Coastal Profiling Float
Depth Control

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

Table of Contents

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.

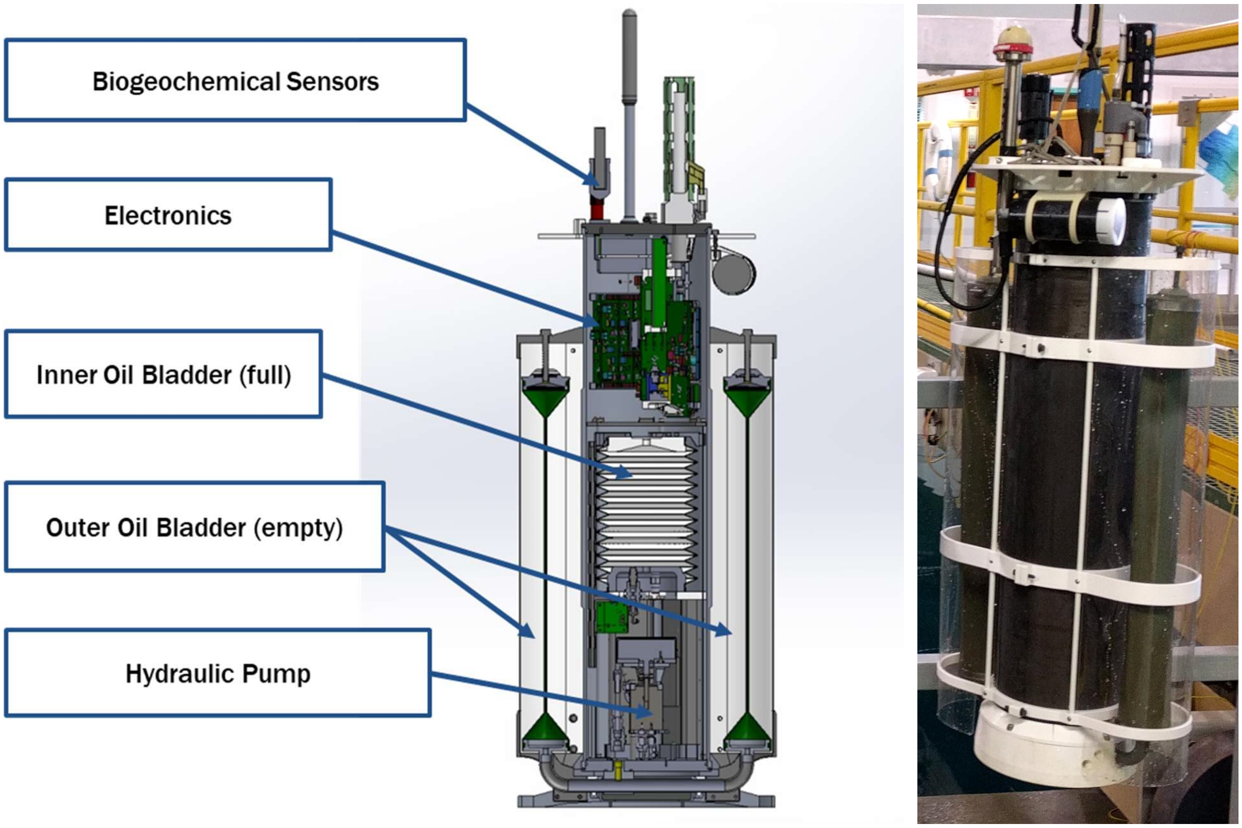


Figure 1: The Coastal Profiling Float (Prototype 3)

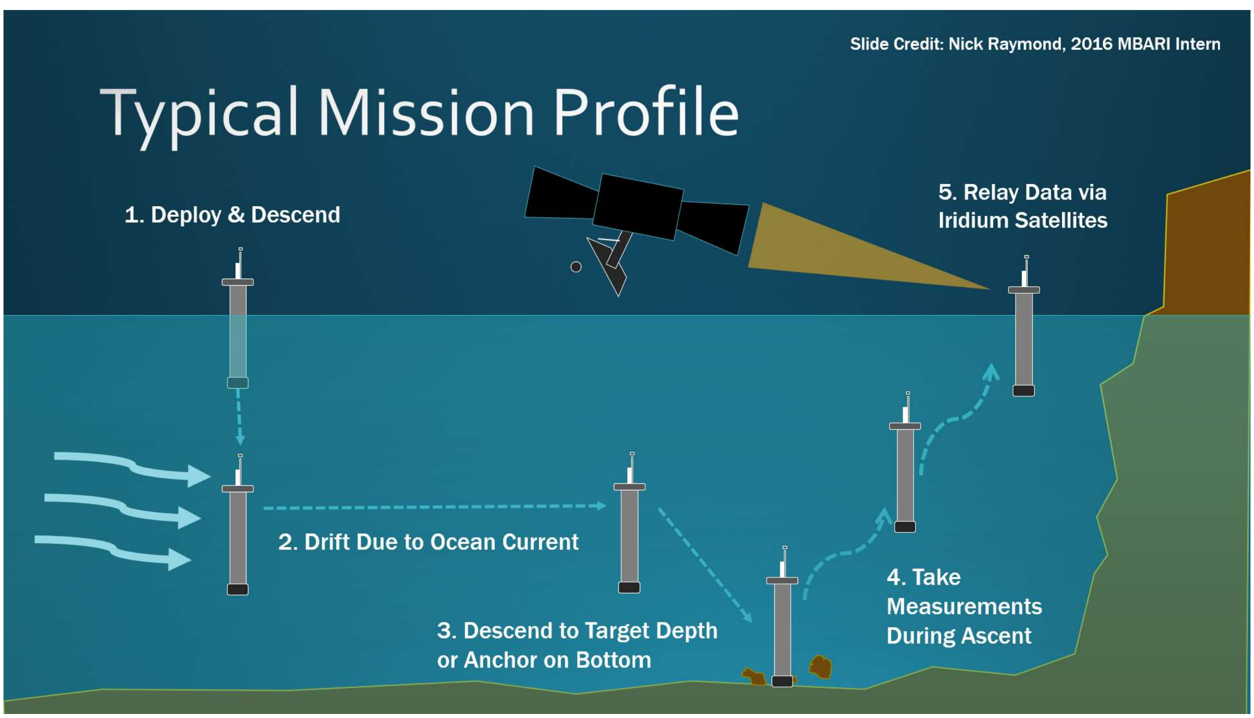


Figure 2: A Typical Mission Profile

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

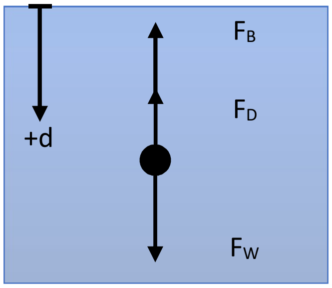


Figure 3: CPF Free Body Diagram

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).

$$\Sigma \mathrm{F} = -\mathrm{F_B} + \mathrm{F_W} - \mathrm{F_D} \tag{1}$$

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

$$\mathrm{F_B} = \mathrm{F_{B_{constant}}} + \mathrm{F_{B_{variable}}} \tag{2}$$

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

$$\Sigma \mathrm{F} = -\mathrm{F_{B_{variable}}} - \mathrm{F_D} \tag{3}$$

where

$$\mathrm{F}_{B_{variable}} = \rho g \Delta V \tag{4}$$

$$\mathrm{F_D} = \frac{1}{2}\rho A C_D \dot{d}^2 \tag{5}$$

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

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

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

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

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

$$\Delta\dot{V} = \varepsilon \delta \omega \tag{8}$$

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 linearized. Define the state vector x(t) and the input u(t) as:

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

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:

$$\delta x = x - x_{eq} \qquad (11)$$
$$\delta u = u - u_{eq} \qquad (12)$$

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:

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

where A and B are Jacobian matrices:

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

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

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{16}$$
$$y(t) = Cx(t) + Du(t) \tag{17}$$

where

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

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):

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

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:

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

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 <u>deterministic</u> 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).

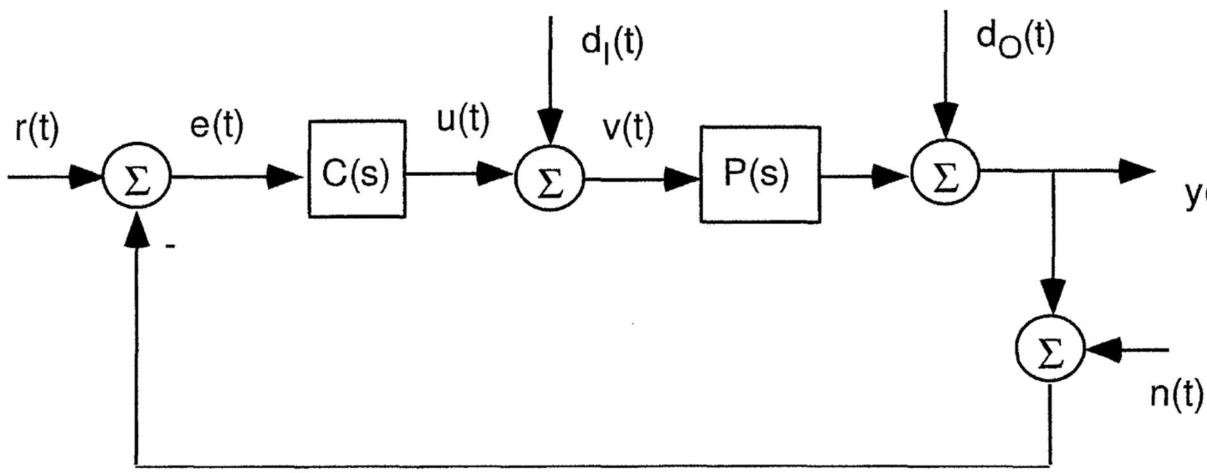


Figure 4: Feedback System Block Diagram. Image Source: [3]

Define the open loop transfer function as:

$$L(s) = P(s)C(s) \tag{24}$$

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)**

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

   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)**

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

   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)**

$$CS(s) = C(s) * S(s) \tag{27}$$

   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)**

$$SP(s) = S(s) * P(s) \tag{28}$$

   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:

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

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 revs/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.

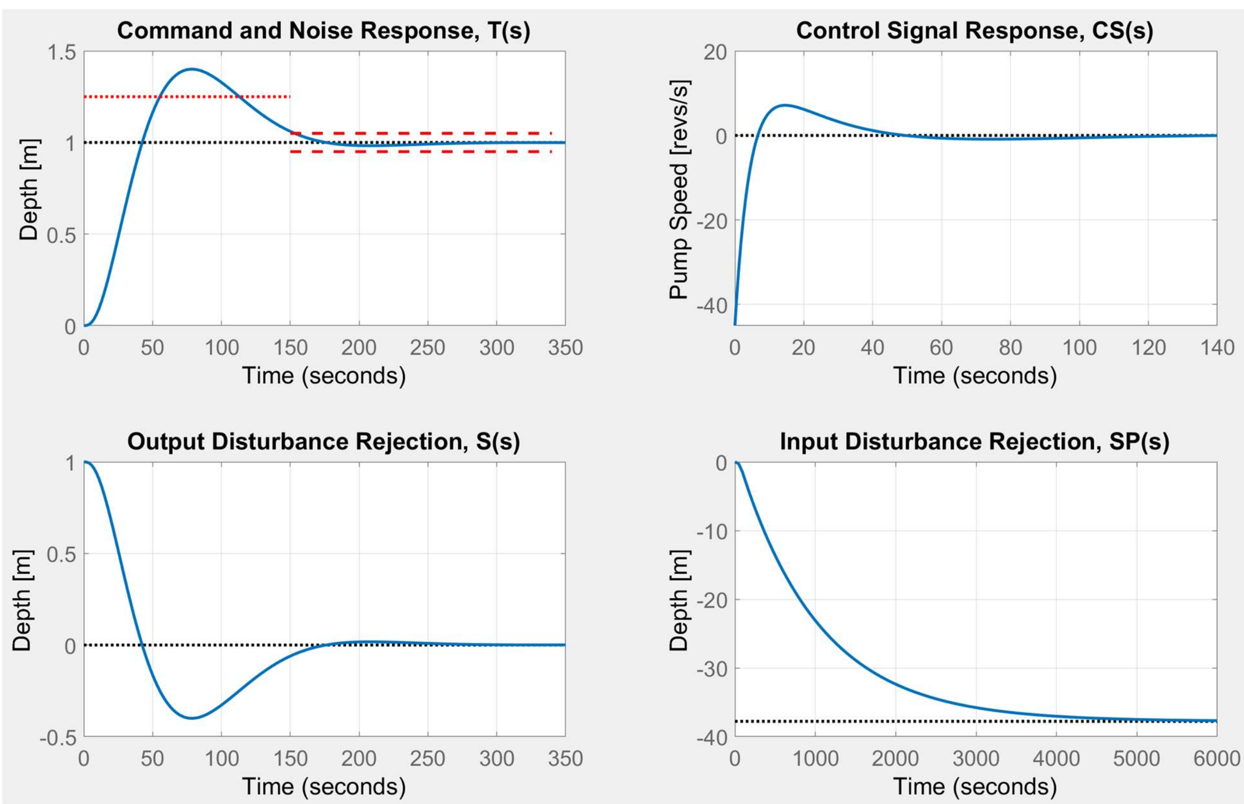


Figure 5: Unit Step Responses of the Four Closed Loop Transfer Functions (drag not simulated)

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.

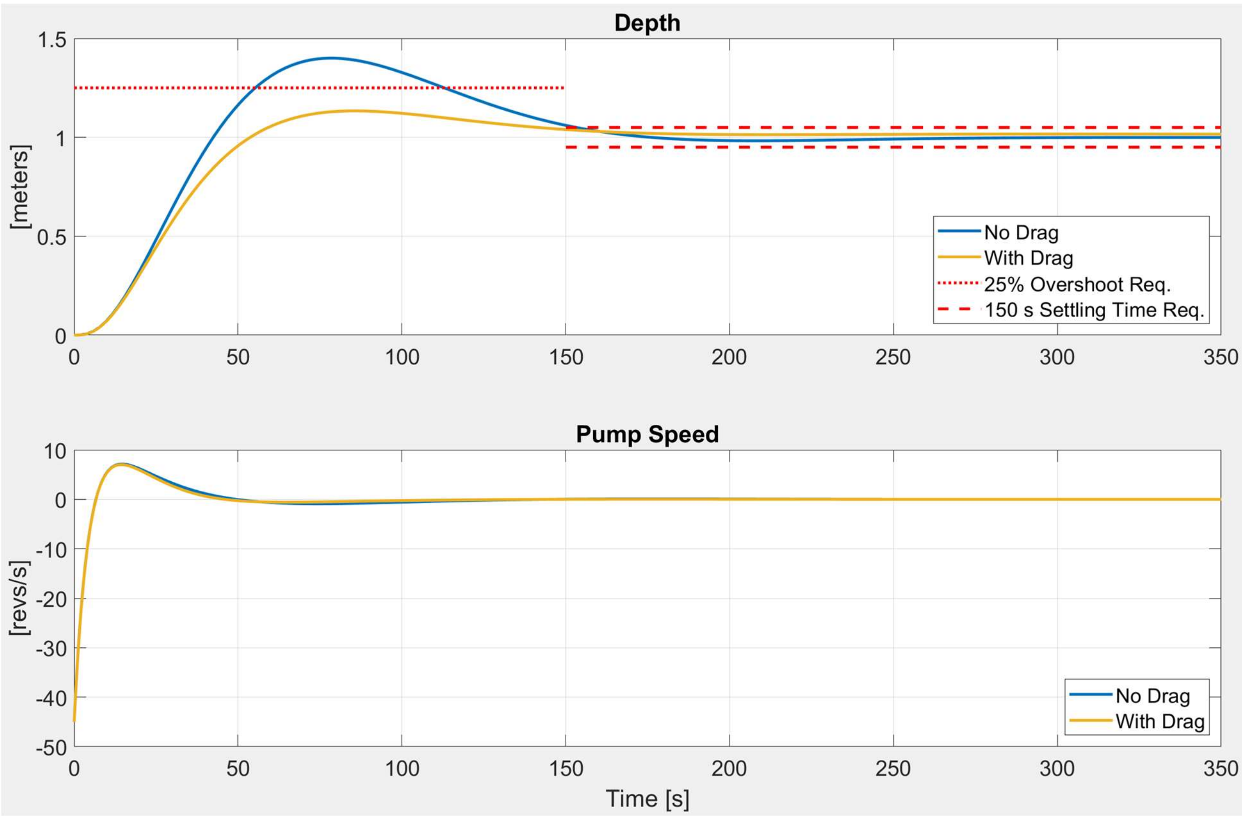


Figure 6: Comparison of Responses With and Without Drag Simulated

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:

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

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

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

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.

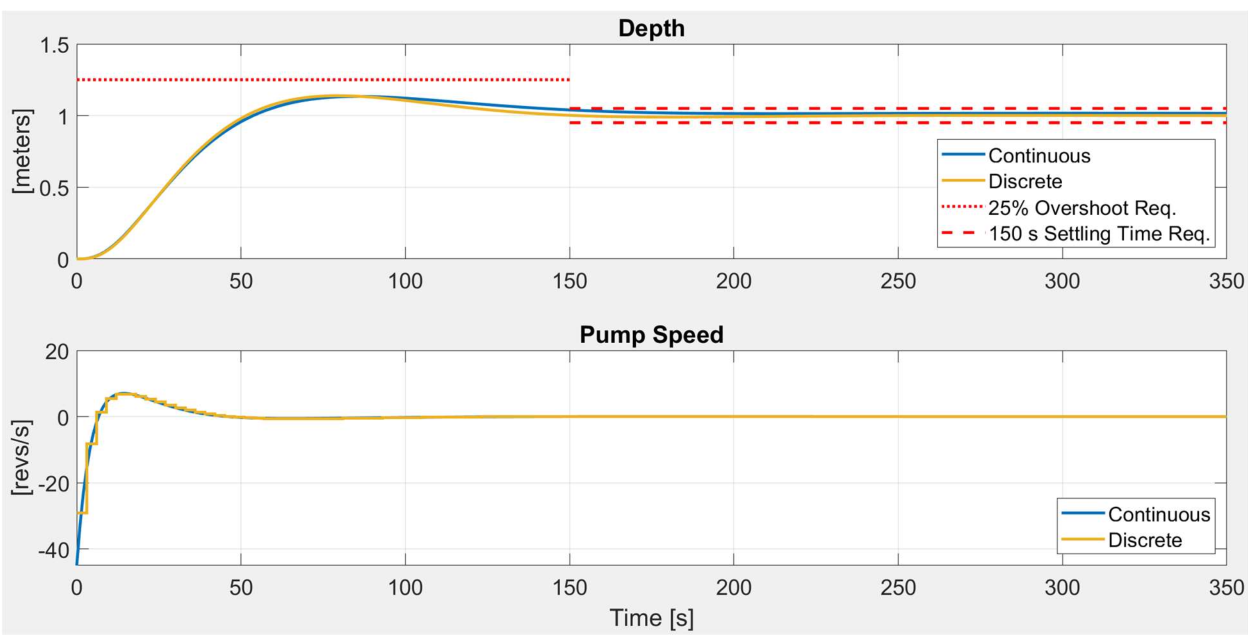


Figure 7: Continuous vs. Discrete Controller Responses

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.

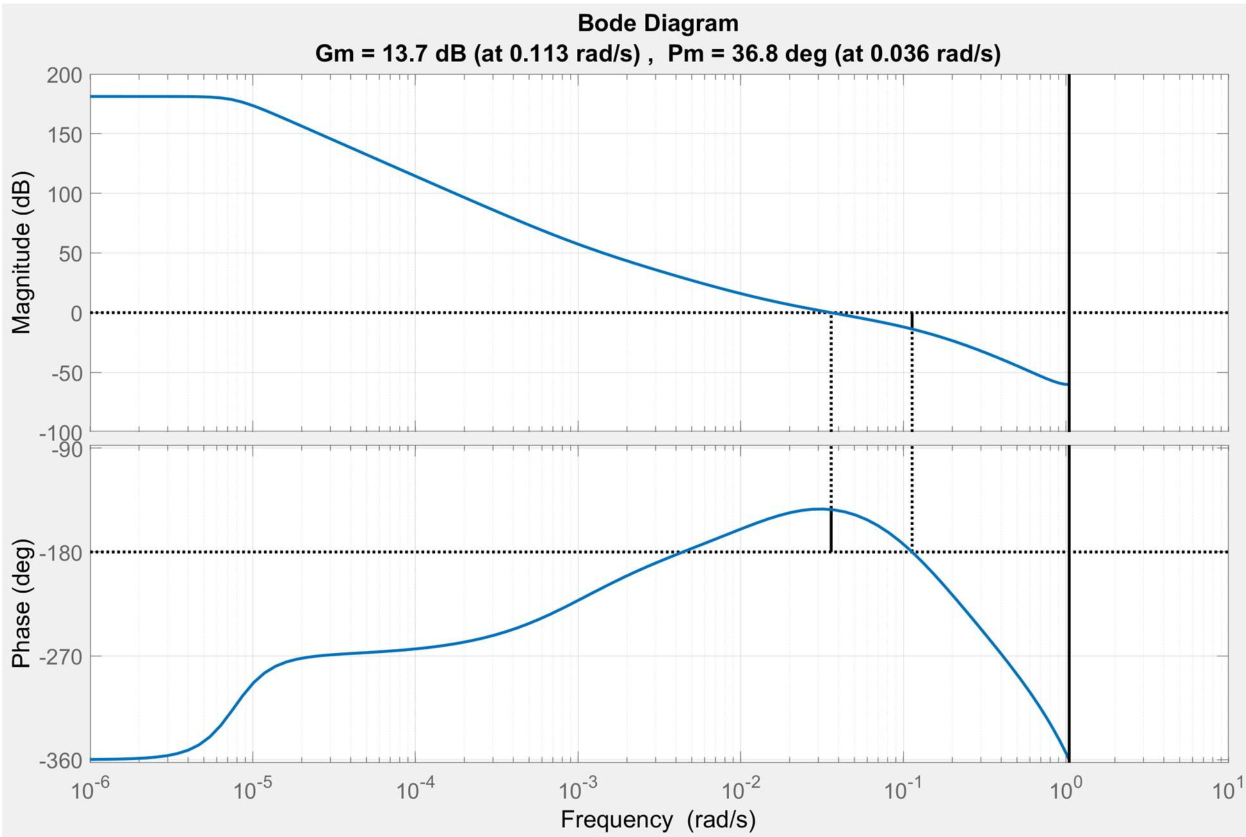


Figure 8: Bode Plot of the Discrete Open Loop Transfer Function, L(z)

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).

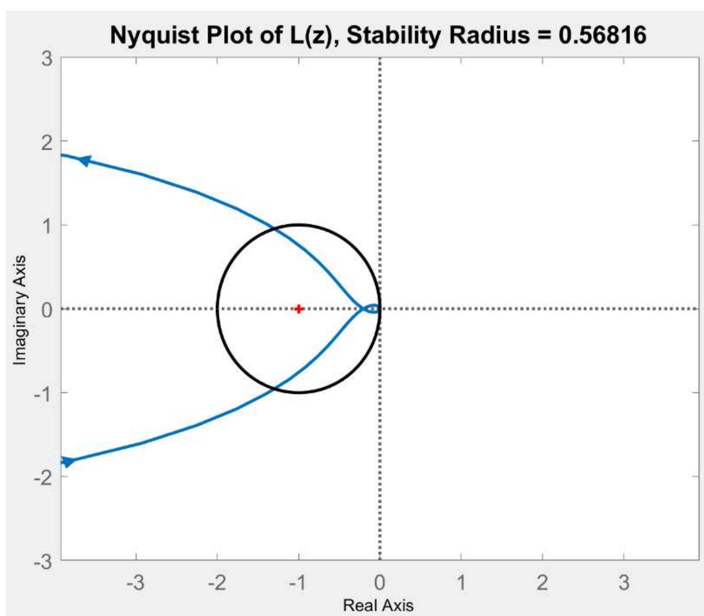


Figure 9: Nyquist Plot of L(z)

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.

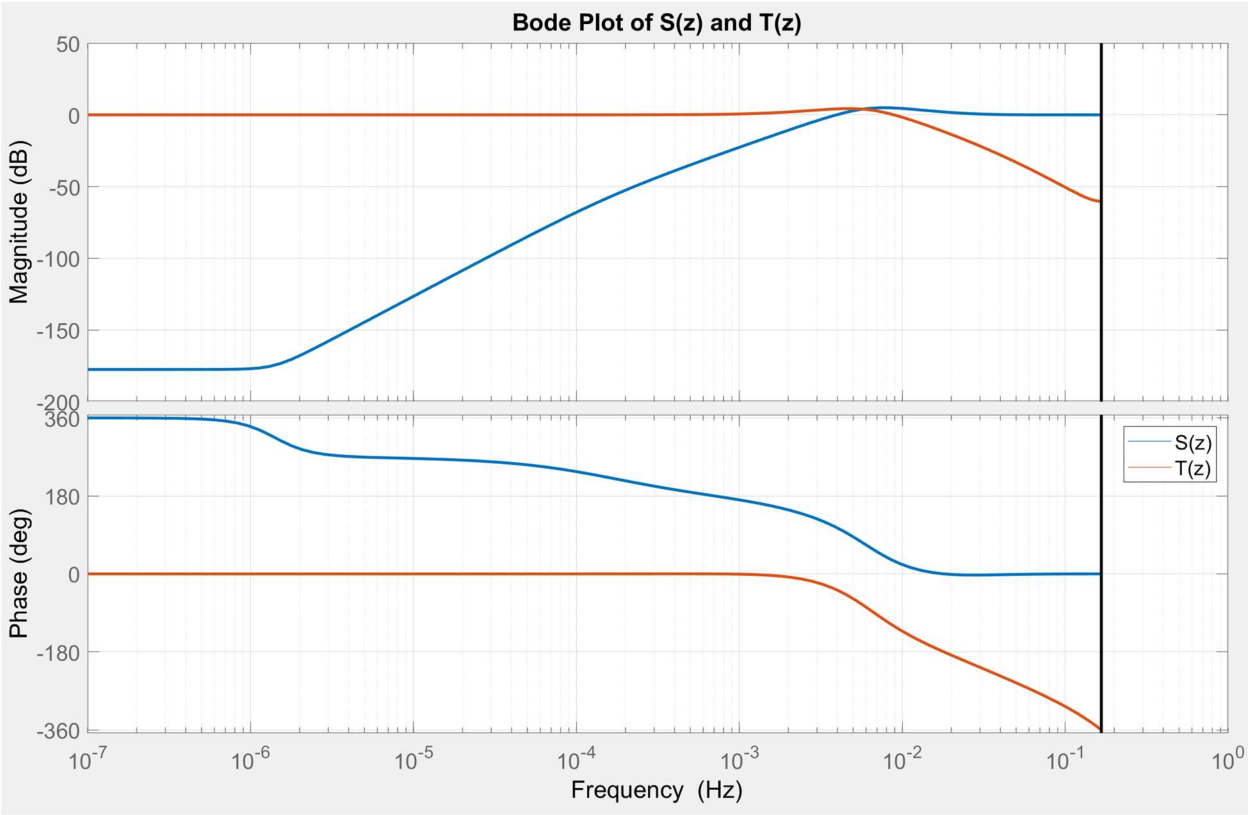


Figure 10: Bode Plots of S(z) and T(z)

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.

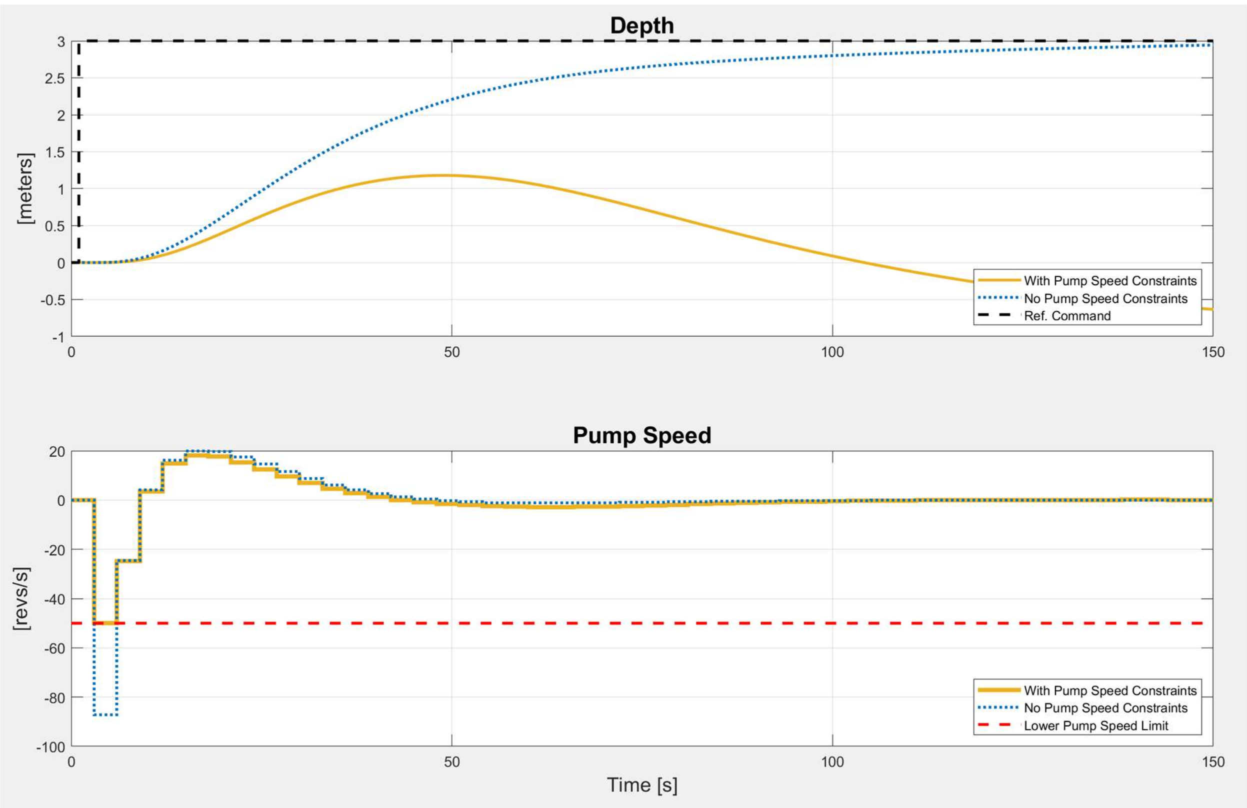


Figure 11: Comparison of Responses With and Without Pump Speed Constraints

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.

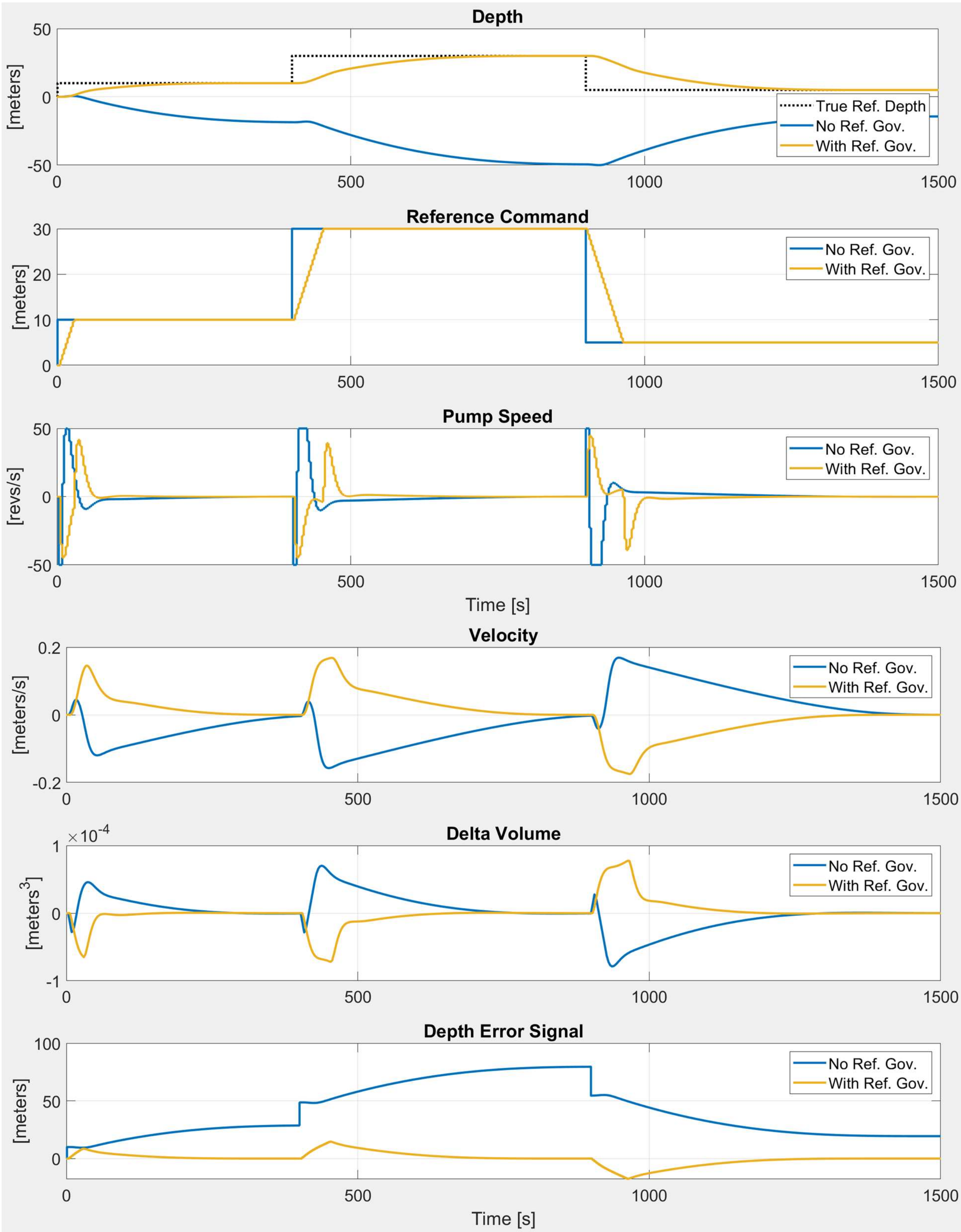


Figure 12: Comparison of Responses With and Without the Reference Governor

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 them 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.

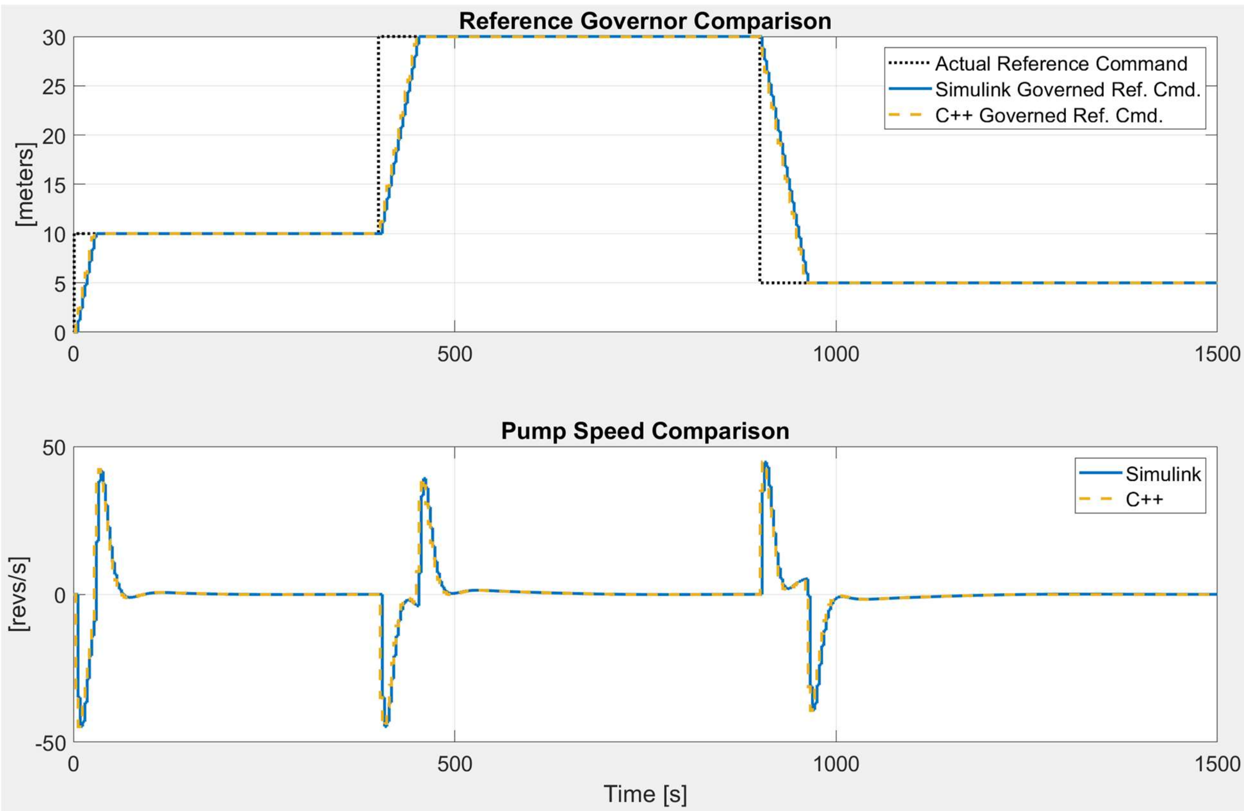


Figure 13: Simulink vs. C++ Controller Responses

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).

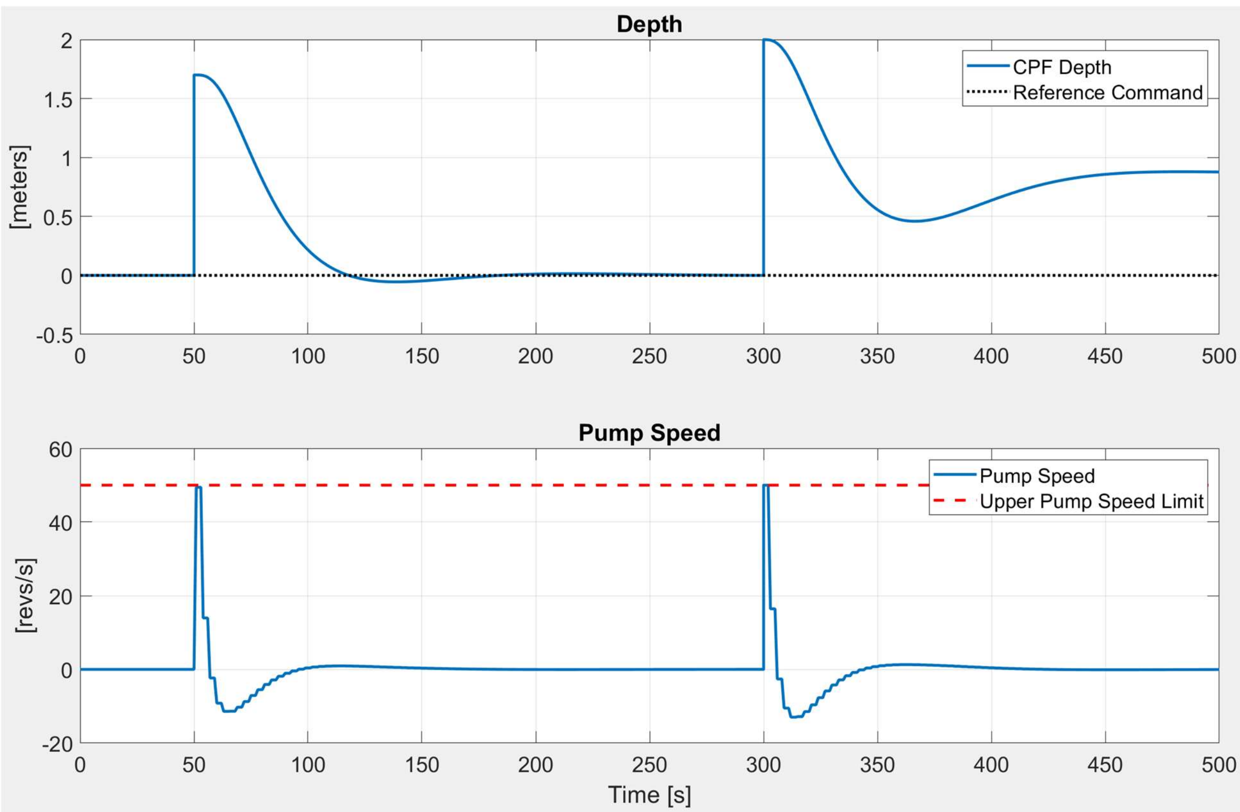


Figure 14: Simulation Showing Failure to Reject Large Depth Disturbances

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

```
% =========================================================================
% AUTHOR       : Brian Ha
% MENTOR       : Gene Massion
% DATE CREATED : 2018-07-11
% LAST REVISED : 2018-09-26
% MBARI GROUP  : Chemical Sensors Lab
% PROJECT      : Coastal Profiling Float (CPF) Depth Control
% =========================================================================


% Clean Up
close all
clear variables
clc


% Plot Formatting
yellow = [0.9290, 0.6940, 0.1250]; % Custom Plot Color Codes
axisLabelFontSize = 15;
titleFontSize = 17;
legendFontSize = 14;


% #########################################################################

% Section 1: The CPF Model

% #########################################################################


% Float Parameters
floatLength = 1.016;                        % [m] This is equal to 40 in.
floatDiameter = 0.2032;                     % [m] This is equal to 8 in.
PlatformArea = (pi/4) * floatDiameter^2;    % [m^2]. Cylinder frontal area.

PlatformMass = 38;                          % [kg]
VirtualMassFactor = 1.25;                   % Accounts for water being dragged
m = PlatformMass * VirtualMassFactor;       % [kg]

pumpDisplacement = 1.56e-7;                 % [m^3/rev] Per Oildyne spec sheet
pumpEff = 0.6;

samplingPeriod = 3;                         % [s] New pressure meas. every 3 s


% Constraints
upperPumpSpeedLimit = 50;                   % [revs/s] This is equal to 3000 rpm.
lowerPumpSpeedLimit = -50;                  % [revs/s]


```

```
% Assumptions
densitySW = 1025;                    % [kg/m^3] This is for seawater. Ignores compressibility.
g = 9.807;                           % [m/s^2] Gravitational acceleration.
velFreeStream = 0.1;                 % [m/s] This is equal to 10 cm/s.
kinematicViscosity = 13.60e-7;       % [m^2/s]. Taken from http://web.mit.edu/seawater/2017_MIT_Seawater_Property_Tables_r2a.pdf


% Estimation of drag coefficient range
Re = velFreeStream*floatDiameter/kinematicViscosity;  % Just checking Reynolds Number...

% CdMin is interpolated below from the values (based on frontal area)
% given for a flat-faced cylinder on Table 7.3 (page 491) of Fluid
% Mechanics, 7th Ed. by Frank M. White. Copyright 2011. Available at
% https://hellcareers.files.wordpress.com/2016/01/fluid-mechanics-seventh-edition-by-frank-m-white.pdf.

CdMin = (0.99 - 0.87)/(8 - 4) * (floatLength/floatDiameter - 4) + 0.87;

CdMax = 1.5; % Taken from Laughlin Barker's paper (2014 MBARI intern).

Cd = CdMin;  % User can set this to either CdMax or CdMin


% Equilibrium Point
x_eq = [0;0;0];
u_eq = 0;

% Symbolic Linearization about Equilibrium (via Taylor Series Expansion)
[Aeq, Beq] = symLin(x_eq, u_eq, densitySW, Cd, PlatformArea, g, m, pumpDisplacement, pumpEff);
C = [1 0 0];
D = 0;

disp('CPF Model')
P = ss(Aeq, Beq, C, D)

%{
% Linearization about Equilibrium by hand calculation (for verification)
A = [0 1 0; 0 densitySW*Cd*PlatformArea*x_eq(2)/m -densitySW*g/m; 0 0 0];
B = [0; 0; pumpDisplacement*pumpEff];

p = ss(A,B,C,D);
%}


% Kalman Controllability Test
disp('KALMAN CONTROLLABILITY TEST')
```

```
disp(['Rank of Controllability Matrix = ', num2str(rank(ctrb(P)))])
disp('If rank = 3, system is fully controllable.')
disp(' ')


% Kalman Observability Test
disp('KALMAN OBSERVABILITY TEST')
disp(['Rank of Observability Matrix = ', num2str(rank(obsv(P)))])
disp('If rank = 3, system is fully observable.')
disp(' ')


% ##########################################################################

% Section 2: The Controller

% ##########################################################################


% The Dual Lead Compensator
s = tf('s');
disp('Continuous Controller, C(s) = ')
C = -45*(s+0.015)*(s+0.001)/((s+0.17)*(s+0.15));
zpk(C)


% Important Transfer Functions
L = tf(P)*C;             % Open-Loop Transfer Function
S = minreal(1/(1 + L));  % Plant Output Disturbance Sensitivity Function
T = minreal(1 - S);      % Command and Noise Sensitivity Function
CS = minreal(C*S);       % Control Signal Sensitivity Function
SP = minreal(S*P);       % Plant Input Disturbance Sensivity Function




% ==============================
% LTI CONTINUOUS-TIME SIMULATION
% ==============================

% Unit Step Responses of the Four Closed Loop Transfer Functions
referenceCommandSize = 1;     % [m]

figure(1)

opt = stepDataOptions('StepAmplitude', referenceCommandSize');

subplot(2,2,1)
step(T, opt)
[y, t] = step(T, opt);
```

```
hold on
plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])), 'r:')
plot([150:5:t(end)], 1.05*referenceCommandSize*ones(1, length([150:5:t(end)])), 'r--')
plot([150:5:t(end)], 0.95*referenceCommandSize*ones(1, length([150:5:t(end)])), 'r--')
grid on
title('Command and Noise Response, T(s)', 'FontSize', titleFontSize)
xlabel('Time', 'FontSize', axisLabelFontSize)
ylabel('Depth [m]', 'FontSize', axisLabelFontSize)


subplot(2,2,2)
step(CS, opt)
[y, t] = step(C*S);
hold on
plot(t, upperPumpSpeedLimit*ones(1, length(t)), 'r--')
plot(t, lowerPumpSpeedLimit*ones(1, length(t)), 'r--')
grid on
title('Control Signal Response, CS(s)', 'FontSize', titleFontSize)
xlabel('Time', 'FontSize', axisLabelFontSize)
ylabel('Pump Speed [revs/s]', 'FontSize', axisLabelFontSize)


subplot(2,2,3)
step(S, opt)
grid on
title('Output Disturbance Rejection, S(s)', 'FontSize', titleFontSize)
xlabel('Time', 'FontSize', axisLabelFontSize)
ylabel('Depth [m]', 'FontSize', axisLabelFontSize)


subplot(2,2,4)
step(SP, opt)
grid on
title('Input Disturbance Rejection, SP(s)', 'FontSize', titleFontSize)
xlabel('Time', 'FontSize', axisLabelFontSize)
ylabel('Depth [m]', 'FontSize', axisLabelFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)




% ========================================
% LTI vs. NLTI, CONTINUOUS-TIME COMPARISON
% ========================================

referenceCommandSize = 1;       % [m]
```

```
referenceCommandTime = 0;        % [s]
outputDisturbanceSize = 0;       % [m]
outputDisturbanceTime = 0;       % [s]
tFinal = 350;                    % [s]


[depth_lti, time_lti] = step(T, tFinal);
[time_c, ~, ~, depth_c, ~, ctrl_c, ~] = sim('cpf_model_with_continuous_controller');

figure(2)
subplot(2,1,1)
plot(time_lti, depth_lti)
hold on
plot(time_c, depth_c, 'color', yellow)
plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])), 'r:')
plot(150:5:time_c(end), 1.05*referenceCommandSize*ones(1, length(150:5:time_c(end))), 'r--')
plot(150:5:time_c(end), 0.95*referenceCommandSize*ones(1, length(150:5:time_c(end))), 'r--')
grid on
legend({'No Drag', 'With Drag', '25% Overshoot Req.', '150 s Settling Time Req.'}, 'location', 'southeast', 'FontSize', legendFontSize)
title('Depth', 'FontSize', titleFontSize)
ylabel('[meters]', 'FontSize', axisLabelFontSize)


[ctrl_lti, time_lti] = step(CS, tFinal);
subplot(2,1,2)
plot(time_lti, ctrl_lti);
hold on
plot(time_c, ctrl_c, 'color', yellow)
grid on
legend({'No Drag', 'With Drag'}, 'location', 'southeast', 'FontSize', legendFontSize)
title('Pump Speed', 'FontSize', titleFontSize)
ylabel('[revs/s]', 'FontSize', axisLabelFontSize)
xlabel('Time [s]', 'FontSize', axisLabelFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)



% ==================================================
% NLTI, CONTINUOUS-TIME vs. DISCRETE-TIME COMPARISON
% ==================================================

% Discretization
disp('Discrete Controller, C(z) = ')
```

```
Cz = c2d(C, samplingPeriod, 'matched')
Pz = c2d(P, samplingPeriod, 'zoh');
Lz = tf(Pz)*Cz;             % Open-Loop Transfer Function
Sz = minreal(1/(1 + Lz));   % Plant Output Disturbance Sensitivity Function
Tz = minreal(1 - Sz);       % Command and Noise Sensitivity Function
CSz = minreal(Cz*Sz);       % Control Signal Sensitivity Function
SPz = minreal(Sz*Pz);       % Plant Input Disturbance Sensivity Function


% Simulation
referenceCommandSize2 = 0;       % [m]
referenceCommandTime2 = 0;       % [s]
referenceCommandSize3 = 0;       % [m]
referenceCommandTime3 = 0;       % [s]
[time_d, ~, ~, depth_d, ~, ctrl_d, ~, ~] = sim('cpf_model_with_discrete_controller');


figure(3)
subplot(2,1,1)
plot(time_c, depth_c);
hold on
plot(time_d, depth_d, 'color', yellow)
plot(0:5:150, 1.25*referenceCommandSize*ones(1, length([0:5:150])),  'r:')
plot(150:5:time_c(end), 1.05*referenceCommandSize*ones(1, length(150:5:time_c(end))), 'r--')
plot(150:5:time_c(end), 0.95*referenceCommandSize*ones(1, length(150:5:time_c(end))), 'r--')
legend({'Continuous', 'Discrete', '25% Overshoot Req.', '150 s Settling Time Req.'}, 'location', 'southeast', 'FontSize', legendFontSize)
grid on
ylabel('[meters]', 'FontSize', axisLabelFontSize)
title('Depth', 'FontSize', titleFontSize)


subplot(2,1,2)
plot(time_c, ctrl_c);
hold on
stairs(time_d, ctrl_d, 'linewidth', 2, 'color', yellow)
legend({'Continuous', 'Discrete'}, 'location', 'southeast', 'FontSize', legendFontSize)
grid on
title('Pump Speed', 'FontSize', titleFontSize)
xlabel('Time [s]', 'FontSize', axisLabelFontSize)
ylabel('[revs/s]', 'FontSize', axisLabelFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)

% Bode Plot of the Open Loop Transfer Function, L(z)
```

```matlab
figure(4)
margin(Lz)
grid on
set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)


% Nyquist Plot of the Open Loop Transfer Function, L(z)
figure(5)
nyquist(Lz)

% Draw a black unit circle
circle = -1+ (-s+1)/(s+1);
omegac = logspace(-3,3,400);
[reC,imC] = nyquist(circle,omegac);
reC = squeeze(reC);
imC = squeeze(imC);
hold on
plot(reC,imC,'k-',reC,-imC,'k-')

% calculate stability radius as the radius of the circle
% centered at the critical point that is avoided by the Nyquist plot
[mag,~] = bode(Sz);
mag = squeeze(mag);
stab_rad = 1/max(mag);

axis([-3 3 -3 3])
axis('equal')
set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
title(['Nyquist Plot of L(z), Stability Radius = ' , num2str(stab_rad)], 'FontSize', titleFontSize)


% Bode Plot of S(z) and T(z)
figure(6)
options = bodeoptions('cstprefs');
options.FreqUnits = 'Hz';
options.Xlabel.FontSize = axisLabelFontSize;
options.Ylabel.FontSize = axisLabelFontSize;
%options.MagUnits = 'abs';
bode(Sz, options)
hold on
bode(Tz, options)
grid on
title('Bode Plot of S(z) and T(z)', 'FontSize', titleFontSize)
legend({'S(z)', 'T(z)'}, 'FontSize', legendFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
```

```
% ##########################################################################

% Section 3: The Reference Governor

% ##########################################################################

% Simulation Demonstrating Ability to Track Large Reference Commands
startingDepth = 0;                % [m]
referenceCommandSize = 10;        % [m]
referenceCommandTime = 1;         % [s]
referenceCommandSize2 = 20;       % [m]
referenceCommandTime2 = 400;      % [s]
referenceCommandSize3 = -25;      % [m]
referenceCommandTime3 = 900;      % [s]
outputDisturbanceSize = 0;        % [m]
outputDisturbanceTime = 0;
outputDisturbanceSize2 = 0;       % [m]
outputDisturbanceTime2 = 0;
tFinal = 1500;                    % [s]
maxSetPointIncrement = 0.40;      % [m]
maxSetPointDecrement = -1*maxSetPointIncrement;  % [m]


[time_d, ~, ref_d, depth_d, vel_d, ctrl_d, delta_vol_d, error_d] = sim('cpf_model_with_discrete_controller');
[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');


figure(7)
subplot(3,1,1)
plot(time_d, ref_d, 'k:')
hold on
plot(time_d, depth_d)
plot(time_gov, depth_gov, 'color', yellow)
legend({'True Ref. Depth', 'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'southeast', 'FontSize', legendFontSize)
grid on
title('Depth', 'FontSize', titleFontSize)
ylabel('[meters]', 'FontSize', axisLabelFontSize)


subplot(3,1,2)
plot(time_d, ref_d)
hold on
stairs(time_gov, ref_gov, 'color', yellow, 'linewidth', 2)
legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
grid on
```

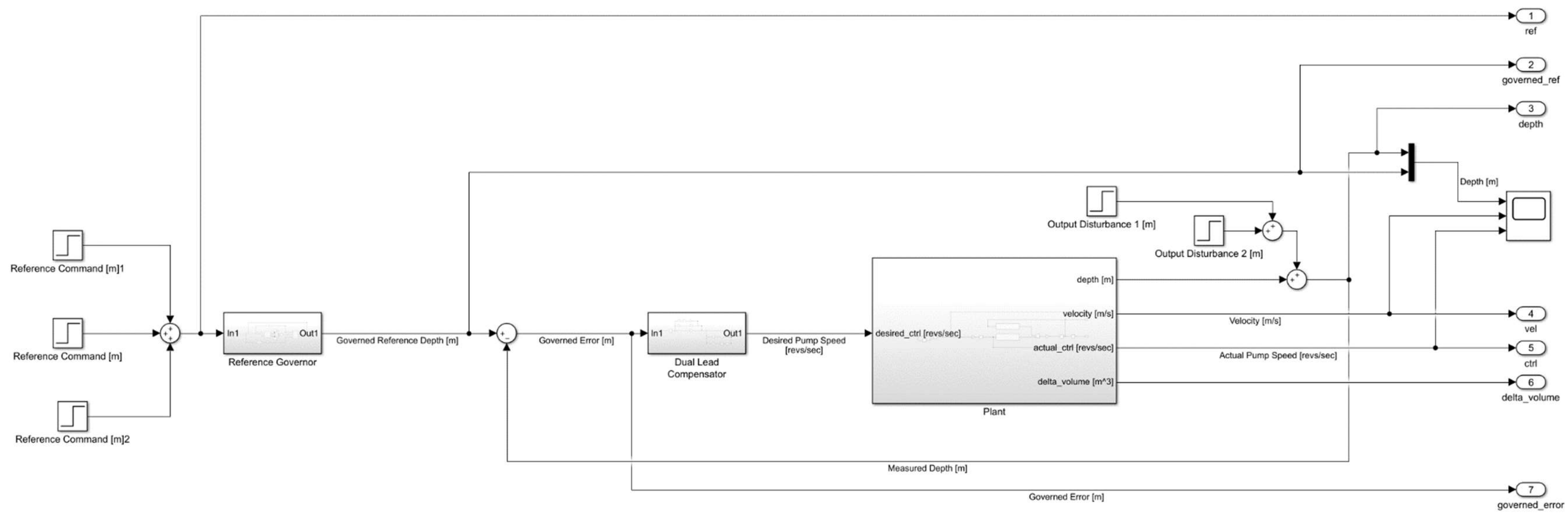
```
title('Reference Command', 'FontSize', titleFontSize)
ylabel('[meters]', 'FontSize', axisLabelFontSize)


subplot(3,1,3)
stairs(time_d, ctrl_d, 'linewidth', 2)
hold on
stairs(time_gov, ctrl_gov, 'color', yellow, 'linewidth', 2)
legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
grid on
title('Pump Speed', 'FontSize', titleFontSize)
xlabel('Time [s]', 'FontSize', axisLabelFontSize)
ylabel('[revs/s]', 'FontSize', axisLabelFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)


figure(8)

subplot(3,1,1)
plot(time_d, vel_d)
hold on
plot(time_gov, vel_gov, 'color', yellow)
legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
grid on
ylabel('[meters/s]', 'FontSize', axisLabelFontSize)
title('Velocity', 'FontSize', titleFontSize)


subplot(3,1,2)
plot(time_d, delta_vol_d)
hold on
plot(time_gov, delta_vol_gov, 'color', yellow)
grid on
ylabel('[meters^3]', 'FontSize', axisLabelFontSize)
legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
title('Delta Volume', 'FontSize', titleFontSize)


subplot(3,1,3)
plot(time_d, error_d)
hold on
plot(time_gov, governed_error, 'color', yellow)
grid on
xlabel('Time [s]', 'FontSize', axisLabelFontSize)
ylabel('[meters]', 'FontSize', axisLabelFontSize)
```

```
legend({'No Ref. Gov.', 'With Ref. Gov.'}, 'location', 'northeast', 'FontSize', legendFontSize)
title('Depth Error Signal', 'FontSize', titleFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)


% Simulation Showing Failure to Reject Large Depth Disturbances
referenceCommandSize = 0;       % [m]
referenceCommandTime = 0;       % [s]
referenceCommandSize2 = 0;      % [m]
referenceCommandTime2 = 0;      % [s]
referenceCommandSize3 = 0;      % [m]
referenceCommandTime3 = 0;      % [s]
outputDisturbanceSize = 1.7;    % [m]
outputDisturbanceTime = 50;     % [s]
outputDisturbanceSize2 = 2;     % [m]
outputDisturbanceTime2 = 300;   % [s]
tFinal = 500;                   % [s]

[time_gov, ~, ref, ~, depth_gov, ~, ctrl_gov, ~, ~] = sim('cpf_model_with_discrete_controller_and_ref_gov');

figure(9)
subplot(2,1,1)
plot(time_gov, depth_gov)
hold on
plot(time_gov, ref, 'k:')
grid on
title('Depth', 'FontSize', titleFontSize)
ylabel('[meters]', 'FontSize', axisLabelFontSize)
legend({'CPF Depth', 'Reference Command'}, 'FontSize', legendFontSize)

subplot(2,1,2)
plot(time_gov, ctrl_gov)
hold on
plot(time_gov, upperPumpSpeedLimit*ones(1, length(time_gov)),  'r--')
grid on
legend({'Pump Speed', 'Upper Pump Speed Limit'}, 'FontSize', legendFontSize)
title('Pump Speed', 'FontSize', titleFontSize)
xlabel('Time [s]', 'FontSize', axisLabelFontSize)
ylabel('[revs/s]', 'FontSize', axisLabelFontSize)

set(findall(gcf, 'type', 'line'), 'linewidth', 2)
set(findall(gcf, 'type', 'axes'), 'FontSize', axisLabelFontSize)
```

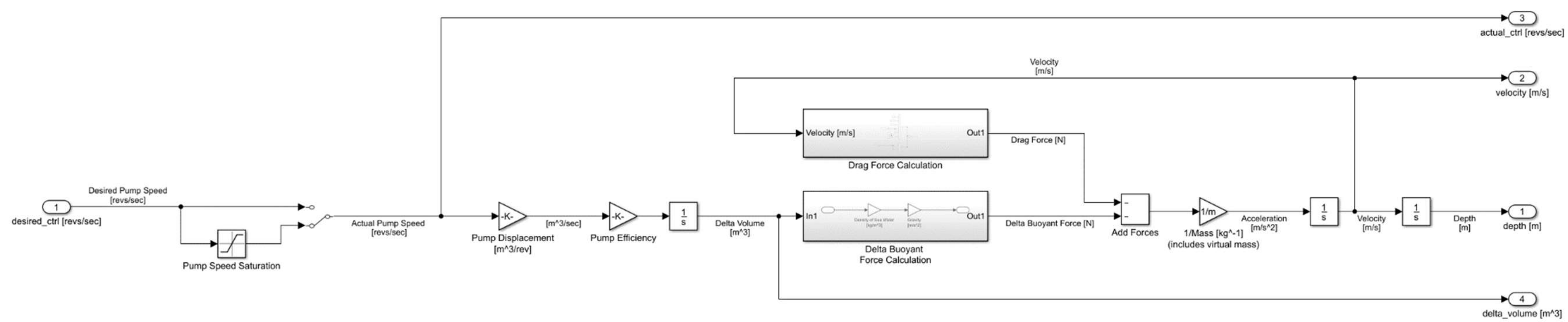
```
%% Functions
function [A, B] = symLin(x_eq, u_eq, densitySW, Cd, area, grav, mass, pumpDisplacement, pumpEff)
% Calculates the Jacobian A and B matrices about x_eq and u_eq

% Part I of II: Symbolic Calculation of General Jacobian

% Symbolic State Vector, x
position = sym('position');
velocity = sym('velocity');
delta_volume = sym('delta_volume');

x = [position; velocity; delta_volume];

% Symbolic Input Vector, u
motor_speed = sym('motor_speed'); % [rev/s]

u = [motor_speed];

% Non-Linear ODEs
x_dot(1,1) = velocity;
x_dot(2,1) = (-0.5*densitySW*area*Cd*velocity^2 - densitySW*grav*delta_volume)/mass;
x_dot(3,1) = motor_speed*pumpDisplacement*pumpEff;

A = jacobian(x_dot, x);
B = jacobian(x_dot, u);

% Part II of II: Calculation of Jacobian at x_eq and u_eq
position = x_eq(1);
velocity = x_eq(2);
delta_volume = x_eq(3);

motor_speed = u_eq;

A = eval(subs(A));
B = eval(subs(B));

end
```

Appendix A.2: Simulink Models

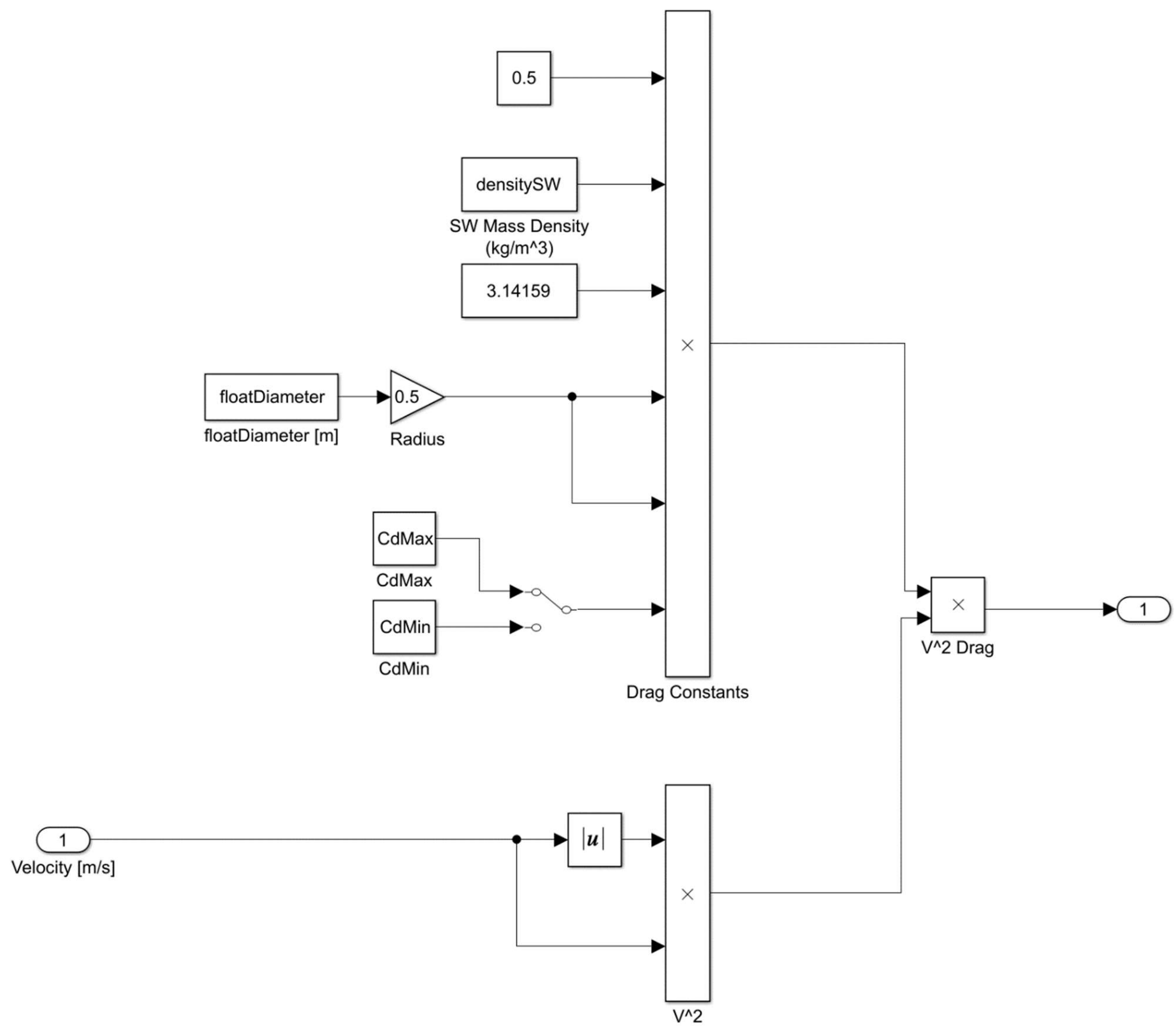
High Level Block Diagram



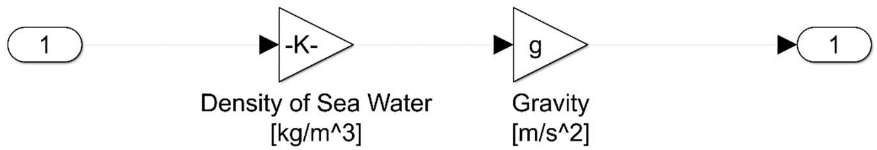
High Level Block Diagram → Plant



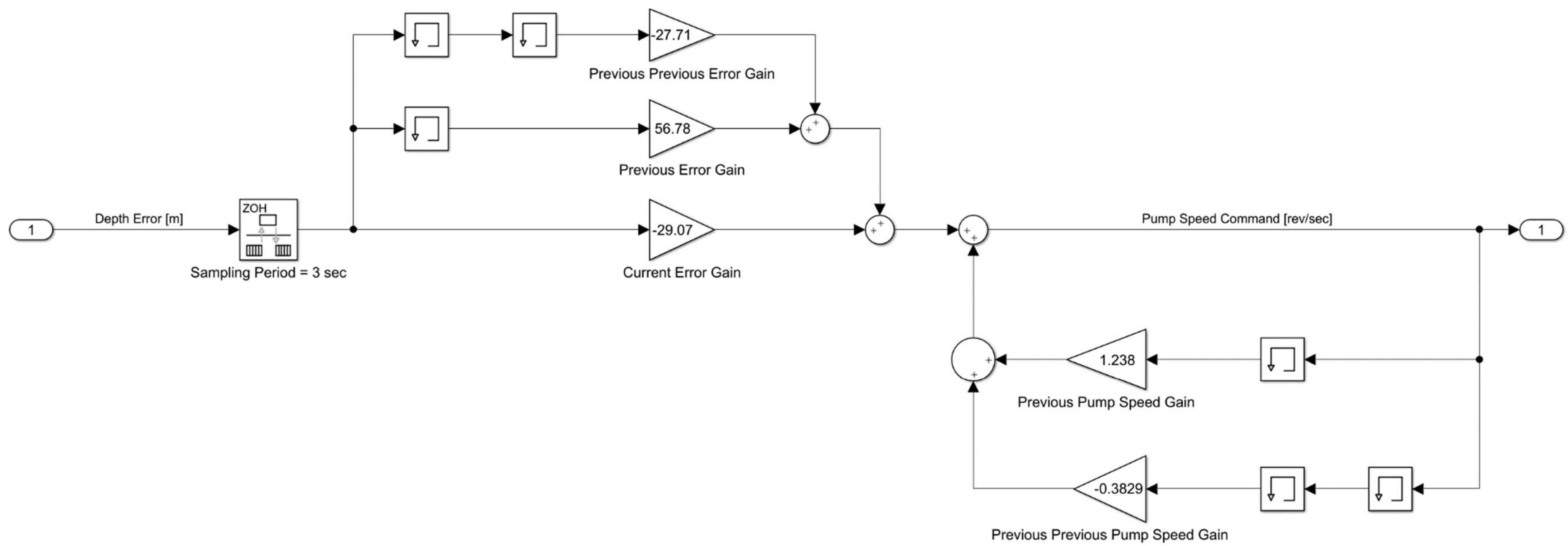
High Level Block Diagram → Plant → Drag Force Calculation



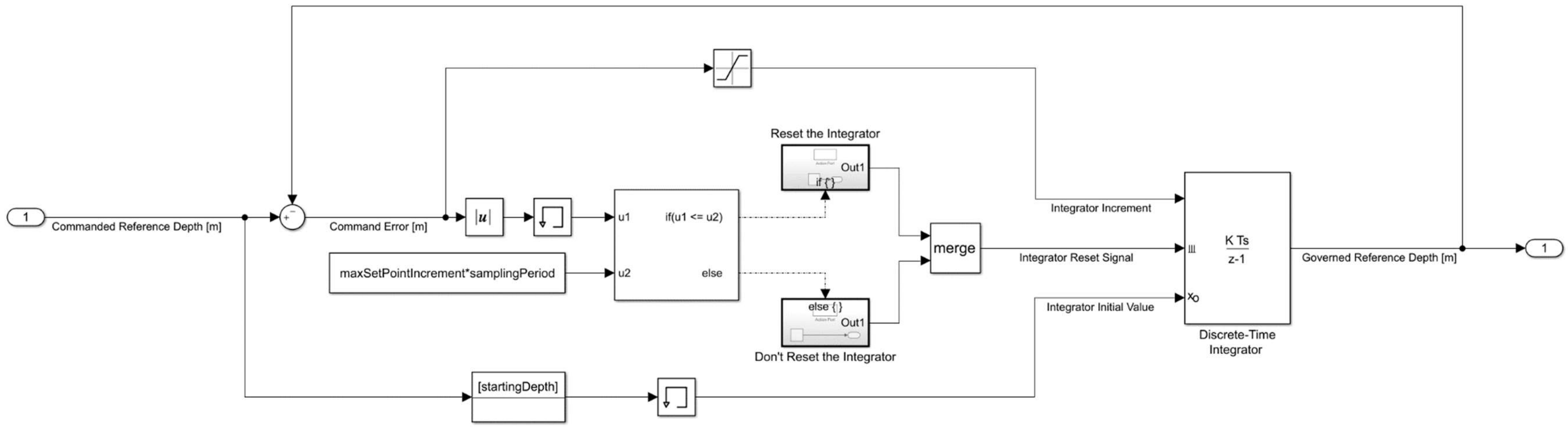
High Level Block Diagram → Plant → Delta Buoyant Force Calculation



High Level Block Diagram → Dual Lead Compensator



High Level Block Diagram → Reference Governor



```cpp
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;

// Dual Lead Compensator Values
const double currentErrorGain = -29.07;
const double prevErrorGain = 56.78;
const double prevPrevErrorGain = -27.71;
const double prevPumpSpeedGain = 1.238;
const double prevPrevPumpSpeedGain = -0.3829;

const double pumpSpeedUpperLimit = 50.0;      // [revs/s]
const double pumpSpeedLowerLimit = -50.0;     // [revs/s]

static double currentPressureError = 0.0;
static double prevPressureError = 0.0;
static double prevPrevPressureError = 0.0;

static double nextPumpSpeedCmd = 0.0;
static double prevPumpSpeedCmd = 0.0;
static double prevPrevPumpSpeedCmd = 0.0;


// Reference Governor Values
const double maxSetPointIncrement = 1.2;    // [decibar]
const double maxSetPointDecrement = -1.2;   // [decibar]
static double integrator; // [decibar]


void InitializeGovernor(double startingPressure)
{
    integrator = startingPressure;
}


double ReferenceGovernor(double desiredPressureSetPoint)
{
    // AUTHOR: Brian Ha (2018 Summer Intern)

    double setPointDelta;    // [decibar]

    cout << "TARGET PRESSURE = " << desiredPressureSetPoint << " decibar;   ";


    // If the integrator value is within 1.2 decibar of the target pressure,
    // set it equal to the target pressure.
    if (abs(desiredPressureSetPoint - integrator) <= maxSetPointIncrement)
    {
```

```cpp
        integrator = desiredPressureSetPoint;
    }
    // If not, increment/decrement the integrator
    else
    {
        setPointDelta = desiredPressureSetPoint - integrator;

        // Limit how quickly the integrator can increment/decrement.
        if (setPointDelta > maxSetPointIncrement)
        {
            setPointDelta = maxSetPointIncrement;
        }
        else if (setPointDelta < maxSetPointDecrement)
        {
            setPointDelta = maxSetPointDecrement;
        }

        integrator = integrator + setPointDelta;
    }

    cout << "INTEGRATOR = " << integrator << " decibar;   ";

    return integrator;
}


double DepthController(double governedPressureSetPoint, double currentPlatformPressure)
{
    // AUTHOR: Brian Ha (2018 Summer Intern)

    // Calculate current pressure error
    currentPressureError = governedPressureSetPoint - currentPlatformPressure;
    cout << "PRESSURE ERROR = " << currentPressureError << " decibar;   ";


    // Calculate the next pump speed command.
    nextPumpSpeedCmd = ((currentErrorGain * currentPressureError) +
            (prevErrorGain * prevPressureError) +
            (prevPrevErrorGain * prevPrevPressureError) +
            (prevPumpSpeedGain * prevPumpSpeedCmd) +
            (prevPrevPumpSpeedGain * prevPrevPumpSpeedCmd));


    // Limit pump speed command magnitude to 50 revs/sec.
    if (nextPumpSpeedCmd > pumpSpeedUpperLimit)
    {
        nextPumpSpeedCmd = pumpSpeedUpperLimit;
    }
    else if(nextPumpSpeedCmd < pumpSpeedLowerLimit)
```

```cpp
    {
        nextPumpSpeedCmd = pumpSpeedLowerLimit;
    }

    // Deadband
    if (nextPumpSpeedCmd < 0.05 && nextPumpSpeedCmd > -0.05)
    {
        nextPumpSpeedCmd = 0.0;
    }

    // Set historical values for next function call
    prevPrevPumpSpeedCmd = prevPumpSpeedCmd;
    prevPrevPressureError = prevPressureError;
    prevPumpSpeedCmd = nextPumpSpeedCmd;
    prevPressureError = currentPressureError;


    // Convert from rev/sec to counts/sec
    //nextPumpSpeedCmd = nextPumpSpeedCmd * COUNTSperREV;


    // Return next pump speed command [counts/sec].
    cout << "PUMP SPEED COMMAND: " << nextPumpSpeedCmd << " [revs/s]" << endl;
    return (nextPumpSpeedCmd);
}



int main() {
    double desiredPressureSetPoint;
    double currentPlatformPressure;
    double governedPressureSetPoint; // [decibar]
    double startingPressure = 0.0;    // [decibar]

    // Setup File I/O
    ifstream measuredPressures("pressures.txt");
    ifstream targetPressures("target_pressures.txt");
    ofstream pumpSpeeds("pump_speeds.txt");
    ofstream governedTargetPressures("governed_target_pressures.txt");



    // Initialize the integrator to the pressure at the current depth
    InitializeGovernor(startingPressure);

    // Read each value in the pressures.txt file, one at a time
    while (measuredPressures >> currentPlatformPressure)
    {
        // Read each value in the target_pressures.txt file, one at a time
        targetPressures >> desiredPressureSetPoint;
```

```

        // Run the Reference Governor
        governedPressureSetPoint = ReferenceGovernor(desiredPressureSetPoint);

        // Write the governed target pressure to the governed_target_pressures file
        governedTargetPressures << governedPressureSetPoint << endl;

        // Run the Dual Lead Compensator
        nextPumpSpeedCmd = DepthController(governedPressureSetPoint, currentPlatformPressure);

        // Write the next pump speed command to the ctrl_signals.txt file.
        pumpSpeeds << nextPumpSpeedCmd << endl;

    }

    // Close the output files.
    pumpSpeeds.close();
    governedTargetPressures.close();


    return 0;
}
```