figure 8 is a Bode plot of the discrete open loop transfer function $L(z)$. Note that the system has a gain margin of 13.7 dB at 0.113 rad/s and a phase margin of 36.8 deg at 0.036 rad/s.

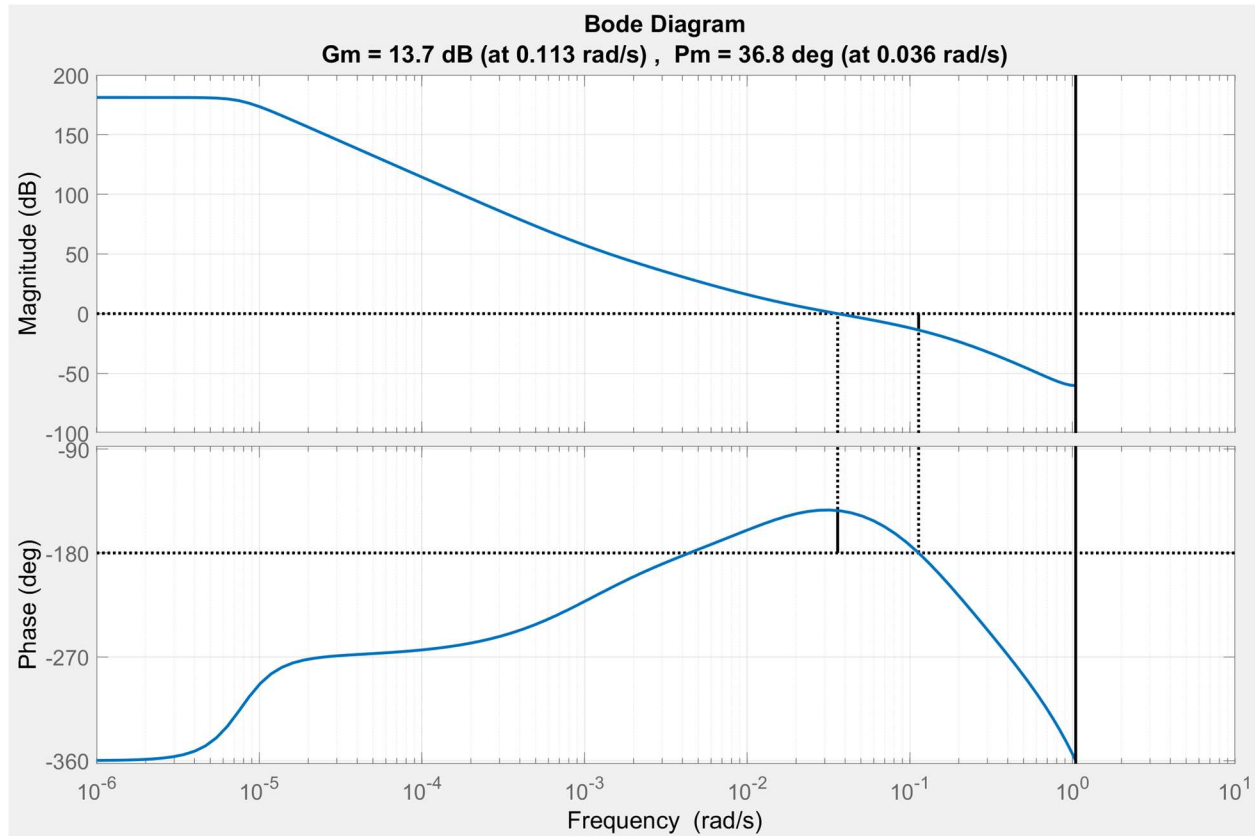


Figure 8: Bode Plot of the Discrete Open Loop Transfer Function, $L(z)$

On top of indicating closed loop stability, the Nyquist plot in Figure 9 can also be used to determine the peak of $S(z)$. The stability radius, or the distance from the critical point $(-1,0)$ to the closest point on the Nyquist plot, was 0.568. The peak magnitude of $S(z)$ is determined by the inverse: $1/0.568 = 1.76$. As predicted, the Nyquist plot penetrates the unit circle centered at $(-1,0)$.

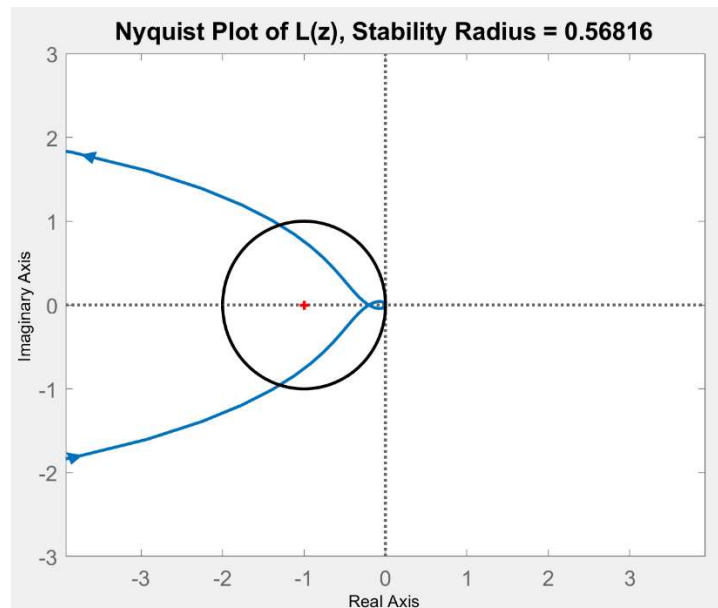


Figure 9: Nyquist Plot of $L(z)$

Bode Plots of $S(z)$ and $T(z)$

Figure 10 is a Bode plot of both discrete depth disturbance sensitivity $S(z)$ and command sensitivity $T(z)$. $S(z)$ amplifies depth disturbances between approx. 0.004 Hz and 0.05 Hz. This matches performance expectation #5. Its peak is 1.76 (4.91 dB) at 0.008 Hz, which corroborates the Nyquist plot. $T(z)$ has a peak of 1.65 (4.34 dB) at 0.004 Hz. The peak of $S(z)$ and $T(z)$ are both relatively small, indicating robustness against resonant frequencies.

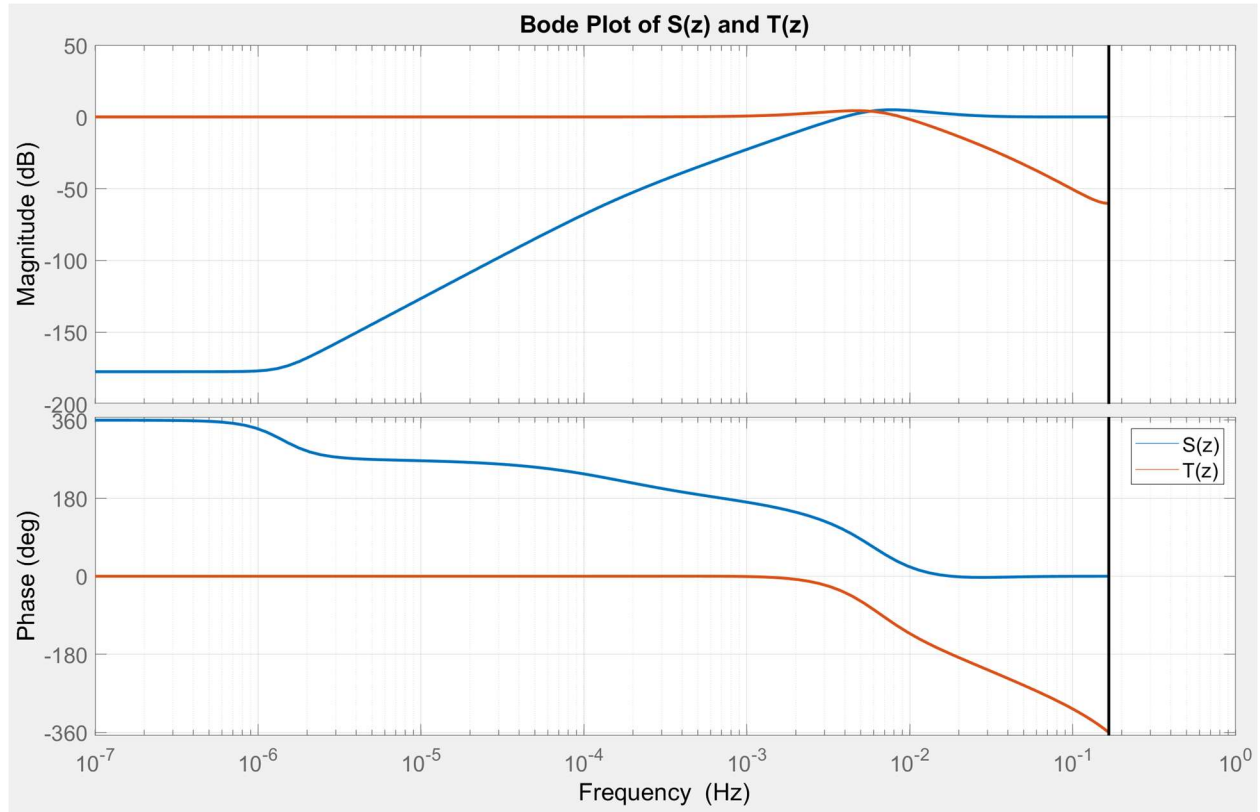


Figure 10: Bode Plots of $S(z)$ and $T(z)$

Section 3: The Reference Governor

Motivation & Simulation

Although its external bladders can displace an ample amount of water for depth control, the CPF is heavily constrained by the oil pump's speed limitation (± 50 revs/s). To illustrate this point, Figure 11 compares 3-meter depth change simulations with and without the pump speed constraint included in the Simulink model. Observe, not only does control saturation result in failure to meet performance requirements, the CPF may even start heading in the opposite direction! This is unacceptable and motivates the use of a reference governor.

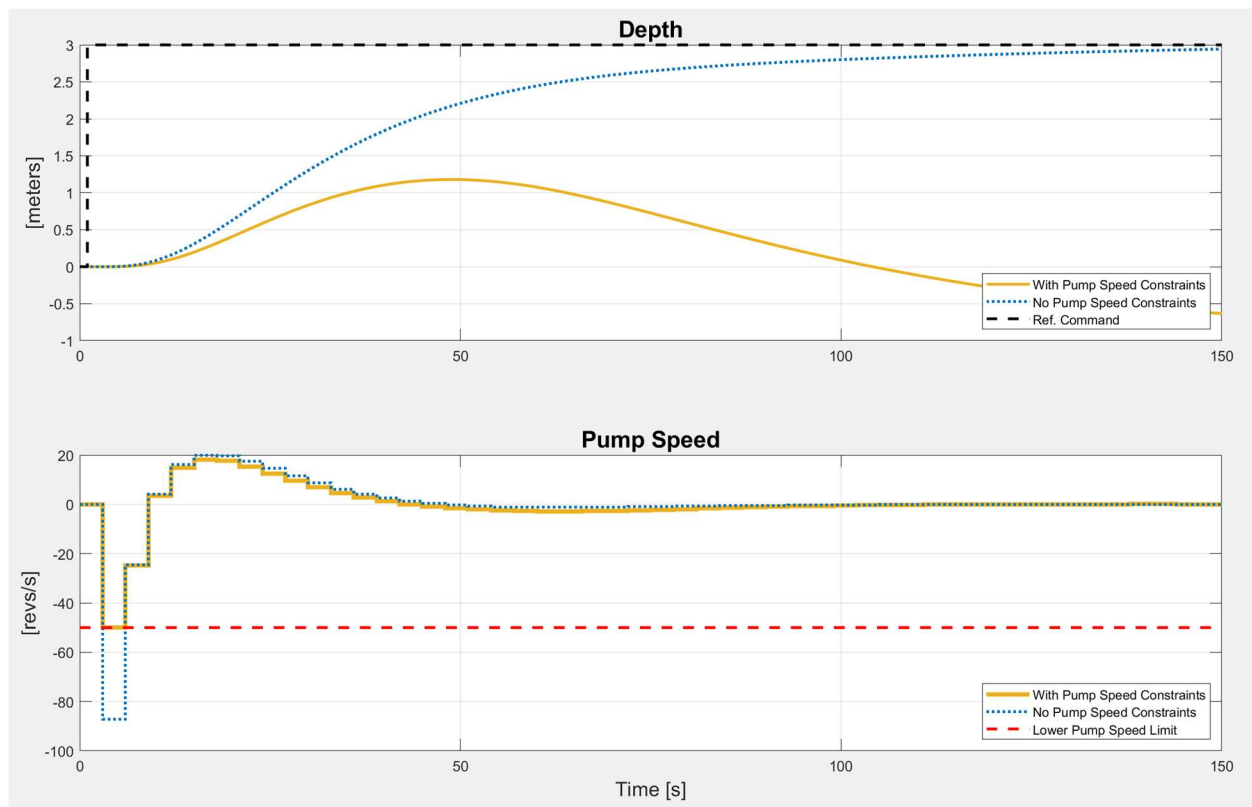


Figure 11: Comparison of Responses With and Without Pump Speed Constraints

Using a reference governor to handle constraints is an add-on strategy which has the benefit of leaving the dual lead compensator intact. Based on a recommendation from Prof. Rock, a reference governor was created to ramp the reference signal fed to the control loop. It does this at a maximum rate of 1.2 meters every 3 seconds. This rate was determined to be safe through trial and error. If the magnitude of the difference between the desired depth and the governed reference depth drops below 1.2, then the governed reference depth is simply set to the desired depth. In other words, if the desired depth is 4 m and the starting depth is zero, then the governed reference depth will be 1.2 m after 3 seconds, 2.4 m after 6 seconds, 3.6 m after 9 seconds, and 4.0 m after 12 seconds. A Simulink model (`cpf_model_with_discrete_controller_and_ref_gov.slx`) was created to simulate system performance. The block diagram can be found in the Appendix. A simulation of three large depth changes is presented in Figure 12. Observe, the CPF is now capable of tracking these large step commands because the pump speed does not saturate. Furthermore, the reference governor has removed the overshoot.

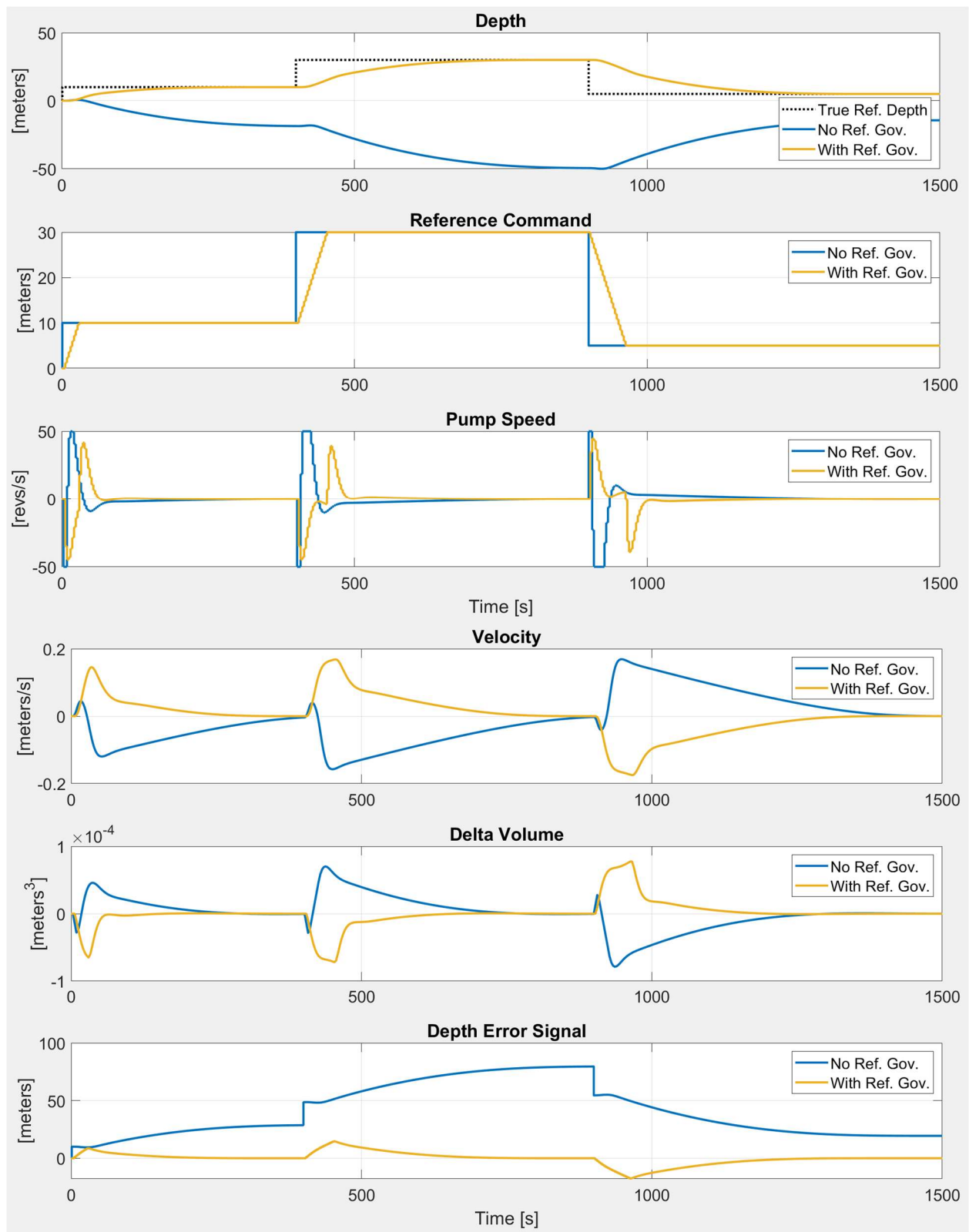


Figure 12: Comparison of Responses With and Without the Reference Governor

C++ Implementation

A C++ script was created to demonstrate how the depth controller and reference governor combination can be implemented. It is available in the Appendix. The script was validated by exporting the actual reference depth commands and simulated depth measurements from MATLAB to .txt files and feeding them into the C++ script. The resulting pump speed commands were then saved to another .txt file. Figure 13 below compares the Simulink reference governor plus dual lead compensator to the C++ reference governor plus dual lead compensator. Observe, the results are equivalent.

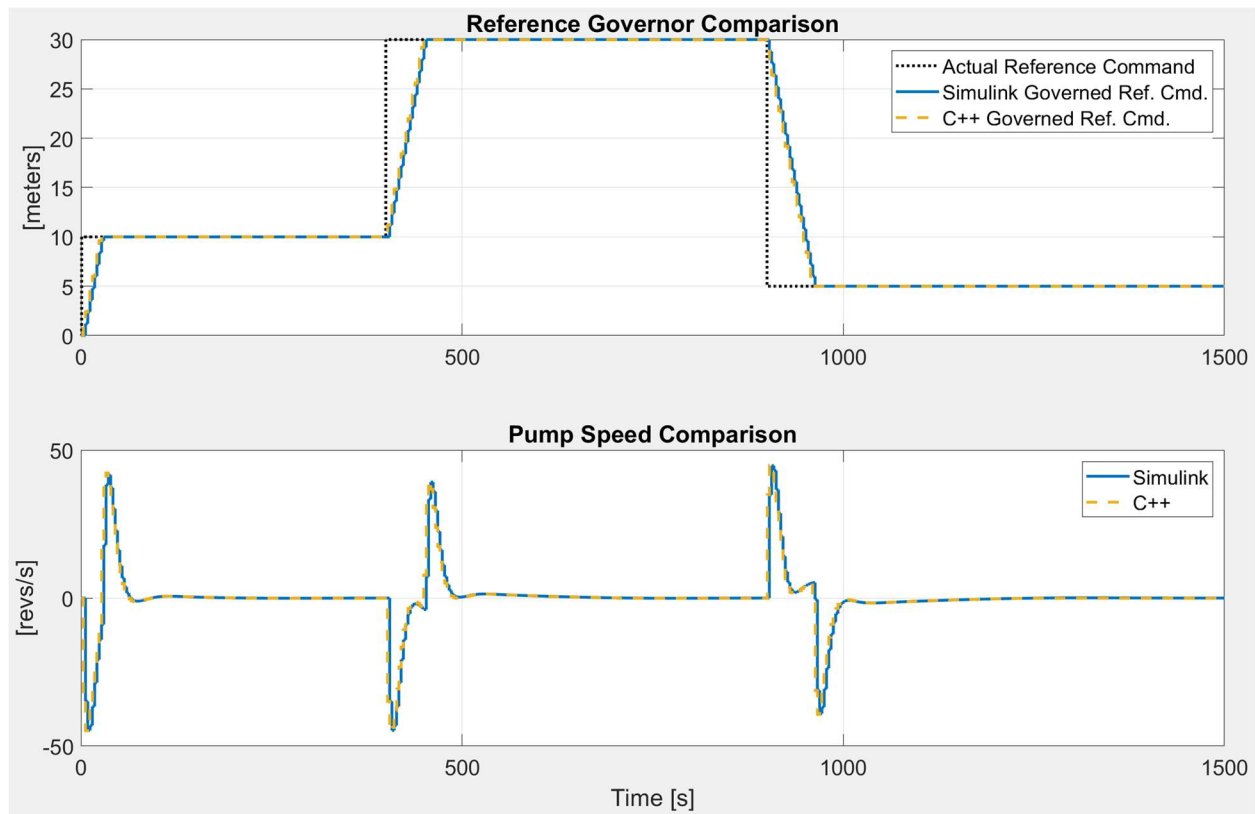


Figure 13: Simulink vs. C++ Controller Responses

Suggested Future Work

Add Integral Control

There are two practical reasons for why this should be done. First, adding integral control would improve robustness against model parameter uncertainty (e.g. the drag coefficient) by removing any steady state error. Second, and more importantly, this will allow the CPF to reject large depth disturbances. Currently, this control system can only reject depth disturbances if the pump speed does not saturate. If it saturates, steady state error results. Figure 14 below illustrates this point. At $t = 50$ s, a 1.7 m step disturbance is introduced. Notice that the pump speed almost saturates (49.4 revs/s), but the disturbance is completely rejected by $t = 250$ s. However, at $t = 300$ s, a 2 m step disturbance is introduced. The pump speed saturates and fails to bring the CPF back to the reference depth. This can also happen with smaller disturbances if the CPF is changing depth at the same time (i.e. when the depth error signal is not zero).

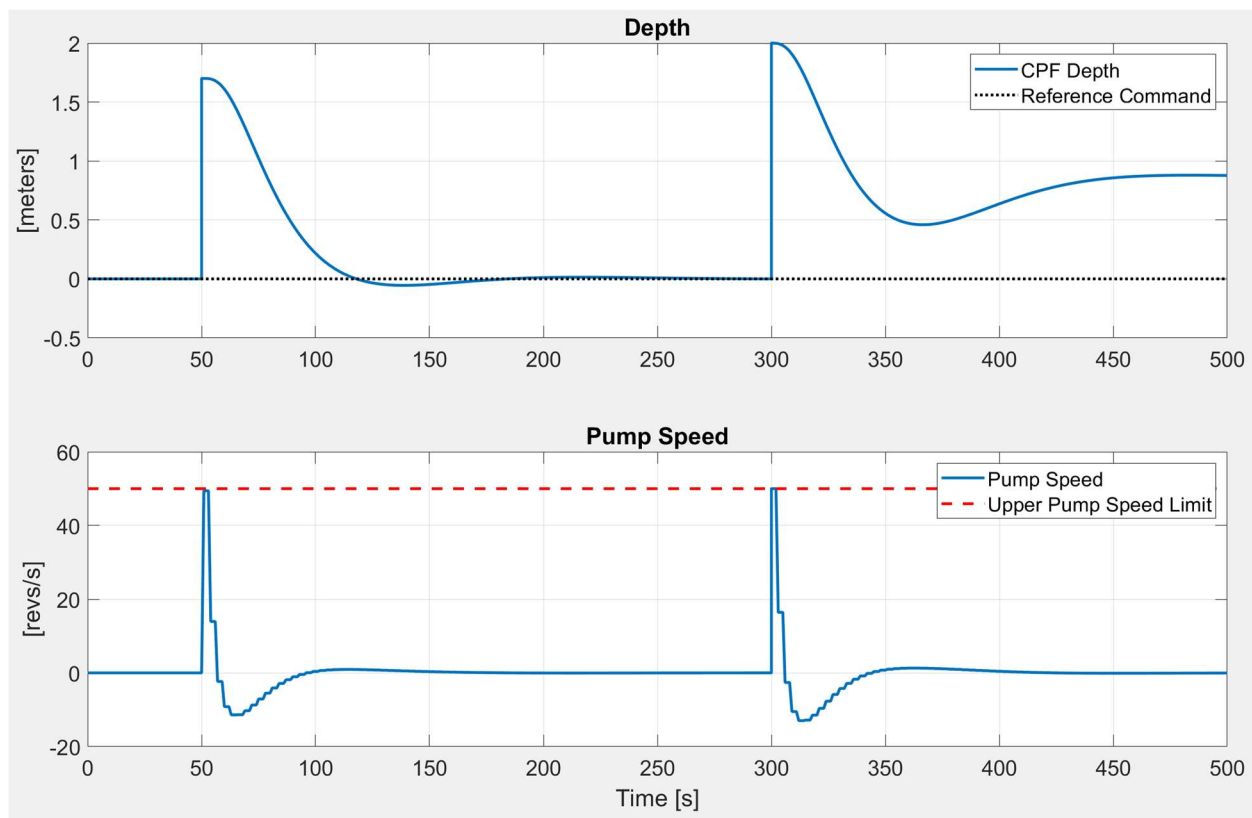


Figure 14: Simulation Showing Failure to Reject Large Depth Disturbances

Design and Simulate a Pressure Sensor Noise Filter

The stochastic nature of the pressure sensor noise is a reality that needs to be handled before this depth control system can go to sea. Median and/or moving average filters would likely provide the simplest fix. More complex, but higher performance, solutions are the Kalman filter or the Savitsky-Golay filter. The author recommends the use of a Kalman filter because it minimizes estimation error variance and would open the door to more advanced control techniques such as the Linear Quadratic Regulator (LQR).

Conclusion

The performance requirements for the depth control system of MBARI's CPF are more demanding than those of an open ocean float. Specifically, overshoot must be minimized to avoid striking the sea floor and to take advantage of seaward currents. On top of this, the controller must not violate the oil pump's relatively-tight max speed constraint. To meet these specifications, a dual lead compensator with reference governor control strategy was proposed, evaluated in MATLAB and Simulink, and finally tested in C++. Although this code is ready to be ported to C# and implemented on the CPF, the author believes that the addition of integral control and a pressure sensor noise filter are required before this system is ready for the ocean. The CPF project stands to completely change the way we understand coastal ecosystem processes, enabling us to make more effective policy decisions regarding marine resource conservation. It is the hope of the author that this internship project helps MBARI move closer to achieving that goal.

Acknowledgements

First, I would like to thank my mentor, Gene Massion, for opening the door to the world of ocean engineering for me. Had Gene not encouraged me to pursue this bonus internship project, I would have missed out on a fantastic and practical learning experience. Gene's emphasis on a disciplined approach to engineering is a lesson I take with me to future projects. Second, I would like to thank both Dr. George Matsumoto and Linda Kuhnz for coordinating MBARI's intern program. It is clear that they genuinely care about the professional growth and well-being of the interns through their support every step of the way. Third, I would like to thank all the MBARI staff, especially Eric Martin, Brian Kieft, and Carole Sakamoto. I had a terrific time working in the Chemical Sensors Lab, in the LRAUV Lab, and on the R/V Paragon with you all. Fourth, I would like to thank the David & Lucile Packard Foundation without whose generous support none of this would have been possible. Last, but definitely not least, I would like to thank my fellow 2018 summer interns. I am so proud of the fact that we completed everything on our list of summer adventure plans and I can easily say that this was one of the best summers I've ever had. It was wonderful getting to know you all and I do hope that we stay in touch.

References

- [1] J. H. Ryther, "Photosynthesis and fish production in the sea". *Science*, 166, pp. 72-76, 1969
- [2] Benway, et. al. (Editors), "An interdisciplinary science plan for research in North American continental margin systems." *Report of the Coastal CARbon Synthesis (CCARS) community workshop, August 19-21, 2014, Ocean Carbon and Biogeochemistry Program and North American Carbon Program*, pp. 68, 2015
- [3] J. S. Freudenberg, *A First Graduate Course in Feedback Control*. University of Michigan Department of Electrical Engineering and Computer Science, EECS 565 Textbook, 2018.
- [4] G. Franklin, J. D. Powell, A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley Publishing Company, 3rd Edition, 1994.

Appendix

A.1 MATLAB Script

A.2 Simulink Model

A.3 C++ Implementation

```
1 % =====
2 % AUTHOR      : Brian Ha
3 % MENTOR      : Gene Massion
4 % DATE CREATED : 2018-07-11
5 % LAST REVISED : 2018-09-26
6 % MBARI GROUP  : Chemical Sensors Lab
7 % PROJECT     : Coastal Profiling Float (CPF) Depth Control
8 % =====
9
10
11 % Clean Up
12 close all
13 clear variables
14 clc
15
16
17 % Plot Formatting
18 yellow = [0.9290, 0.6940, 0.1250]; % Custom Plot Color Codes
19 axisLabelFontSize = 15;
20 titleFontSize = 17;
21 legendFontSize = 14;
22
23
24 % #####
25
26 % Section 1: The CPF Model
27
28 % #####
29
30
31 % Float Parameters
32 floatLength = 1.016; % [m] This is equal to 40 in.
33 floatDiameter = 0.2032; % [m] This is equal to 8 in.
34 PlatformArea = (pi/4) * floatDiameter^2; % [m^2]. Cylinder frontal area.
35
36 PlatformMass = 38; % [kg]
37 VirtualMassFactor = 1.25; % Accounts for water being dragged
38 m = PlatformMass * VirtualMassFactor; % [kg]
39
40 pumpDisplacement = 1.56e-7; % [m^3/rev] Per Oildyne spec sheet
41 pumpEff = 0.6;
42
43 samplingPeriod = 3; % [s] New pressure meas. every 3 s
44
45
46 % Constraints
47 upperPumpSpeedLimit = 50; % [revs/s] This is equal to 3000 rpm.
48 lowerPumpSpeedLimit = -50; % [revs/s]
49
50
```

```
51 % Assumptions
52 densitySW = 1025; % [kg/m^3] This is for seawater. Ignores compressibility.
53 g = 9.807; % [m/s^2] Gravitational acceleration.
54 velFreeStream = 0.1; % [m/s] This is equal to 10 cm/s.
55 kinematicViscosity = 13.60e-7; % [m^2/s]. Taken from http://web.mit.edu/seawater/2017/MIT_Seawater_Property_Tables_r2a.pdf
56
57
58 % Estimation of drag coefficient range
59 Re = velFreeStream*floatDiameter/kinematicViscosity; % Just checking Reynolds Number...
60
61 % CdMin is interpolated below from the values (based on frontal area)
62 % given for a flat-faced cylinder on Table 7.3 (page 491) of Fluid
63 % Mechanics, 7th Ed. by Frank M. White. Copyright 2011. Available at
64 % https://hellcareers.files.wordpress.com/2016/01/fluid-mechanics-seventh-edition-by-frank-m-white.pdf.
65
66 CdMin = (0.99 - 0.87)/(8 - 4) * (floatLength/floatDiameter - 4) + 0.87;
67
68 CdMax = 1.5; % Taken from Laughlin Barker's paper (2014 MBARI intern).
69
70 Cd = CdMin; % User can set this to either CdMax or CdMin
71
72
73 % Equilibrium Point
74 x_eq = [0;0;0];
75 u_eq = 0;
76
77 % Symbolic Linearization about Equilibrium (via Taylor Series Expansion)
78 [Aeq, Beq] = symLin(x_eq, u_eq, densitySW, Cd, PlatformArea, g, m, pumpDisplacement, pumpEff);
79 C = [1 0 0];
80 D = 0;
81
82 disp('CPF Model')
83 P = ss(Aeq, Beq, C, D)
84
85 %{
86 % Linearization about Equilibrium by hand calculation (for verification)
87 A = [0 1 0; 0 densitySW*Cd*PlatformArea*x_eq(2)/m -densitySW*g/m; 0 0 0];
88 B = [0; 0; pumpDisplacement*pumpEff];
89
90 p = ss(A,B,C,D);
91 %}
92
93
94 % Kalman Controllability Test
95 disp('KALMAN CONTROLLABILITY TEST')
```



```
96 disp(['Rank of Controllability Matrix = ', num2str(rank(ctrb(P)))] )
97 disp('If rank = 3, system is fully controllable.')
98 disp(' ')
99
100
101 % Kalman Observability Test
102 disp('KALMAN OBSERVABILITY TEST')
103 disp(['Rank of Observability Matrix = ', num2str(rank(observ(P)))] )
104 disp('If rank = 3, system is fully observable.')
105 disp(' ')
106
107
108 % #####
109
110 % Section 2: The Controller
111
112 % #####
113
114
115 % The Dual Lead Compensator
116 s = tf('s');
117 disp('Continuous Controller, C(s) = ')
118 C = -45*(s+0.015)*(s+0.001)/((s+0.17)*(s+0.15));
119 zpkm(C)
120
121
122 % Important Transfer Functions
123 L = tf(P)*C; % Open-Loop Transfer Function
124 S = minreal(1/(1 + L)); % Plant Output Disturbance Sensitivity Function
125 T = minreal(1 - S); % Command and Noise Sensitivity Function
126 CS = minreal(C*S); % Control Signal Sensitivity Function
127 SP = minreal(S*P); % Plant Input Disturbance Sensitivity Function
128
129
130
131
132 % =====
133 % LTI CONTINUOUS-TIME SIMULATION
134 % =====
135
136 % Unit Step Responses of the Four Closed Loop Transfer Functions
137 referenceCommandSize = 1; % [m]
138
139 figure(1)
140
141 opt = stepDataOptions('StepAmplitude', referenceCommandSize);
142
143 subplot(2,2,1)
144 step(T, opt)
145 [y, t] = step(T, opt);
```

```
146 hold on
147 plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])), 'r:')
148 plot([150:5:t(end)], 1.05*referenceCommandSize*ones(1, length([150:5:t(end)])), 'r--')
149 plot([150:5:t(end)], 0.95*referenceCommandSize*ones(1, length([150:5:t(end)])), 'r--')
150 grid on
151 title('Command and Noise Response, T(s)', 'FontSize', titleFontSize)
152 xlabel('Time', 'FontSize', axisLabelFontSize)
153 ylabel('Depth [m]', 'FontSize', axisLabelFontSize)
154
155
156 subplot(2,2,2)
157 step(CS, opt)
158 [y, t] = step(C*S);
159 hold on
160 plot(t, upperPumpSpeedLimit*ones(1, length(t)), 'r--')
161 plot(t, lowerPumpSpeedLimit*ones(1, length(t)), 'r--')
162 grid on
163 title('Control Signal Response, CS(s)', 'FontSize', titleFontSize)
164 xlabel('Time', 'FontSize', axisLabelFontSize)
165 ylabel('Pump Speed [revs/s]', 'FontSize', axisLabelFontSize)
166
167
168 subplot(2,2,3)
169 step(S, opt)
170 grid on
171 title('Output Disturbance Rejection, S(s)', 'FontSize', titleFontSize)
172 xlabel('Time', 'FontSize', axisLabelFontSize)
173 ylabel('Depth [m]', 'FontSize', axisLabelFontSize)
174
175
176 subplot(2,2,4)
177 step(SP, opt)
178 grid on
179 title('Input Disturbance Rejection, SP(s)', 'FontSize', titleFontSize)
180 xlabel('Time', 'FontSize', axisLabelFontSize)
181 ylabel('Depth [m]', 'FontSize', axisLabelFontSize)
182
183 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
184 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
185
186
187
188
189 % =====
190 % LTI vs. NLTI, CONTINUOUS-TIME COMPARISON
191 % =====
192
193 referenceCommandSize = 1; % [m]
```

```

194 referenceCommandTime = 0;           % [s]
195 outputDisturbanceSize = 0;           % [m]
196 outputDisturbanceTime = 0;          % [s]
197 tFinal = 350;                        % [s]
198
199
200 [depth_lti, time_lti] = step(T, tFinal);
201 [time_c, ~, ~, depth_c, ~, ctrl_c, ~] = sim(
('cpf_model_with_continuous_controller'));
202
203 figure(2)
204 subplot(2,1,1)
205 plot(time_lti, depth_lti)
206 hold on
207 plot(time_c, depth_c, 'color', yellow)
208 plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])), 'r:')
209 plot(150:5:time_c(end), 1.05*referenceCommandSize*ones(1, length(150:5:time_c
(end))), 'r--')
210 plot(150:5:time_c(end), 0.95*referenceCommandSize*ones(1, length(150:5:time_c
(end))), 'r--')
211 grid on
212 legend({'No Drag', 'With Drag', '25% Overshoot Req.', '150 s Settling Time Req.'},
'location', 'southeast', 'FontSize', legendFontSize)
213 title('Depth', 'FontSize', titleFontSize)
214 ylabel(['meters'], 'FontSize', axisLabelFontSize)
215
216
217 [ctrl_lti, time_lti] = step(CS, tFinal);
218 subplot(2,1,2)
219 plot(time_lti, ctrl_lti);
220 hold on
221 plot(time_c, ctrl_c, 'color', yellow)
222 grid on
223 legend({'No Drag', 'With Drag'}, 'location', 'southeast', 'FontSize',
legendFontSize)
224 title('Pump Speed', 'FontSize', titleFontSize)
225 ylabel(['revs/s'], 'FontSize', axisLabelFontSize)
226 xlabel('Time [s]', 'FontSize', axisLabelFontSize)
227
228 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
229 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
230
231
232
233 % =====
234 % NLTI, CONTINUOUS-TIME vs. DISCRETE-TIME COMPARISON
235 % =====
236
237 % Discretization
238 disp('Discrete Controller, C(z) = ')

```

```

239 Cz = c2d(C, samplingPeriod, 'matched')
240 Pz = c2d(P, samplingPeriod, 'zoh');
241 Lz = tf(Pz)*Cz; % Open-Loop Transfer Function
242 Sz = minreal(1/(1 + Lz)); % Plant Output Disturbance Sensitivity Function
243 Tz = minreal(1 - Sz); % Command and Noise Sensitivity Function
244 CSz = minreal(Cz*Sz); % Control Signal Sensitivity Function
245 SPz = minreal(Sz*Pz); % Plant Input Disturbance Sensitivity Function
246
247
248 % Simulation
249 referenceCommandSize2 = 0; % [m]
250 referenceCommandTime2 = 0; % [s]
251 referenceCommandSize3 = 0; % [m]
252 referenceCommandTime3 = 0; % [s]
253 [time_d, ~, ~, depth_d, ~, ctrl_d, ~, ~] = sim(
('cpf_model_with_discrete_controller'));
254
255
256 figure(3)
257 subplot(2,1,1)
258 plot(time_c, depth_c);
259 hold on
260 plot(time_d, depth_d, 'color', yellow)
261 plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])), 'r:')
262 plot(150:5:time_c(end), 1.05*referenceCommandSize*ones(1, length(150:5:time_c
(end))), 'r--')
263 plot(150:5:time_c(end), 0.95*referenceCommandSize*ones(1, length(150:5:time_c
(end))), 'r--')
264 legend({'Continuous', 'Discrete', '25% Overshoot Req.', '150 s Settling Time
Req.'}, 'location', 'southeast', 'FontSize', legendFontSize)
265 grid on
266 ylabel('[meters]', 'FontSize', axisLabelFontSize)
267 title('Depth', 'FontSize', titleFontSize)
268
269
270 subplot(2,1,2)
271 plot(time_c, ctrl_c);
272 hold on
273 stairs(time_d, ctrl_d, 'linewidth', 2, 'color', yellow)
274 legend({'Continuous', 'Discrete'}, 'location', 'southeast', 'FontSize',
legendFontSize)
275 grid on
276 title('Pump Speed', 'FontSize', titleFontSize)
277 xlabel('Time [s]', 'FontSize', axisLabelFontSize)
278 ylabel('[revs/s]', 'FontSize', axisLabelFontSize)
279
280 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
281 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
282
283 % Bode Plot of the Open Loop Transfer Function, L(z)

```

```
284 figure(4)
285 margin(Lz)
286 grid on
287 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
288 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
289
290
291 % Nyquist Plot of the Open Loop Transfer Function, L(z)
292 figure(5)
293 nyquist(Lz)
294
295 % Draw a black unit circle
296 circle = -1+ (-s+1)/(s+1);
297 omegac = logspace(-3,3,400);
298 [reC,imC] = nyquist(circle,omegac);
299 reC = squeeze(reC);
300 imC = squeeze(imC);
301 hold on
302 plot(reC,imC,'k-',reC,-imC,'k-')
303
304 % calculate stability radius as the radius of the circle
305 % centered at the critical point that is avoided by the Nyquist plot
306 [mag,~] = bode(Sz);
307 mag = squeeze(mag);
308 stab_rad = 1/max(mag);
309
310 axis([-3 3 -3 3])
311 axis('equal')
312 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
313 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
314 title(['Nyquist Plot of L(z), Stability Radius = ', num2str(stab_rad)], 'FontSize', titleFontSize)
315
316
317 % Bode Plot of S(z) and T(z)
318 figure(6)
319 options = bodeoptions('cstprefs');
320 options.FreqUnits = 'Hz';
321 options.Xlabel.FontSize = axisLabelFontSize;
322 options.Ylabel.FontSize = axisLabelFontSize;
323 %options.MagUnits = 'abs';
324 bode(Sz, options)
325 hold on
326 bode(Tz, options)
327 grid on
328 title('Bode Plot of S(z) and T(z)', 'FontSize', titleFontSize)
329 legend({'S(z)', 'T(z)'}, 'FontSize', legendFontSize)
330
331 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
332 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
```

```
333
334 % #####
335
336 % Section 3: The Reference Governor
337
338 % #####
339
340 % Simulation Demonstrating Ability to Track Large Reference Commands
341 startingDepth = 0;           % [m]
342 referenceCommandSize = 10;   % [m]
343 referenceCommandTime = 1;    % [s]
344 referenceCommandSize2 = 20;  % [m]
345 referenceCommandTime2 = 400; % [s]
346 referenceCommandSize3 = -25; % [m]
347 referenceCommandTime3 = 900; % [s]
348 outputDisturbanceSize = 0;   % [m]
349 outputDisturbanceTime = 0;
350 outputDisturbanceSize2 = 0;  % [m]
351 outputDisturbanceTime2 = 0;
352 tFinal = 1500;              % [s]
353 maxSetPointIncrement = 0.40;  % [m]
354 maxSetPointDecrement = -1*maxSetPointIncrement; % [m]
355
356
357 [time_d, ~, ref_d, depth_d, vel_d, ctrl_d, delta_vol_d, error_d] = sim(
('cpf_model_with_discrete_controller'));
358 [time_gov, ~, ref, ref_gov, depth_gov, vel_gov, ctrl_gov, delta_vol_gov,
governed_error] = sim('cpf_model_with_discrete_controller_and_ref_gov');
359
360
361 figure(7)
362 subplot(3,1,1)
363 plot(time_d, ref_d, 'k:')
364 hold on
365 plot(time_d, depth_d)
366 plot(time_gov, depth_gov, 'color', yellow)
367 legend({'True Ref. Depth', 'No Ref. Gov.', 'With Ref. Gov.'}, 'location',
'southeast', 'FontSize', legendFontSize)
368 grid on
369 title('Depth', 'FontSize', titleFontSize)
370 ylabel('[meters]', 'FontSize', axisLabelFontSize)
371
372
373 subplot(3,1,2)
374 plot(time_d, ref_d)
375 hold on
376 stairs(time_gov, ref_gov, 'color', yellow, 'linewidth', 2)
377 legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize',
legendFontSize)
378 grid on
```

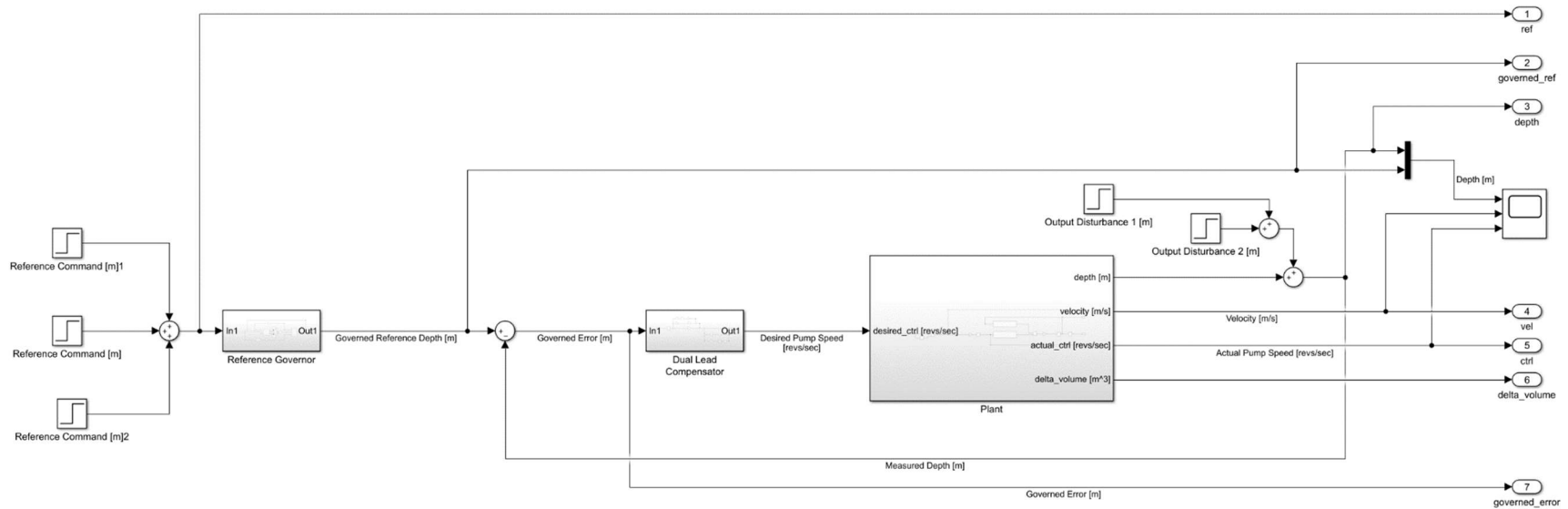
```
379 title('Reference Command', 'FontSize', titleFontSize)
380 ylabel(['meters'], 'FontSize', axisLabelFontSize)
381
382
383 subplot(3,1,3)
384 stairs(time_d, ctrl_d, 'linewidth', 2)
385 hold on
386 stairs(time_gov, ctrl_gov, 'color', yellow, 'linewidth', 2)
387 legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
388 grid on
389 title('Pump Speed', 'FontSize', titleFontSize)
390 xlabel('Time [s]', 'FontSize', axisLabelFontSize)
391 ylabel(['revs/s'], 'FontSize', axisLabelFontSize)
392
393 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
394 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
395
396
397 figure(8)
398
399 subplot(3,1,1)
400 plot(time_d, vel_d)
401 hold on
402 plot(time_gov, vel_gov, 'color', yellow)
403 legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
404 grid on
405 ylabel(['meters/s'], 'FontSize', axisLabelFontSize)
406 title('Velocity', 'FontSize', titleFontSize)
407
408
409 subplot(3,1,2)
410 plot(time_d, delta_vol_d)
411 hold on
412 plot(time_gov, delta_vol_gov, 'color', yellow)
413 grid on
414 ylabel(['meters^3'], 'FontSize', axisLabelFontSize)
415 legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
416 title('Delta Volume', 'FontSize', titleFontSize)
417
418
419 subplot(3,1,3)
420 plot(time_d, error_d)
421 hold on
422 plot(time_gov, governed_error, 'color', yellow)
423 grid on
424 xlabel('Time [s]', 'FontSize', axisLabelFontSize)
425 ylabel(['meters'], 'FontSize', axisLabelFontSize)
```

```
426 legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
427 title('Depth Error Signal', 'FontSize', titleFontSize)
428
429 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
430 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
431
432
433 % Simulation Showing Failure to Reject Large Depth Disturbances
434 referenceCommandSize = 0;      % [m]
435 referenceCommandTime = 0;      % [s]
436 referenceCommandSize2 = 0;     % [m]
437 referenceCommandTime2 = 0;     % [s]
438 referenceCommandSize3 = 0;     % [m]
439 referenceCommandTime3 = 0;     % [s]
440 outputDisturbanceSize = 1.7;   % [m]
441 outputDisturbanceTime = 50;    % [s]
442 outputDisturbanceSize2 = 2;    % [m]
443 outputDisturbanceTime2 = 300;  % [s]
444 tFinal = 500;                 % [s]
445
446 [time_gov, ~, ref, ~, depth_gov, ~, ctrl_gov, ~, ~] = sim(
('cpf_model_with_discrete_controller_and_ref_gov' ));
447
448 figure(9)
449 subplot(2,1,1)
450 plot(time_gov, depth_gov)
451 hold on
452 plot(time_gov, ref, 'k:')
453 grid on
454 title('Depth', 'FontSize', titleFontSize)
455 ylabel('[meters]', 'FontSize', axisLabelFontSize)
456 legend({'CPF Depth', 'Reference Command'}, 'FontSize', legendFontSize)
457
458 subplot(2,1,2)
459 plot(time_gov, ctrl_gov)
460 hold on
461 plot(time_gov, upperPumpSpeedLimit*ones(1, length(time_gov)), 'r--')
462 grid on
463 legend({'Pump Speed', 'Upper Pump Speed Limit'}, 'FontSize', legendFontSize)
464 title('Pump Speed', 'FontSize', titleFontSize)
465 xlabel('Time [s]', 'FontSize', axisLabelFontSize)
466 ylabel('[revs/s]', 'FontSize', axisLabelFontSize)
467
468 set(findall(gcf, 'type', 'line'), 'linewidth', 2)
469 set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
470
471
472
473
```

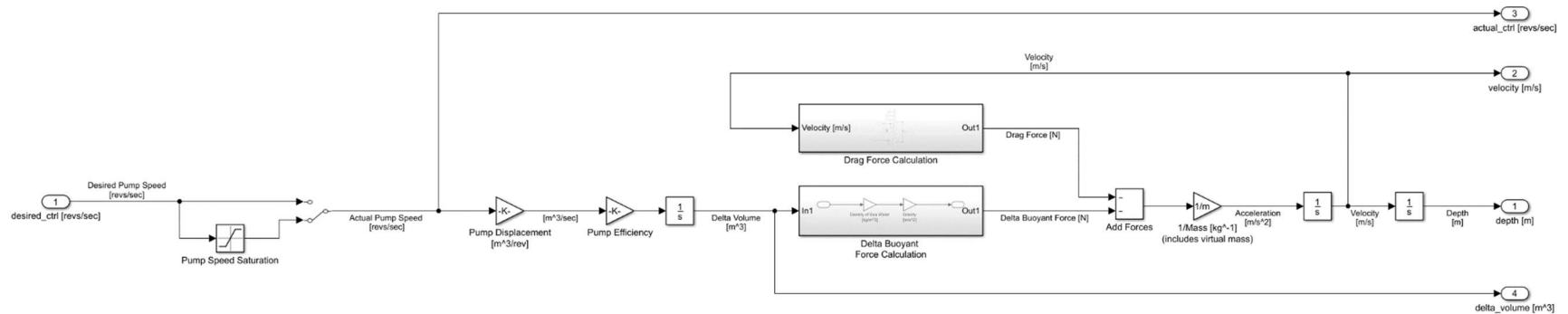


```
474
475 %% Functions
476 function [A, B] = symLin(x_eq, u_eq, densitySW, Cd, area, grav, mass, pumpDisplacement, pumpEff)
477 % Calculates the Jacobian A and B matrices about x_eq and u_eq
478
479 % Part I of II: Symbolic Calculation of General Jacobian
480
481 % Symbolic State Vector, x
482 position = sym('position');
483 velocity = sym('velocity');
484 delta_volume = sym('delta_volume');
485
486 x = [position; velocity; delta_volume];
487
488 % Symbolic Input Vector, u
489 motor_speed = sym('motor_speed'); % [rev/s]
490
491 u = [motor_speed];
492
493 % Non-Linear ODEs
494 x_dot(1,1) = velocity;
495 x_dot(2,1) = (-0.5*densitySW*area*Cd*velocity^2 - densitySW*grav*delta_volume)/mass;
496 x_dot(3,1) = motor_speed*pumpDisplacement*pumpEff;
497
498 A = jacobian(x_dot, x);
499 B = jacobian(x_dot, u);
500
501 % Part II of II: Calculation of Jacobian at x_eq and u_eq
502 position = x_eq(1);
503 velocity = x_eq(2);
504 delta_volume = x_eq(3);
505
506 motor_speed = u_eq;
507
508 A = eval(subs(A));
509 B = eval(subs(B));
510
511 end
```

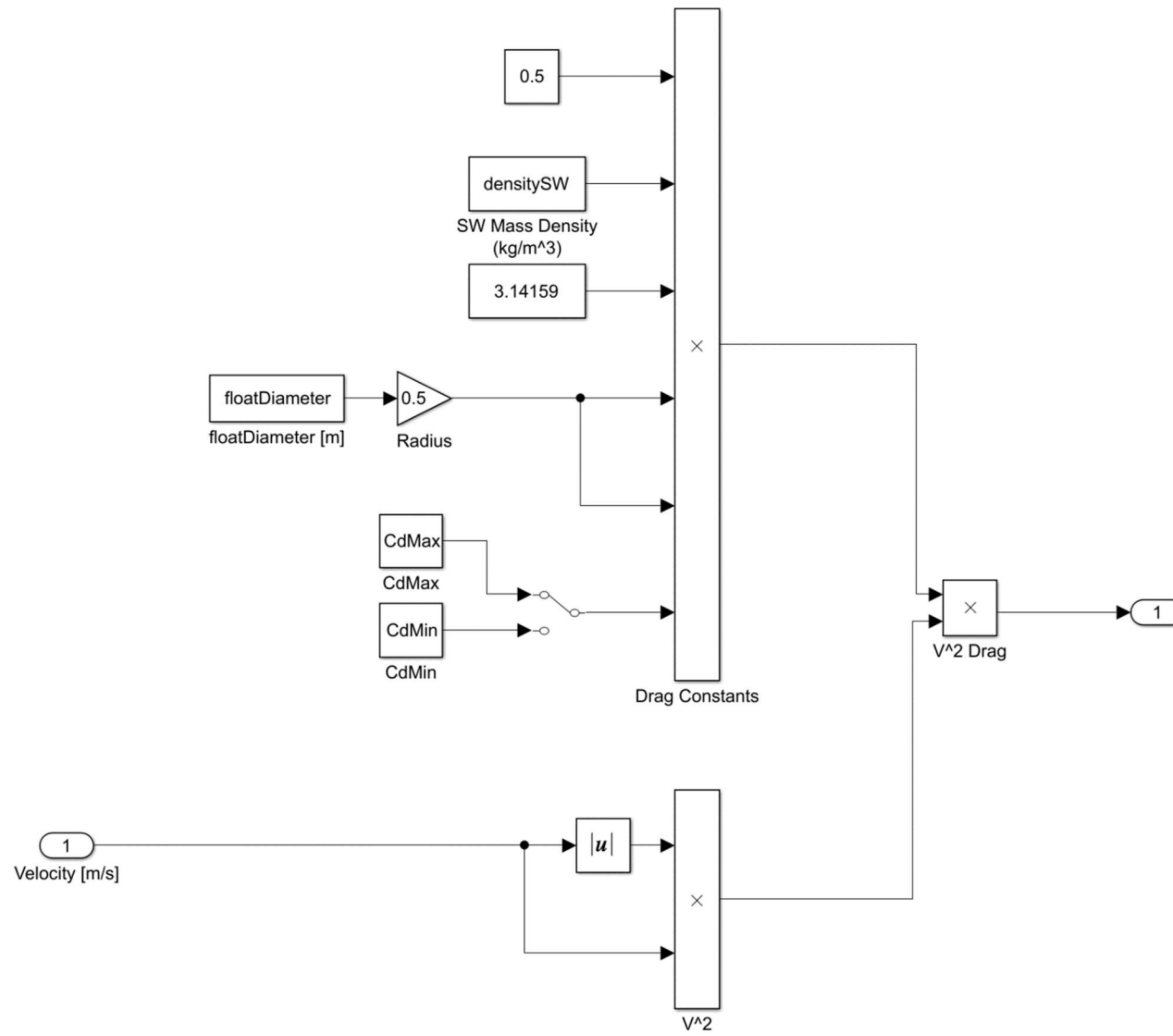
High Level Block Diagram



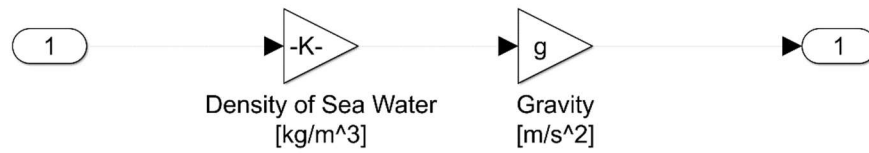
High Level Block Diagram → Plant



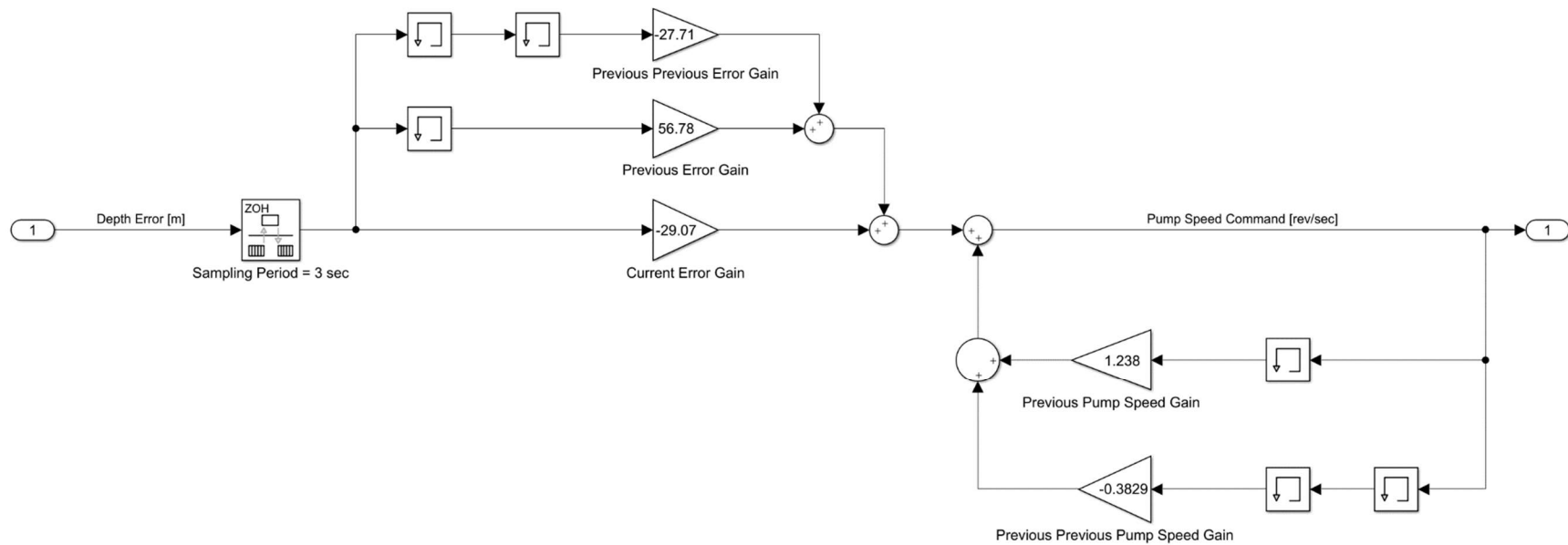
High Level Block Diagram → Plant → Drag Force Calculation



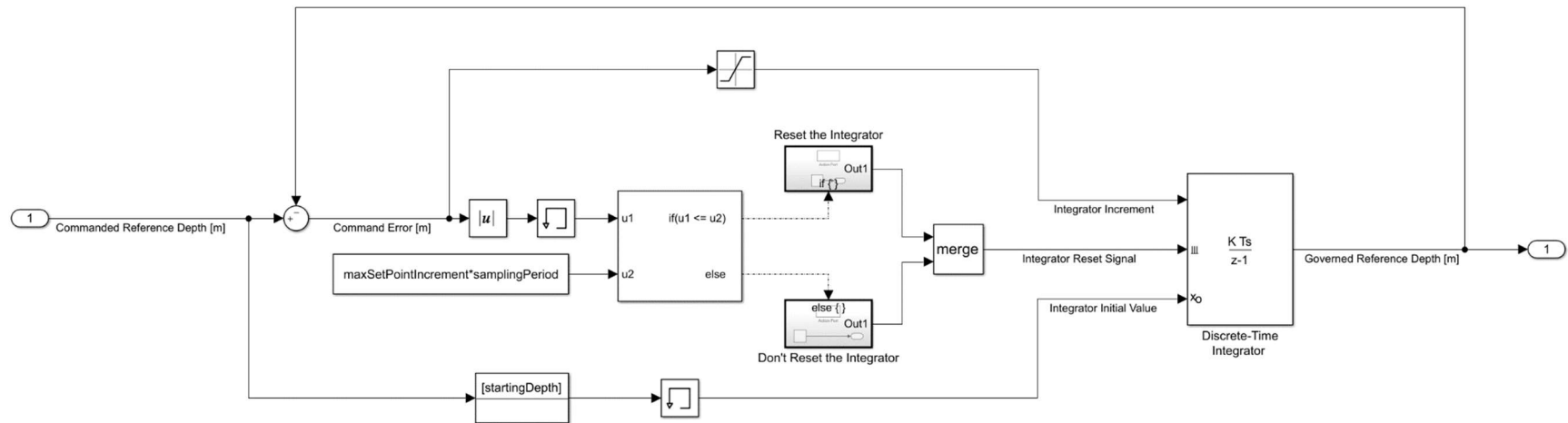
High Level Block Diagram → Plant → Delta Buoyant Force Calculation



High Level Block Diagram → Dual Lead Compensator



High Level Block Diagram → Reference Governor



```
1 #include <iostream>
2 #include <cmath>
3 #include <fstream>
4
5 using namespace std;
6
7 // Dual Lead Compensator Values
8 const double currentErrorGain = -29.07;
9 const double prevErrorGain = 56.78;
10 const double prevPrevErrorGain = -27.71;
11 const double prevPumpSpeedGain = 1.238;
12 const double prevPrevPumpSpeedGain = -0.3829;
13
14 const double pumpSpeedUpperLimit = 50.0;      // [revs/s]
15 const double pumpSpeedLowerLimit = -50.0;     // [revs/s]
16
17 static double currentPressureError = 0.0;
18 static double prevPressureError = 0.0;
19 static double prevPrevPressureError = 0.0;
20
21 static double nextPumpSpeedCmd = 0.0;
22 static double prevPumpSpeedCmd = 0.0;
23 static double prevPrevPumpSpeedCmd = 0.0;
24
25
26 // Reference Governor Values
27 const double maxSetPointIncrement = 1.2;      // [decibar]
28 const double maxSetPointDecrement = -1.2;     // [decibar]
29 static double integrator; // [decibar]
30
31
32 void InitializeGovernor(double startingPressure)
33 {
34     integrator = startingPressure;
35 }
36
37
38 double ReferenceGovernor(double desiredPressureSetPoint)
39 {
40     // AUTHOR: Brian Ha (2018 Summer Intern)
41
42     double setPointDelta; // [decibar]
43
44     cout << "TARGET PRESSURE = " << desiredPressureSetPoint << " decibar; ";
45
46
47     // If the integrator value is within 1.2 decibar of the target pressure,
48     // set it equal to the target pressure.
49     if (abs(desiredPressureSetPoint - integrator) <= maxSetPointIncrement)
50     {
```

```
51     integrator = desiredPressureSetPoint;
52 }
53 // If not, increment/decrement the integrator
54 else
55 {
56     setPointDelta = desiredPressureSetPoint - integrator;
57
58     // Limit how quickly the integrator can increment/decrement.
59     if (setPointDelta > maxSetPointIncrement)
60     {
61         setPointDelta = maxSetPointIncrement;
62     }
63     else if (setPointDelta < maxSetPointDecrement)
64     {
65         setPointDelta = maxSetPointDecrement;
66     }
67
68     integrator = integrator + setPointDelta;
69 }
70
71 cout << "INTEGRATOR = " << integrator << " decibar; ";
72
73 return integrator;
74 }
75
76
77 double DepthController(double governedPressureSetPoint, double
currentPlatformPressure)
78 {
79     // AUTHOR: Brian Ha (2018 Summer Intern)
80
81     // Calculate current pressure error
82     currentPressureError = governedPressureSetPoint - currentPlatformPressure;
83     cout << "PRESSURE ERROR = " << currentPressureError << " decibar; ";
84
85
86     // Calculate the next pump speed command.
87     nextPumpSpeedCmd = ((currentErrorGain * currentPressureError) +
88         (prevErrorGain * prevPressureError) +
89         (prevPrevErrorGain * prevPrevPressureError) +
90         (prevPumpSpeedGain * prevPumpSpeedCmd) +
91         (prevPrevPumpSpeedGain * prevPrevPumpSpeedCmd));
92
93
94     // Limit pump speed command magnitude to 50 revs/sec.
95     if (nextPumpSpeedCmd > pumpSpeedUpperLimit)
96     {
97         nextPumpSpeedCmd = pumpSpeedUpperLimit;
98     }
99     else if (nextPumpSpeedCmd < pumpSpeedLowerLimit)
```

```
100     {
101         nextPumpSpeedCmd = pumpSpeedLowerLimit;
102     }
103
104     // Deadband
105     if (nextPumpSpeedCmd < 0.05 && nextPumpSpeedCmd > -0.05)
106     {
107         nextPumpSpeedCmd = 0.0;
108     }
109
110     // Set historical values for next function call
111     prevPrevPumpSpeedCmd = prevPumpSpeedCmd;
112     prevPrevPressureError = prevPressureError;
113     prevPumpSpeedCmd = nextPumpSpeedCmd;
114     prevPressureError = currentPressureError;
115
116
117     // Convert from rev/sec to counts/sec
118     //nextPumpSpeedCmd = nextPumpSpeedCmd * COUNTSperREV;
119
120
121     // Return next pump speed command [counts/sec].
122     cout << "PUMP SPEED COMMAND: " << nextPumpSpeedCmd << " [revs/s]" << endl;
123     return (nextPumpSpeedCmd);
124 }
125
126
127
128 int main() {
129     double desiredPressureSetPoint;
130     double currentPlatformPressure;
131     double governedPressureSetPoint; // [decibar]
132     double startingPressure = 0.0; // [decibar]
133
134     // Setup File I/O
135     ifstream measuredPressures("pressures.txt");
136     ifstream targetPressures("target_pressures.txt");
137     ofstream pumpSpeeds("pump_speeds.txt");
138     ofstream governedTargetPressures("governed_target_pressures.txt");
139
140
141
142     // Initialize the integrator to the pressure at the current depth
143     InitializeGovernor(startingPressure);
144
145     // Read each value in the pressures.txt file, one at a time
146     while (measuredPressures >> currentPlatformPressure)
147     {
148         // Read each value in the target_pressures.txt file, one at a time
149         targetPressures >> desiredPressureSetPoint;
```



```
150
151     // Run the Reference Governor
152     governedPressureSetPoint = ReferenceGovernor(desiredPressureSetPoint);
153
154     // Write the governed target pressure to the governed_target_pressures file
155     governedTargetPressures << governedPressureSetPoint << endl;
156
157     // Run the Dual Lead Compensator
158     nextPumpSpeedCmd = DepthController(governedPressureSetPoint, ↵
currentPlatformPressure);
159
160     // Write the next pump speed command to the ctrl_signals.txt file.
161     pumpSpeeds << nextPumpSpeedCmd << endl;
162
163 }
164
165 // Close the output files.
166 pumpSpeeds.close();
167 governedTargetPressures.close();
168
169
170 return 0;
171 }
```