UiA University of Agder

Master

Digital Twin of 3D Motion Compensated Gangway

Use of Unity Game Engine and TwinCAT PLC Control for Hardware-in-the-Loop Simulation

HARALD SANGVIK

SUPERVISORS
Morten Hallquist Rudolfsen, MSc
Daniel Hagen, PhD

University of Agder, 2021
Faculty of Engineering and Science
Department of Engineering Sciences

Acknowledgements

First of all, I want to thank Red Rock Marine AS for allowing me to work on this exciting topic for my master thesis project and for the excellent support by providing hardware, CAD files, and office space while writing this thesis. I would also like to thank my supervisor Daniel Hagen for helping me structuring the thesis and good tips on implementing active damping.

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Harald Sangvik
Kristiansand 28.05.2021

Abstract

This thesis presents the development, implementation, and testing of a digital twin representing a 3D compensated gangway located on a service operation vessel for wind farms. The real-time simulator runs on a separate thread in the Unity game engine and interfaces with a PC-based control system, including the control algorithms. The simulations in Unity achieve a step time of 5 μs, and the communication to the control system has a latency of 10ms.

The PLC program in TwinCAT, running real-time on a separate computer, controls the hydraulic actuated gangway simulator with acceptable accuracy in 3m waves. Active damping employing pressure feedback to introduce artificial leakage was implemented to the hydraulic control system, increasing the hydraulic system's damping, hence improving the motion performance.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Motivation and background

Motion-compensated gangways increase uptime and improve operator safety for operations from a ship to a fixed installation.
Red Rock Marine AS is a company producing marine/offshore handling equipment. A digital twin will be helpful for both internal development as well as training and product demonstrations. The task defined for the master's thesis is found in appendix A.

A digital twin is a virtual representation of a physical system. It can be used to develop and optimize the control system, working cycles, and design, even before a physical system is built. The digital twin can also be used for estimating useful life and planning of maintenance of it's physical counterpart. In addition, a digital twin of the whole vessel can be used when planning the wind turbine maintenance operations by simulating upcoming weather conditions like wind and waves based on forecasts. Furthermore, a digital twin can be used to detect upcoming sea states that the gangway is designed to compensate for and then stop the operation that would be unsafe for the personnel passing on the gangway.

DNVGL-RP-A204 [2] defines six levels of a digital twin; Level 0 - standalone, through level 5 - autonomous. As the levels increase, more data must be available from both "twins", to plan and predict operation.
Level 0 is comparable to an offline simulation, where data is entered manually. At level 5, the system should be able to replace the operator and make decisions autonomously.

State of the art

There have been some research into control and simulation of a motion compensated gangway. This thesis will build further on this work, and incorporate real-time interfacing to an interactive simulator and control system.

A few papers mentions using TwinCAT with unity for simulation of an industrial system. A bachelor's thesis by Jesse Reinikka gives a good example on how to use Beckhoff's TwinCAT ADS interface in Unity. [12]

Modeling and simulation of a small gangway was done in a master's thesis by Feilong Yu at NTNU, covering system sizing, control theory and simulation using bond graph theory. [17]

Daniel Hagen covered the implementation of an active damping system on a single boom crane [7]. This builds further on the work done by Hansen and Andersen (2010) [8].

Identification and modeling of hydraulic directional control valves used in a crane was done in a dissertation by Morten Kollerup Bak. [4], modelling of flexible bodies were also covered.

Jesper Sørensen's thesis [15] investigated a novel concept for suppressing oscillations in a negatively loaded boom system.

Waurich et al. Implemented a real-time simulation using Modelica and functional-mock-up-interface (FMU) in Unity, controlled with an Arduino. [16]

Problem statement

This report will cover the design and implementation of a hardware-in-the-loop system containing a PC-based game engine application, combining both simulation and high quality 3D visualization in a standalone system.
State of the art control strategy for controlling an offshore gangway will also be investigated.

The main objectives to be realized are:

- Modeling and real-time simulation of multibody systems and hydraulic actuation systems
- Control design and implementation on real-time control system
- Development of a simulator environment visualized in real-time
- Communication between simulator and control system

Outline

The remainder of this thesis is divided into three main chapters. Chapter 2 describes the theory behind the modeling and simulation. Chapter 3 presents the research methodologies. Chapter 4 presents the results and discussion. Finally, Chapter 5 concludes the thesis by presenting the concluding remark, the contributions, and identifying possible areas for further work. Necessary background materials and details are included in the appendix A-D.

Chapter 2

Theory

2.1 Multibody System

Rigid body motion assumes there is no internal deformation in the components; that is, the local vector between two points on a body is constant.
The gangway consists of five main components, shown in figure 2.1. The dynamic simulation will only consider the king, luffing boom, and telescope as moving objects. The pedestal can be moved up and down on the tower to set the correct height relative to the wind turbine but is not used for motion compensation.

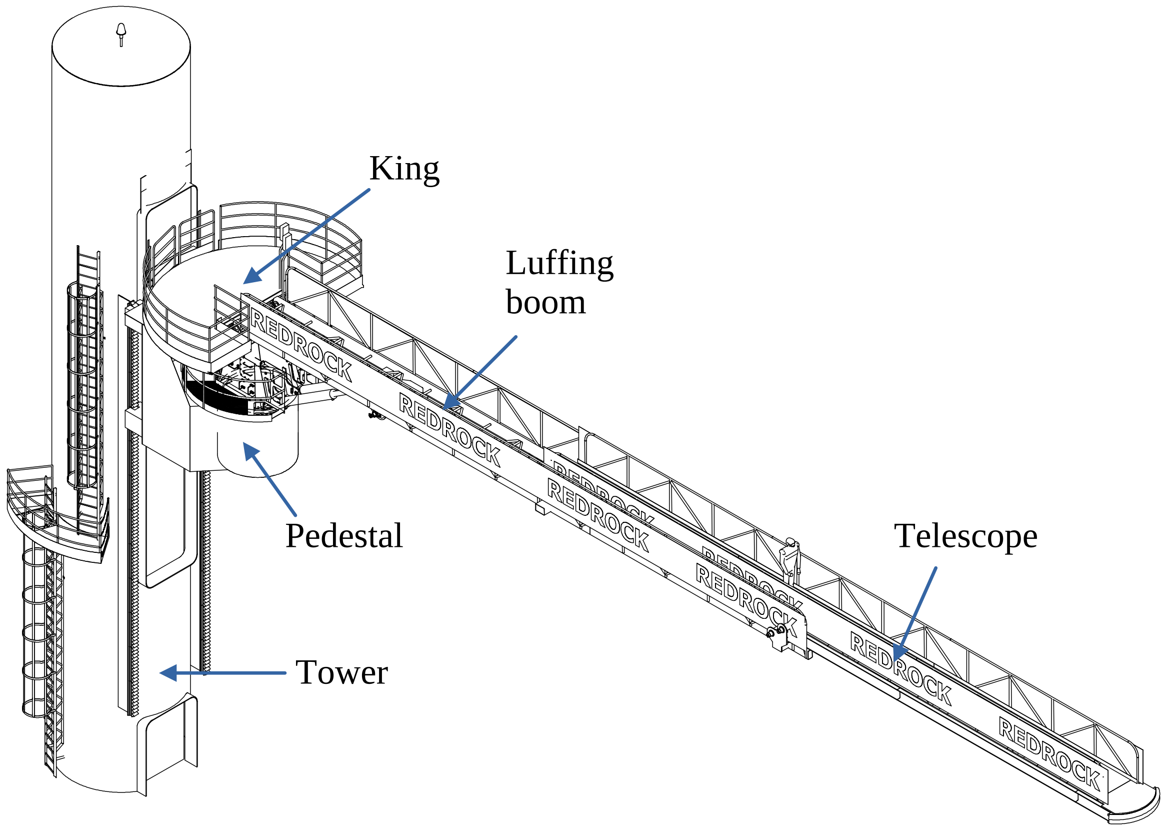


Figure 2.1: The different bodies of the gangway

2.1.1 Kinematics

Right and left handed coordinate systems are not compatible with each other. Unity and other game engines use left handed coordinate systems to simplify rendering to a 2D screen.

Handedness can be converted by inverting one axis, for translation between coordinates in Unity and the kinematics, modeled using right handed coordinates, the Z-axis is flipped.

$$[x, y, z]_r \Rightarrow [x, y, -z]_l \tag{2.1}$$

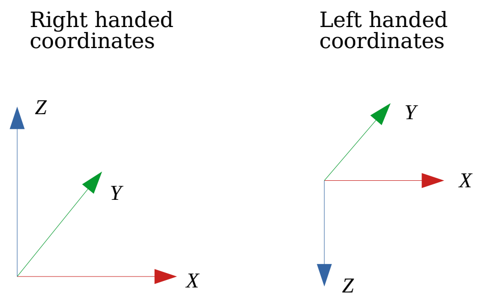


Figure 2.2: Right vs left handed coordinate system

Cylindrical coordinates

Cylindrical coordinates is a useful intermediate coordinate system for controlling the gangway. In manual mode, the joints are controlled directly but the motion compensation is using coordinates in a cartesian space. By controlling coordinates in a cylindrical coordinate system, the manual behaviour can be emulated when controlling the gangway in cartesian space.

Conversion from cartesian (ρ, ϕ, z) to cylindrical coordinates (x, y, z),

$$\rho = \sqrt{x^2 + y^2}, \tag{2.2}$$

$$\phi = atan2(y, x), \tag{2.3}$$

$$z = z, \tag{2.4}$$

Conversion from cylindrical- to cartesian coordinates:

$$x = \rho\ cos(\phi), \tag{2.5}$$

$$y = \rho\ sin(\phi), \tag{2.6}$$

$$z = z \tag{2.7}$$

Transformation matrices

Homogeneous coordinates introduces an extra "virtual" dimension, allowing for translation of a vector using a linear transformation. The extra dimension can be used to scale the vector, but is usually set to 1. The homogenous vector H can be defined as,

$$H = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{2.8}$$

H can be translated in X-Y-Z by multiplying the vector with a 4x4 matrix,

$$T(x,y,z) = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Rotating H around the X-axis is done using a 4x4 matrix,

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Same for rotations around Y,

$$R_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

And for rotation around Z,

$$R_z(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

Ship coordinates

The axis convention of the ship is taken from "Handbook of Marine Craft Motion Dynamics and Control" [6], defining surge-sway-heave and roll-pitch-yaw, as translations and rotations around x-y-z in a right handed coordinate system (figure 2.3)

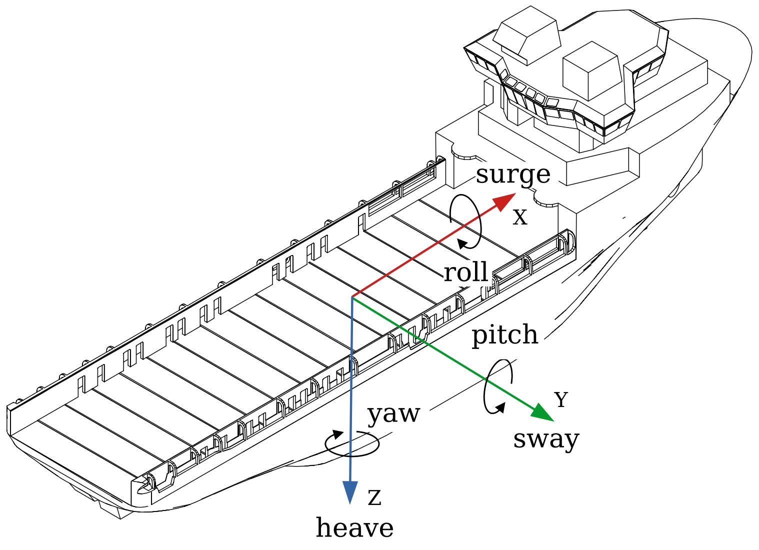


Figure 2.3: Ship coordinate system

Reference frames

Transformation between two reference frames is done using homogeneous transformations (figure 2.4). The homogenous transformation can be found by multiplying a translation and a rotation matrix together. This is done for finding the kinematics.

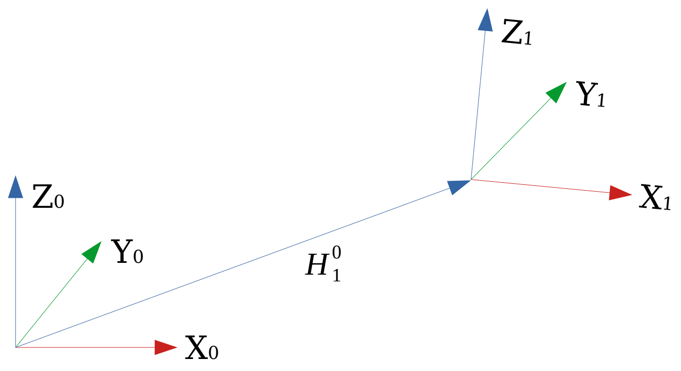


Figure 2.4: Transformation between two reference frames

Unity handles all rotations using quaternions, and gives easy access to the quaternions of the bodies in a scene. The equivalent to a homogenous transformation will be a translation followed by a quaternion rotation.
A vector **v** is rotated by a quaternion q through quaternion multiplication:

$$\mathbf{v}' = q\mathbf{v}q^{-1} \tag{2.13}$$

The initial position, with all displacements taken as zero is with the boom in a horizontal position, and the telescope fully retracted, and the boom parallel with the x-axis. (Figure 2.5). The kinematic diagram is shown in figure 2.6, with the kinematic lengths defined in table 2.1.

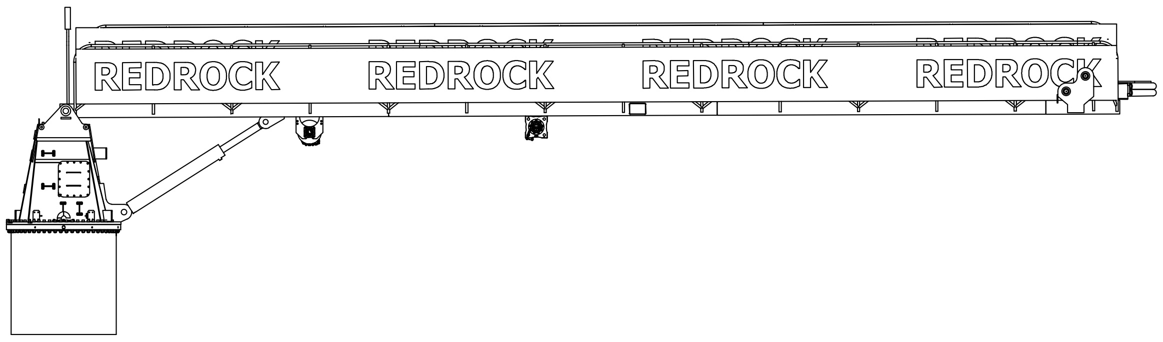


Figure 2.5: Gangway shown in parked position

Table 2.1: Dimensions of gangway

l_1	l_2	l_3	l_4
2187mm	40mm	19960mm	290mm

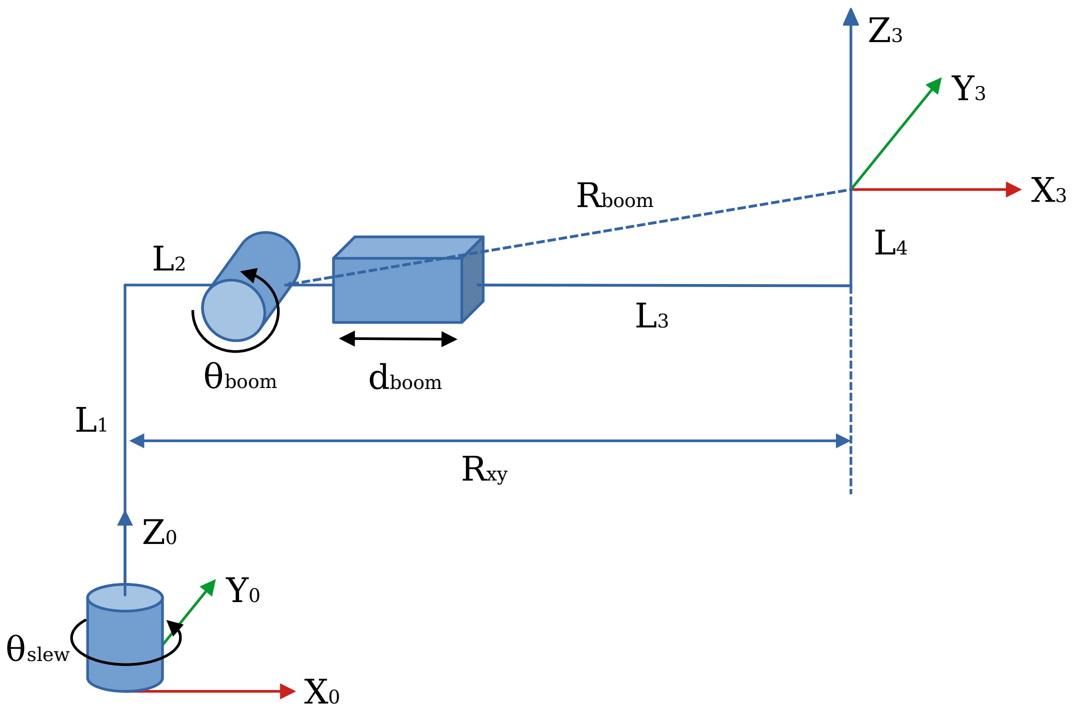


Figure 2.6: Kinematic diagram of gangway

The forward kinematics of the gangway is found based on the Denavit-Hartenberg (DH) method.[5] A combination of translations and rotations around the Z- and X-axis is used to find the kinematics.

Table 2.2: DH table for the kinematic chain

n	**Rot Z**	**Trans Z**	**Trans X**	**Rot X**
1	θ_{slew}	L_1	L_2	$\pi/2$
2	θ_{boom}	0	0	$-\pi/2$
3	0	L_4	$L_3 + d_{boom}$	0

The resulting transformations is:

$$T_1^0 = R_z(\theta_{slew})T_z(L_1)T_x(L_2)R_x(\pi/2) \tag{2.14}$$

$$T_2^1 = R_z(\theta_{boom})R_x(-\pi/2) \tag{2.15}$$

$$T_3^2 = T_z(L_4)T_x(L_3 + d_{boom}) \tag{2.16}$$

The total transformation from origin to boom tip is found by multiplying all the transformation matrices:

$$T_3^0 = T_1^0 T_2^1 T_3^2 \tag{2.17}$$

This is a 4x4 matrix, with element 1,1 to 3,3 being the rotation matrix of the TCP in frame 0, and element 1,4 to 3,4 being the cartesian coordinates in frame 0.

The forward kinematics found from equation 2.17 is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} L_2cos(\theta_{boom}) + cos(\theta_{boom})cos(\theta_{slew})(L_3 + d_{boom}) - L_4cos(\theta_{slew})sin(\theta_{boom}) \\ L_2sin(\theta_{slew}) + cos(\theta_{boom})sin(\theta_{slew})(L_3 + d_{boom}) - L_4sin(\theta_{boom})sin(\theta_{slew}) \\ L_1 + sin(\theta_{boom})(L_3 + d_{boom}) + L_4cos(\theta_{boom}) \end{bmatrix} \tag{2.18}$$

The inverse kinematics is solved analytically, using geometric considerations.
The slew angle is:

$$\theta_{slew} = atan2(y, x) \tag{2.19}$$

The working radius in the XY plane is:

$$R_{xy} = \sqrt{x^2 + y^2} \tag{2.20}$$

The distance from boom bearing to boom tip is:

$$R_{boom} = \sqrt{(R_{xy} - L_2)^2 + (Z - L_1)^2} \tag{2.21}$$

Then the boom displacement can be found:

$$(L_3 + d_{boom})^2 + L_4^2 = R_{boom}^2 \tag{2.22}$$

$$\Rightarrow d_{boom} = \sqrt{R_{boom}^2 - L_4^2} - L_3 \tag{2.23}$$

Finally, the boom angle can be computed:

$$\theta_{boom} = atan2(Z - L_1, R_{xy} - L_2) - atan\left(\frac{L_4}{d_{boom} + L_3}\right) \tag{2.24}$$

Defining a linear transformation between tip and joint velocity:

$$\dot{X} = J\dot{q} \tag{2.25}$$

Where:

$$\dot{q} = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} \dot{\theta}_{slew} \\ \dot{\theta}_{boom} \\ \dot{d}_{boom} \end{bmatrix}, \quad \dot{X} = \begin{bmatrix} \dot{X}_1 \\ \dot{X}_2 \\ \dot{X}_3 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{2.26}$$

Then the jacobian will be:

$$J = \begin{bmatrix} \frac{\partial X_1}{\partial q_1} & \frac{\partial X_1}{\partial q_2} & \frac{\partial X_1}{\partial q_3} \\ \frac{\partial X_2}{\partial q_1} & \frac{\partial X_2}{\partial q_2} & \frac{\partial X_2}{\partial q_3} \\ \frac{\partial X_3}{\partial q_1} & \frac{\partial X_3}{\partial q_2} & \frac{\partial X_3}{\partial q_3} \end{bmatrix} \tag{2.27}$$

$$= \begin{bmatrix} -(L_2 - L_4s_2 + (L_3 + q_3)c_2)s_1 & -(L_4c_2 + (L_3 + q_3)s_2)c_1 & c_2c_1 \\ (L_2 - L_4s_2 + (L_3 + q_3)c_2)c_1 & -(L_4c_2 + (L_3 + q_3)s_2)s_1 & c_2s_1 \\ 0 & -L_4s_2 + (L_3 + q_3)c_2 & s_2 \end{bmatrix}$$

Where: $c_n = cos(q_n)$, $s_n = sin(q_n)$

The joint velocities can be found using the inverse jacobian, provided it is nonsingular:

$$\dot{q} = J^{-1}\dot{X} \tag{2.28}$$

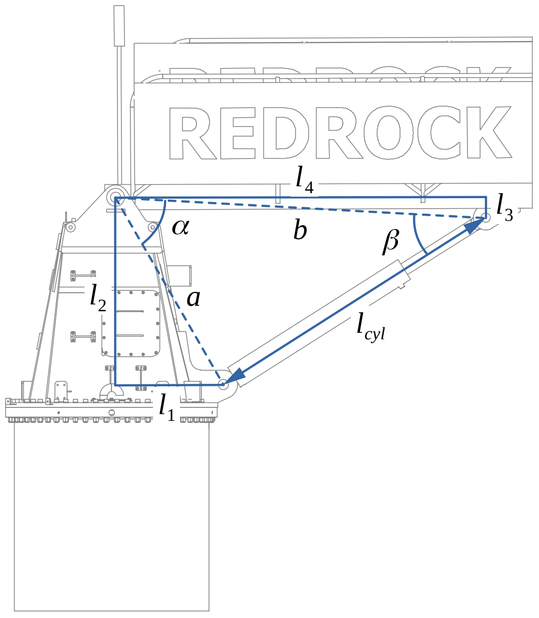


Figure 2.7: Boom cylinder geometry

Since the kinematics is solved using the boom angle as a variable, a conversion between boom angle and cylinder length is needed.

The cosine rule can be used to find the conversion between boom angle and cylinder length:

$$l_{cyl} = \sqrt{a^2 + b^2 - 2ab\ cos(\alpha)} \tag{2.29}$$

Where: $\alpha = \theta_{boom} + \frac{\pi}{2} - atan(\frac{l_1}{l_2}) - atan(\frac{l_3}{l_4})$, $\quad a = \sqrt{l_1^2 + l_2^2}$, $\quad b = \sqrt{l_3^2 + l_4^2}$

Differentiating equation 2.29 gives the cylinder speed:

$$v_{cyl} = \frac{d}{dt}(l_{cyl}) = \frac{ab\ sin(\alpha)}{\sqrt{a^2 + b^2 - 2ab\ cos(\alpha)}}\ \dot{\theta}_{boom} \tag{2.30}$$

Table 2.3: Boom cylinder geometric parameters

l_1	l_2	l_3	l_4
1060mm	1844mm	215mm	3645mm

Which gives: a = 2126.95 mm, b = 3651.34 mm.
The cosine rule can also be used to find the angle of the boom cylinder relative to the boom:

$$\beta = cos^{-1}\left(\frac{b^2 + l_{cyl}^2 - a^2}{2b\ l_{cyl}}\right) \tag{2.31}$$

Then the cylinder force can be related to the moment applied to the boom:

$$F_{cyl} = \frac{T_{boom}}{b\ sin(\beta)} \tag{2.32}$$

Active Motion Compensation

Active motion compensation (AMC) has the goal of keeping the tip of the gangway at the same global position when the ship moves.

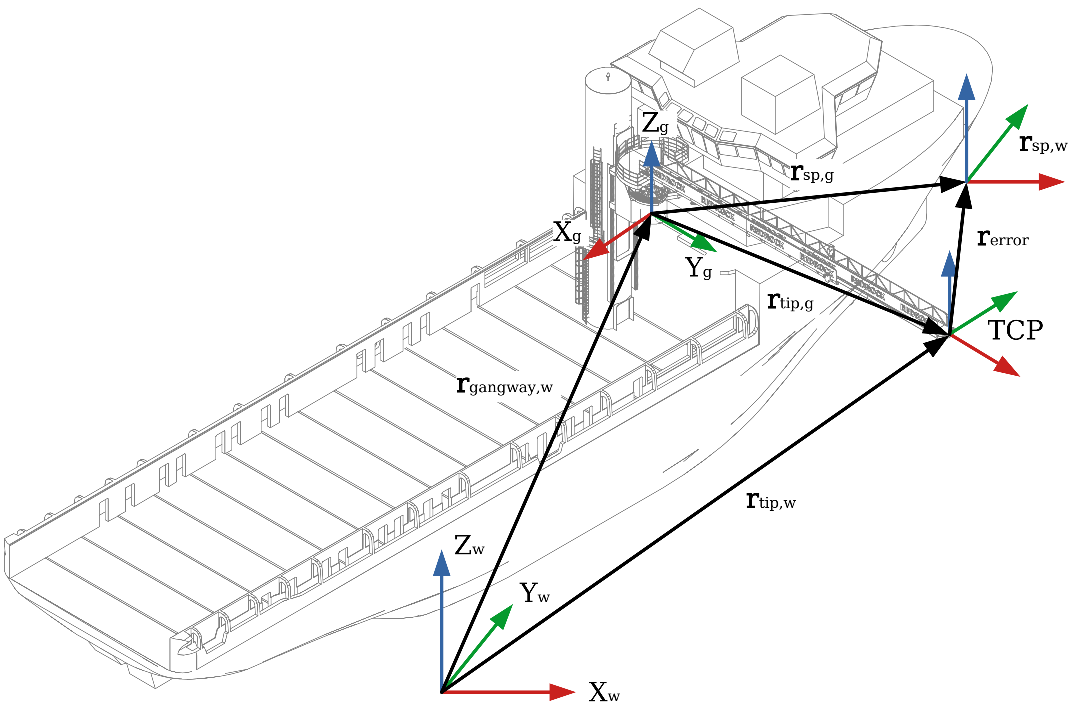


Figure 2.8: Global and local vectors used for calculating position error

The global coordinates of the gangway tip can be found from:

$$\mathbf{r}_{tip,w} = \mathbf{r}_{gangway,w} + \mathbf{R}_g^{w-1}\mathbf{r}_{tip,g} \tag{2.33}$$

Where: $\mathbf{R}_g^w$ is the rotation matrix between world and gangway coordinates. Subscript w, g and t denotes world, gangway and tip coordinates respectively.

Upon AHC entry, the current global position is stored, and the global position error is calculated:

$$\mathbf{r}_{error,w} = \mathbf{r}_{sp,w} - \mathbf{r}_{tip,w} \tag{2.34}$$

The global error can be converted into a position setpoint in local coordinates:

$$\mathbf{r}_{sp,g} = \mathbf{r}_{tip,g} + \mathbf{R}_g^w\mathbf{r}_{error,w} \tag{2.35}$$

Finally, the joint position setpoints is found using inverse kinematics.
A spherical collider with the wind turbine is used.
The vector from boom tip to collider is:

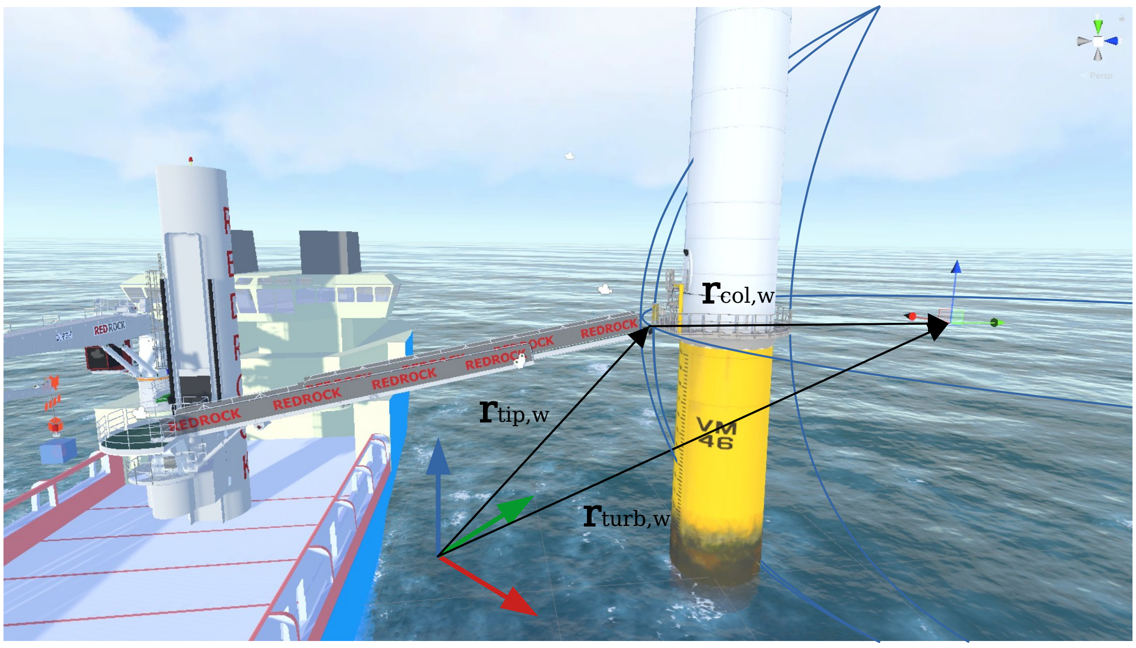
$$\mathbf{r}_{col,w} = \mathbf{r}_{tip,w} - \mathbf{r}_{turb,w} \tag{2.36}$$

The distance from collision between the gangway and turbine is:

$$d_{col} = |\mathbf{r}_{col,w}| - r_{col} \tag{2.37}$$

Where: $\mathbf{r}_{col,w}$ is the center of the spherical collider and r_{col} is the radius of the collider
The collision force is defined as:

$$F_{col} = d_{col}k_s \tag{2.38}$$



$\mathbf{r}_{\text{col,w}}$
$\mathbf{r}_{\text{tip,w}}$
$\mathbf{r}_{\text{turb,w}}$
REDROCK
VM 46

Figure 2.9: Collision between gangway and wind turbine

With k_s being the spring stiffness of the collider, set to 1e5 [N/m]
The collision force is applied locally to the boom tip along the vector from tip to collider:

$$\mathbf{r}_{fcol,t} = \mathbf{R}_{tip}^{w-1} \hat{\mathbf{r}}_{col,w} F_{col} \tag{2.39}$$

Sliding friction between gangway tip and wind turbine can be calculated using hyperbolic tangent friction scaled by the contact force:

$$F_f = -|F_{col}| tanh\left(\frac{v_{tip,w}}{v_0}\right) \tag{2.40}$$

Then this friction force can be transformed into the gangway frame:

$$\mathbf{r}_{ff,t} = \mathbf{R}_{tip}^{w-1} F_f \tag{2.41}$$

2.1.2 Dynamics

Using Lagrangian mechanics, the effective mass on the luffing cylinder can be calculated:

$$M_{eq} = \frac{2E_k}{\dot{x}^2} \tag{2.42}$$

With E_k being the kinetic energy of the boom and telescope combined, and $\dot{x}$ being the cylinder velocity
The rotational kinetic energy of the boom is:

$$E_k = \frac{1}{2} J \omega^2 \tag{2.43}$$

Inserting equation 2.30 and 2.43 into equation 2.42 gives the following expression for equivalent mass for the boom cylinder:

$$M_{eq} = J \frac{a^2 + b^2 - 2ab\ cos(\alpha)}{(ab\ sin(\alpha))^2} \tag{2.44}$$

The parallel axis theorem can be used to find the rotational inertia of a rigidbody with mass m rotated around an axis with distance d from the center of mass:

$$J' = J + md^2 \tag{2.45}$$

The equivalent inertia referenced to the input side of a gearbox is:

$$J_{eq,gb} = \frac{J}{i^2}, \tag{2.46}$$

with i being the gear ratio of the gearbox.
The equivalent inertia of a mass connected to a winch is:

$$J_{eq,winch} = mr^2, \tag{2.47}$$

with m being the connected mass, and r being the winch radius.
The inertias is calculated using the above equations based on the position of the gangway. The implementation is found in appendix C.4.
Gravity force is added at the center of mass of each body. The gravity direction is rotated to match the local coordinate system of the body it is applied to.

2.2 Hydraulic Actuation systems

2.2.1 System Modeling

The orifice equation can be used to calculate the flow through a turbulent cross-section:

$$Q = \alpha C_d A_d \sqrt{\frac{2\Delta p}{\rho}} \tag{2.48}$$

With α being the relative opening bounded to: $[0, 1]$.
For easier calculations, this can be rewritten to the form:

$$Q = \alpha K_v \sqrt{\Delta p} \tag{2.49}$$

Where: $K_v = C_d A_d \sqrt{\frac{2}{\rho}}$

The flow constant K_v can be found from a given nominal flow and pressure drop of a component:

$$K_v = \frac{Q_{nom}}{\sqrt{\Delta p_{nom}}} \tag{2.50}$$

Whenever there is a node without a fixed pressure, a volume must be introduced, and the pressure becomes a state variable. The pressure in this volume is calculated using the hydraulic capacitance model.
The pressure build up of in a closed volume is given by:

$$\dot{p} = \frac{\beta}{V}(Q_{in} - Q_{out} - \dot{V}) \tag{2.51}$$

Where: β is the bulk modulus of the fluid, $\dot{V}$ is the volume change, eg. in a hydraulic cylinder volume.
The bulk modulus of the hydraulic oil is assumed constant at 875 [MPa]

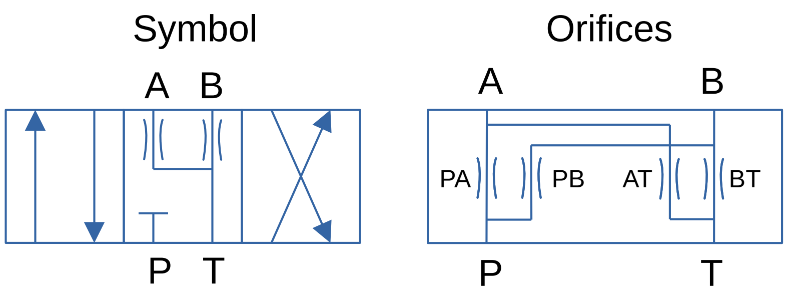


Figure 2.10: DCV model

The directional valves is modeled as a series of variable orifices (fig 2.10), with valve dynamics modeled as a second order transfer function:

$$G(s) = \ = \frac{\omega_{bw}^2}{s^2 + 2\xi\omega_{bw}s + \omega_{bw}^2} \tag{2.52}$$

Where: ω_{bw} is the bandwidth of the valve, and ξ is the damping ratio of the valve. The bandwidth is set to 5 hz with a damping ratio of 0.8.

The valve is modeled with overlap of PA and PB connections, and underlap of AT and BT connections. Both under- and overlap is set to 1%. The under and overlap is achieved by biasing the opening of the orifices. The opening of the orifices is,

$$\alpha_{PA} = u - 0.01, \tag{2.53}$$

$$\alpha_{PB} = -u - 0.01, \tag{2.54}$$

$$\alpha_{AT} = -u - 0.01, \tag{2.55}$$

$$\alpha_{BT} = u - 0.01, \tag{2.56}$$

Since the value is bounded to [0,1] in the orifice model, negative values is ignored.

The load sense (LS) pressure is generated in the valve, with the LS pressure being:

$$p_{ls} = \begin{cases} p_a & \alpha > 0 \\ p_t & \alpha = 0 \\ p_b & \alpha < 0 \end{cases} \tag{2.57}$$

The check valves is modeled as a variable orifice. With the opening following this equation:

$$\alpha_{cv} = \frac{\Delta p - p*}{k_s} \tag{2.58}$$

Where: $p*$ is the opening pressure of the valve and k_s is the spring constant of the valve spring, set to 1 bar.
Modelling of the counter balance valves is based on work done by Morten Bak [4], with a limit added to p_{LS} intended to provide a minimum holdback force for overrunning loads:

$$\alpha_{cbv} = \frac{p_2\psi + min(p_{LS}, p_{LS,max}) - p_{cr,cbv}}{k_s} \tag{2.59}$$

Where: ψ is the area ratio of the valve, and k_s is the spring constant of the valve spring, set to 295 bar.

The pressure compensation valve is based on [4]. Pressure compensation valves is added to keep the pressure drop across the directional valves constant, making the flow through the valve dependent on only the valve opening.
The valve opening is calculated as:

$$\alpha_{pc} = \frac{p_{LS} - p_2 + p^* + k_s}{k_s} \tag{2.60}$$

Where: k_s is the spring constant of the valve, set to 1 bar.
Relief valves are modeled as a variable orifice with the opening calculated as:

$$\alpha_{rv} = \frac{p_1 - p^*}{k_s} \tag{2.61}$$

Where: k_s is the spring constant of the valve, set to 5 bar.
The flow through a hydraulic motor is calculated from the motor speed,

$$Q = \omega D \eta_v, \tag{2.62}$$

with η_v being the volumetric efficiency of the motor.
The torque produced by a hydraulic motor is,

$$T = \Delta p D \eta_{mh} - T_f, \tag{2.63}$$

with η_{mh} being the hydraulic-mechanical efficiency of the motor.

The friction in a hydraulic motor can be found from the load pressure at "creep" speed, and maximum speed.
The static friction constant is found from the pressure drop at creep speed,

$$\mu_s = \Delta p_{creep} V_g, \tag{2.64}$$

and the viscous friction constant is found from the pressure drop at maximum speed,

$$\mu_d = \frac{\Delta p_{creep} - \Delta p_{max}}{\omega_{max}} V_g. \tag{2.65}$$

The friction torque is applied to the motor using a hyperbolic tangent,

$$T_f = \mu_s tanh\left(\frac{\omega}{0.001}\right) + \mu_d \omega \tag{2.66}$$

The hydraulic cylinders is modeled as two variable volumes, with the volume dependent on cylinder position. The position of the cylinder is found via integration of the acceleration:

$$a_{cyl} = \frac{F_{hyd} - F_f}{m_{eq}} \tag{2.67}$$

With m_{eq} being the equivalent mass connected to the cylinder
The flow into volume A is:

$$Q_a = Q_{port,a} - v_{cyl} A_a \tag{2.68}$$

Flow into volume B:

$$Q_{port,b} + v_{cyl} A_b \tag{2.69}$$

The volume at the piston side is:

$$V_a = A_a x_{cyl} \tag{2.70}$$

And for the rod side:

$$V_b = A_b(l_{stroke} - x_{cyl}) \tag{2.71}$$

The force produced by a hydraulic cylinder is

$$F_{hyd} = p_a A_a - p_b A_b - F_f \tag{2.72}$$

The friction in the hydraulic cylinders is modeled with two components. Pressure dependent friction, and static friction.
The pressure dependent friction is:

$$F_{f,p} = |F_{hyd}| k_{pf} \tag{2.73}$$

Where: k_{pf} is the pressure dependent friction constant
The static friction is dependent on cylinder size:

$$F_{f,s} = k_{sf} A_a \tag{2.74}$$

Where: A_a is the area of the cylinder bore side, and k_{sf} is the pressure dependent friction constant.

The friction is applied to the cylinder using a hyperbolic tangent:

$$F_f = -(F_{f,p} + F_{f,s})tanh\left(\frac{v}{0.001}\right) \tag{2.75}$$

The eigenfrequency of the boom system is...

$$\omega_{mh} = \sqrt{\frac{k_x}{m_{eq}}}, \tag{2.76}$$

with k_x defined as,

$$k_x = \frac{\beta A_a^2}{V_{tot,a}} + \frac{\beta A_b^2}{V_{tot,b}} \tag{2.77}$$

With A_a and A_b being the effective piston area of the A and B side of the cylinder. $V_{tot,a}$ and $V_{tot,b}$ is the total volume at the A and B side, being the chamber volume plus the line volume.

2.2.2 Pressure Feedback

The load pressure is defined as:

$$p_l = \frac{F}{A_a} = \frac{p_A A_a - p_b A_b}{A_a} = p_a - \frac{p_b}{\mu_c} \tag{2.78}$$

Where: μ_c is the area ratio of the cylinder defined as $\frac{A_a}{A_b}$
The load pressure is passed through a high pass filter and subtracted from the valve command:

$$G_{pf} = \frac{y_{pf}}{p_l} = K_{pf}\frac{\tau_{pf}}{\tau_{pf}s + 1} \tag{2.79}$$

The time constant of the high pass filter is set to the half natural frequency of the luffing system, then the gain is tuned until oscillation during slow movements disappear. [11]

$$\tau_{pf} = \frac{1}{\omega_{mh}} \tag{2.80}$$

With f_{mh} being the mechanical-hydraulic natural frequency of the luffing system.

The gain is set manually, to where the oscillation stops.

2.3 Numerical Methods

2.3.1 Integration

The differential equations are solved for the highest derivative, then Euler forward integration is used to find the states:

$$x_{k+1} = x_k + \dot{x}_k \cdot dt \tag{2.81}$$

Improving on this, the Taylor expansion can be taken into account for the higher derivatives:

$$x_{k+1} = x_k + \dot{x}_k \cdot dt + \frac{1}{2}\ddot{x}_k \cdot dt^2 \tag{2.82}$$

Solving the differential equations this way simplifies the modeling of the system, to where discontinuities and unlinearities can be added to the system while still being solved in a similar way.

2.3.2 Differentiation

Numerical differentiation can be done by different methods.

- Discretizing the system via Z-transform
- Treating the system as continuous
- A novel method for uncertain sample times, outlined in this subsection.

Pseudocode for the novel differentiation technique:

```
IF input <> last input THEN
    derivative := (input - previous input)/"Time since last update";
ELSE
    derivative := previous derivative;
END_IF
```

Two assumptions make this differentiation approach usable for physical systems:

- Using the equals operator on floating point numbers is usually problematic due to floating point errors, but here the value does not change until the a new value has been received.
- Two readings are unlikely to be exactly the same due to noise and inaccuracies, so this approach works well for finding the time between sensor readings.

A timeout can also be added to set the derivative to zero if the value remains unchanged for too long, eg. if the communication stops.

2.3.3 Atomic Operations

An atomic operation is an operation completed in a single processor cycle. It can not be interrupted by other operations. [13]

When using multiple threads, care needs to be taken when reading and writing variables. A common approach is using mutexes on the variables when doing operations on them.
However, when only reading or writing variables, a simpler approach can be used: In .NET, reading and writing floats are atomic [9]. By keeping the variables as floats and only performing reads and writes we can ensure that no "half written" variables will be read, greatly simplifying usage of multithreading.

2.3.4 Filtering

A first order low pass filter can be implemented discretely as a first order infinite impulse response filter. The k'th output being y_k, and k'th input being x_k,

$$y_{lp,k} = \beta x_k + (1 - \beta) y_{k-1} \tag{2.83}$$

Where β can be calculated from the step dt time and time constant τ,

$$\beta = \frac{dt}{\tau + dt} \tag{2.84}$$

A first order high pass filter can be realized as input minus a first order low pass filter.

$$y_{hp,k} = x_k - y_{lp,k} \tag{2.85}$$

Chapter 3

Methods

3.1 System Identification

3.1.1 Mass and Inertia

The ineria matrices of the rigid bodies (King, boom, telescope) have been simplified to the principal inertias, I_{xx}, I_{yy} and I_{zz}. The ineria at $m\odot$, as well as the mass and center of mass from origin of the body is noted in table 3.1.

Table 3.1: Mass and inertia of bodies

King			**Boom**			**Telescope**		
m	11999	[kg]	m	4114	[kg]	m	2911	[kg]
$m_{\odot,x}$	0.16	[m]	$m_{\odot,x}$	9.953	[m]	$m_{\odot,x}$	10.852	[m]
$m_{\odot,y}$	0.319	[m]	$m_{\odot,y}$	0.048	[m]	$m_{\odot,y}$	0	[m]
$m_{\odot,z}$	1.354	[m]	$m_{\odot,z}$	0.089	[m]	$m_{\odot,z}$	0.12	[m]
Ixx	14728	$[kgm^2]$	Ixx	2365	$[kgm^2]$	Ixx	1142	$[kgm^2]$
Iyy	21407	$[kgm^2]$	Iyy	156321	$[kgm^2]$	Iyy	105726	$[kgm^2]$
Izz	20405	$[kgm^2]$	Izz	157457	$[kgm^2]$	Izz	106386	$[kgm^2]$

3.1.2 Placement of bodies

The origin of the bodies is set at the center of revolute joints for the king and boom. For the telescope the origin is set to the end closest to the king.
The center of the slew bearing is set as the gangway origin, with the placement of the other bodies being:

Table 3.2: Local location of bodies

King			**Boom**			**Telescope**		
x	0	[m]	x	0.04	[m]	x	0.2	[m]
y	0	[m]	y	0	[m]	y	0	[m]
z	0	[m]	z	2.187	[m]	z	0.29	[m]

3.1.3 System sizing

The Red Rock gangway is considerably larger than the system found in [17], and there has not been much work put into detailed design yet. Therefore a preliminary system sizing is carried out to have a system to work with. The dimensions of the components is taken from a CAD design, with gear ratios changed to result in acceptable pressures. The valves are then chosen to get an acceptable speed.

The specifications for the gangway is found in appendix B. The driving factors for the design is:

- Velocity: 2m/s in all axes
- Tip load: 350kg payload + 2000 kg optional winch
- Heel angle: 5 degrees maximum
- Ramp time of 2s from minimum to maximum speed

The system sizing have been done "in reverse", selecting components and verifying that the pressure and speed is acceptable. This makes it easier to input actual components that have discrete values. The chosen parameters is found in table 3.3, and the matlab script used for calculating the values is noted in appendix D
The external loads applied to the boom tip, at maximum radius is:

$$F_{load} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} 10\ [kN] \\ 5\ [kN] \\ 25\ [kN] \end{bmatrix} daf \tag{3.1}$$

With daf being a dynamic amplification factor, set to 1.3

Table 3.3: Chosen hydraulic components

Slew			**Luffing**			**Telescope**		
Gear ratio	624	[-]	Bore diameter	175	[mm]	Gear ratio	13.6	[-]
Displacement	200	[ccm/rev]	Rod diameter	125	[mm]	Drum diameter	0.155	[m]
η_v	0.9	[-]	Valve flow	650	[l/min]	Displacement	100	[ccm/rev]
η_{mh}	0.9	[-]				η_v	0.9	[-]
Valve flow	150	[l/min]				η_{mh}	0.9	[-]
						Valve flow	200	[l/min]

Table 3.4: Resulting speed and pressure

Slew			**Luffing**			**Telescope**		
ΔP	244	[bar]	ΔP	235	[bar]	ΔP	235	[bar]
Angular speed	0.113	[rad/s]	Angular speed	0.106	[rad/s]	Linear speed	2.15	[m/s]
Linear speed	2.27	[m/s]	Linear speed	2.12	[m/s]			

The pressures is a bit on the high side (Table 3.4), leaving little margin for pressure drop across valves, but it is a worst case scenario with all loads applied simultaneously with a dynamic factor on top. It works well for testing the simulation setup.

3.2 High Fidelity Modeling and Simulation

To verify the real-time model, a high fidelity model was created in a multibody simulation software.
OpenModelica is an open source environment for simulation, modeling and optimization based on the Modelica language. It uses an equation driven acausal approach to solving a set of equations.

The multibody simulation is made with the native OpenModelica library, and the hydraulic part is made with components from OpenHydraulics [10], with some adaptions for missing parts.

3.2.1 Multibody system

The multibody simulation consists of rigidbodies and joints. Slew and boom bearings is using revolute joints, and the telescope is connected to the boom using a prismatic joint. The luffing cylinder is modeled using a UPS joint, which is an assembly of Universal-Prismatic-Spherical joints, essentially the same as a distance constraint.

The joints are connected to the hydraulic subsystems via interface ports.

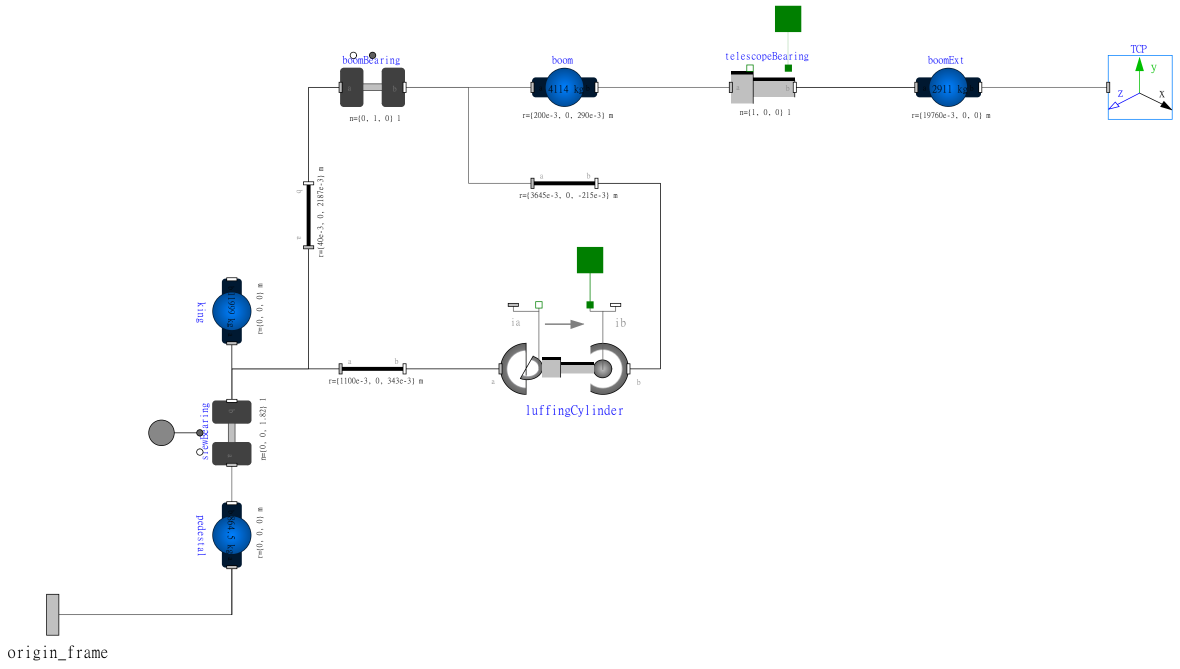


Figure 3.1: The multi body model as modeled in OpenModelica

3.2.2 Hydraulic actuation system

All small volumes in the hydraulic system is set to 1L. This volume is chosen to be large enough for the simulation to be stable with the achievable step time, but low enough to negligibly affect the system dynamics.
The slew system incorporates counterbalance valves on both A and B lines for controlling the slew system during overrunning loads when decelerating. Pressure relief valves is added to protect the system from high dynamic pressures.

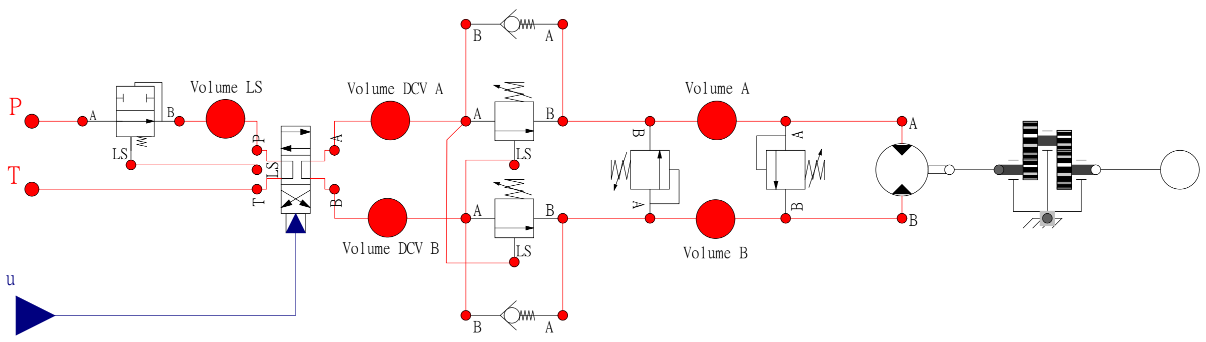


Figure 3.2: The slew system as modeled in OpenModelica

There are five closed volumes used as nodes where the pressure is calculated via flow balance. The luffing system contains two volumes, with further two volumes "hidden" in the cylinder model.

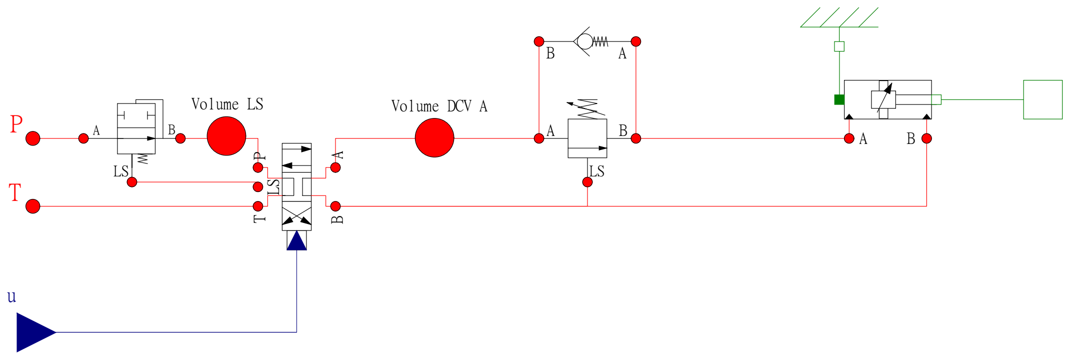


Figure 3.3: The luffing system as modeled in OpenModelica

The telescope system consists of a hydraulic motor connected to a winch drum via a gearbox. In OpenModelica, the winch is simulated using an ideal wheel model, with the shaft fixed, and the outer rim connected to the telescope prismatic joint.

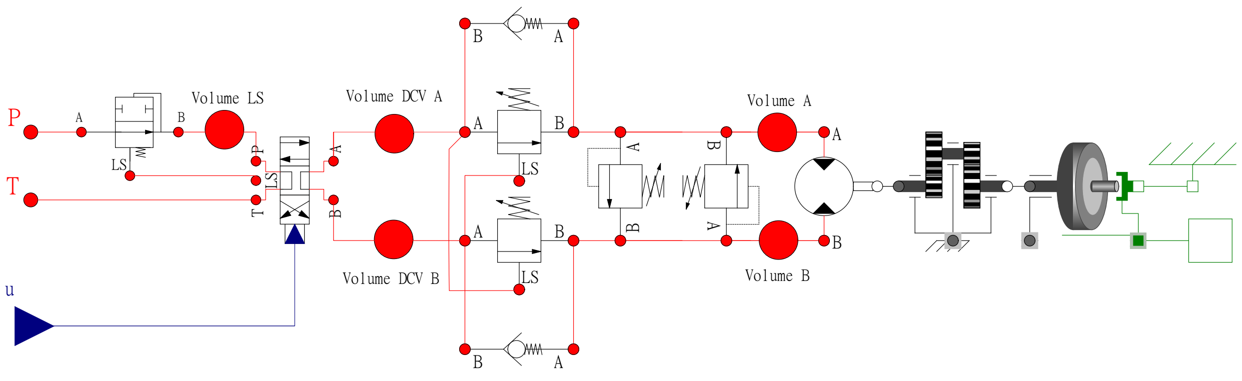


Figure 3.4: The telescope system as modeled in OpenModelica

3.3 Real-time Simulation

The real-time simulation runs in it's own thread in the game engine Unity 3D.

Unity allows scripts written in C# to get easy access to position vectors and rotations of bodies in a scene, and also allows scripts to manipulate the positions.
The simulation itself runs asynchronously from Unity in its own thread. This allows the simulation step time to be orders of magnitude shorter than Unity allows. The communication code takes a while to run, and is also given it's own thread to prevent it from blocking other code from running. The communication and simulation thread is shown in figure 3.5.

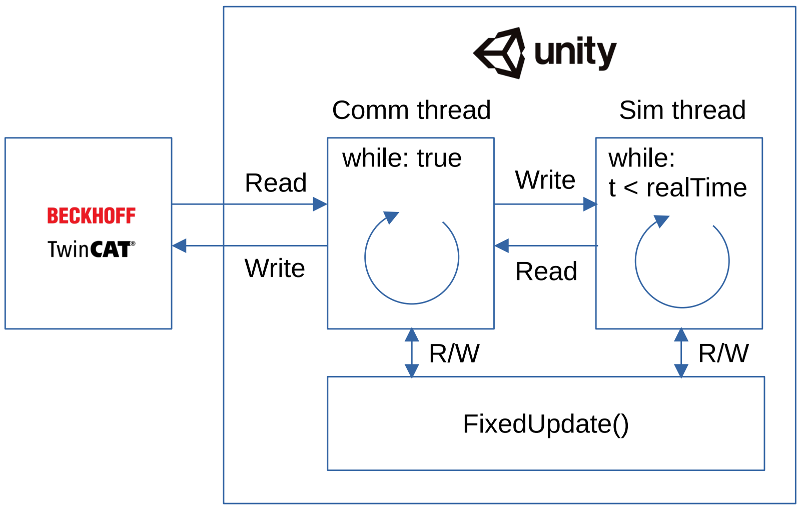


Figure 3.5: Threads in Unity

There are some things that makes Unity less than ideal for real time.

Unity has it's own physics calculation that calculates positions and interactions between objects in the scene. The physics update is tied to the framerate, with the updates running several times before each frame is rendered in the game. There are no guaranteed framerate, so sudden graphics demands may decrease the update rate of the physics, causing instability.

Most of these drawbacks are avoided by using a custom simulation running in it's own thread.

3.3.1 Multibody system

3D models in .obj format can be imported into Unity and used to build a scene with several other objects.

The origin and orientation of 3D models imported into unity is not always correct. A quick way of fixing this is using an empty "gameobject" as the main part, and including the imported part as a child of the main part.

Movement of objects is easily done through the Unity API. The gamobject must have a rigidbody component added, and the transform of the rigidbody can be used and modified in another script.

A script called PositionController (Appendix C.2) was made to simplify moving joints. It contains public variables for Transforms, which will show up in the Unity editor, and can be populated by transforms.

Unity is not fast enough for calculating the contact force smoothly, so this is handled in the simulation thread. The position of the ship and wind turbine is sent from Unity to the simulation thread, then forward kinematics is used to calculate the tip position of the gangway. Appendix C.4 shows the implementation of the simulation script and the contact force calculation.

A place for improvement is adding interpolation of ship position and rotation between Unity updates. Doing this can reduce sudden changes in position when Unity updates the position.

3.3.2 Hydraulic Actuation System

The hydraulic actuation system is simulated using a time-domain based approach. The differential equations in the system is solved for the highest derivative, and then numerical integration is used to find the lower order derivatives.

The time-domain simulation follows five steps:

1. Set pressures of component ports equal to pressure in connected volume.
2. Set control action of controllable valves.
3. Calculate flow through components.
4. Calculate flow going into volumes.
5. Update pressure in volumes.

Appendix C.5 shows how these steps is done for the luffing system.

3.3.3 Communication

The communication between Unity and the control system in TwinCAT is done using the Beckhoff ADS protocol. [3]
Beckhoff provides an ADS client for .NET: TcAdsClient, this can be easily interfaced in Unity, which also runs on .NET. The implementation of the ADS client in unity is found in appendix C.1.

To save overhead, the variables is put into an array before it is sent using the ADS client.

3.4 Control System

3.4.1 Setpoint generation

Controlling the gangway while in closed loop control is done using cylindrical coordinates. The joystick signals is integrated to move the gangway while in AMC mode.

When entering AMC mode, the current global and local position is stored and used to calculate the setpoint.

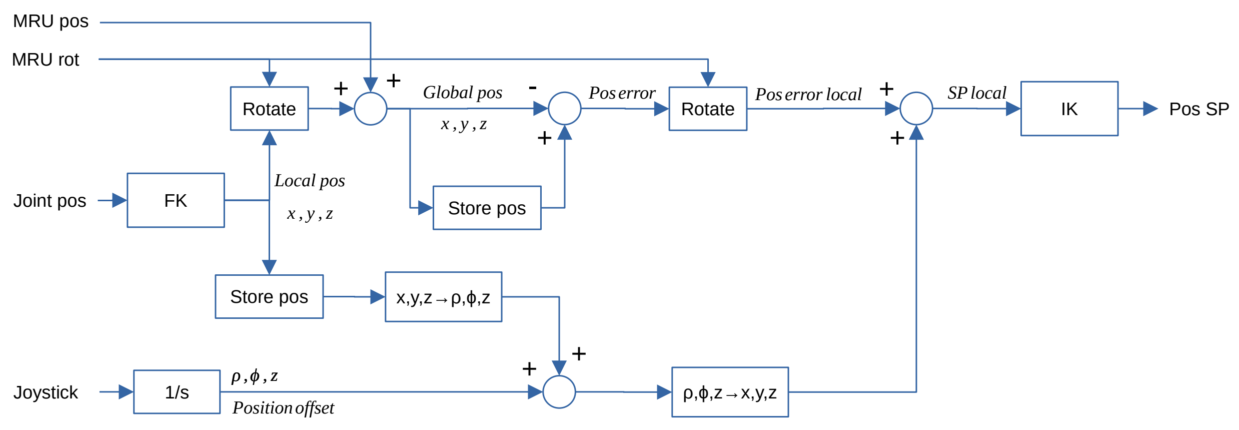


Figure 3.6: Setpoint generation for position controller

3.4.2 Position controller

The implemented controller is a PID controller with velocity feedforward (fig 3.7). Only the K_p gain have been set, so it is a pure P controller:

$$G_c(s) = eK_p + sU(s)K_{ff} \tag{3.2}$$

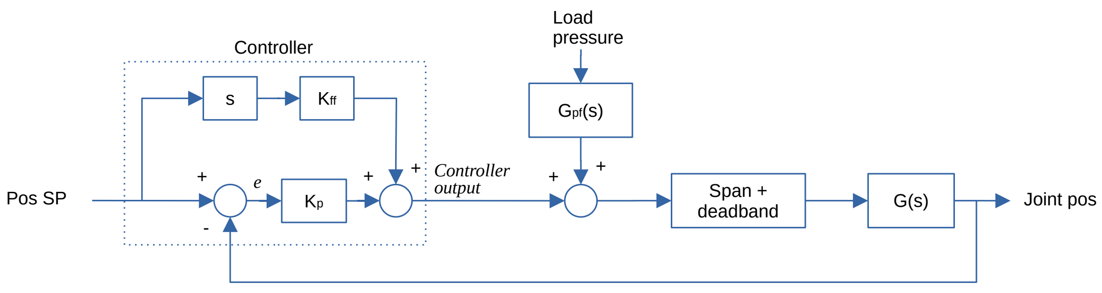


Figure 3.7: Position controller

With K_{ff} being the feed forward gain set to the inverse of the joint speed at maximum valve opening:

$$K_{ff} = \frac{1}{\dot{q}_{max}} \tag{3.3}$$

When not in AMC mode, the whole controller and setpoint generation is bypassed, and the joystick signal is passed directly as controller output. Pressure feedback and span/deadband compensation still remains active in this mode.

3.4.3 Force controller

While force control is active, the force controller functions as a joystick input to the path generation. The axis controlled is the radius, ρ.
The position controller is used as an inner control loop.

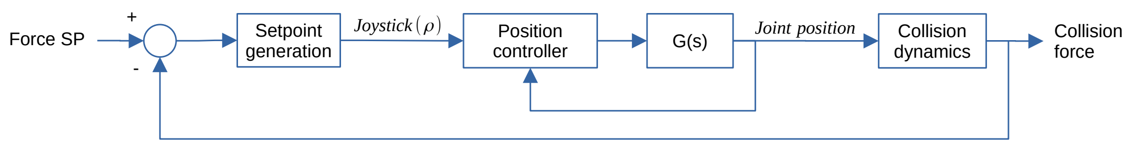


Figure 3.8: Force controller as outer control loop

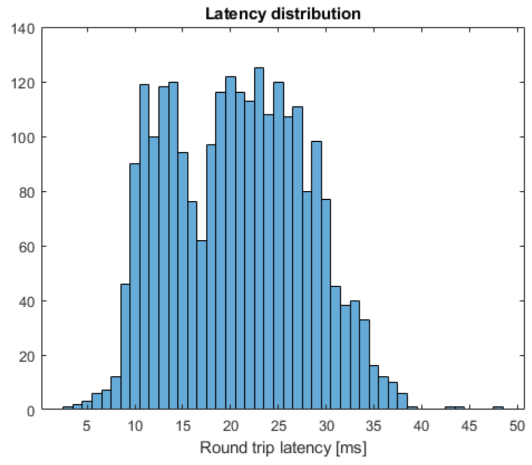
Chapter 4

Results and Discussion

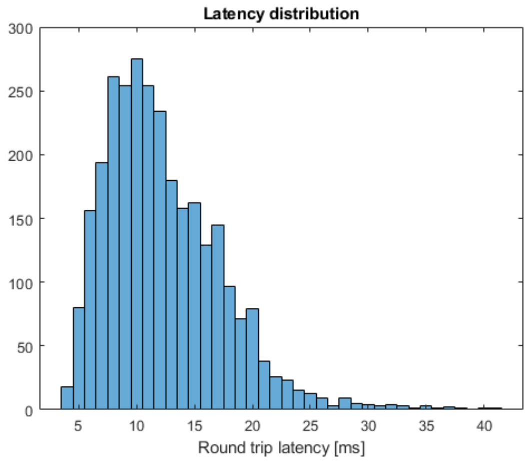
4.1 Communication Performance

4.1.1 Latency

For best controller performance, the latency of the system is critical. A test measuring round trip latency was written, where an output is toggled in the control system running in TwinCAT every 100ms. This signal travels via the communication thread into the simulation thread, and back to the control system; the same path as the control signals. (Figure 4.2) The test was run for about 5 minutes, and the delay between setting and reading the change was logged.



(a) No delay



(b) Added delay

Figure 4.1: Histogram of latency tests

As can be seen in fig 4.1a, the distribution has two peaks, with one peak at twice the latency of the other. This was improved by adding a 1ms delay between reading and writing to ADS in the communication thread.

The most probable reason for the distribution and improvement by adding a delay is that the communication thread moves from "Read" to "Write" before the simulation thread has time to set the "testOut" bit equal to the "testIn" bit, doubling latency whenever that happens.

Further improvements were made by only reading the variable handle once on simulation start in the TwinCAT handler (appendix C.1). The latency improved to where more than 95% of the round trips taking less than 10 ms. This really shows the importance of testing the latency, and how small changes can have a big impact on performance.

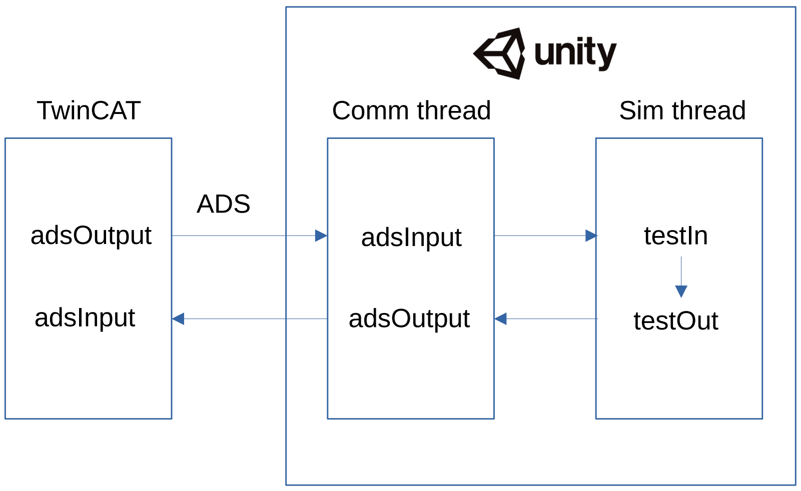


Figure 4.2: Data flow for latency test

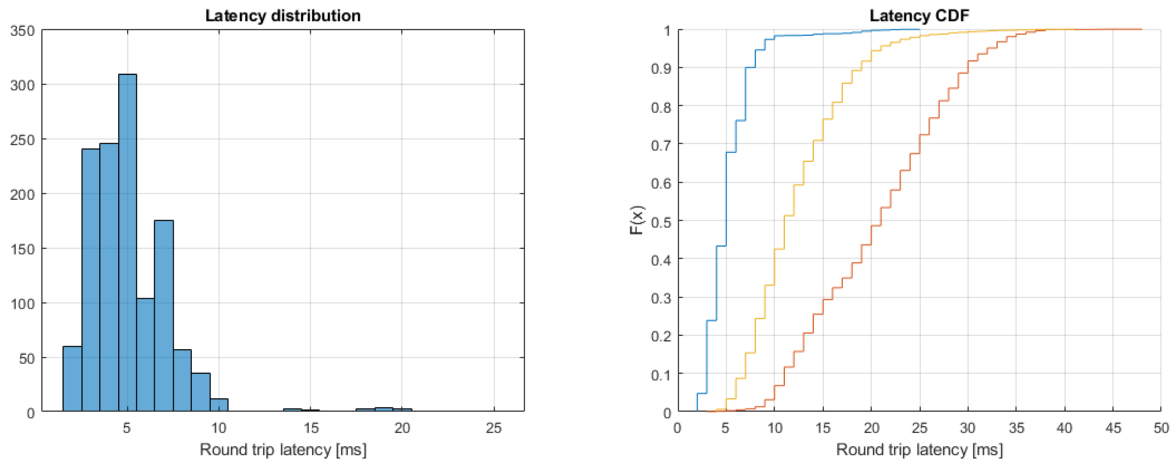


Figure 4.3: Further improvements

4.1.2 Discrete Differetiation

The novel differentiation technique came up as a result of the uncertain sample time that is introduced due to the latency of the ADS protocol.
Position data from the luffing test was differentiated using continuous assumption, and the novel technique finding the sample time on its own.
Contrary to taking the naive approach of differentiating signals every PLC cycle, by only calculating the derivative when the values change, a usable signal is produced even without filtering. Low pass filtering applied, the signal is still a lot less noisy (Figure 4.4). The filter used is a 1st order low pass filter with a time constant of 0.01s
Further improvements upon this may be discretizing the system, or sending the time as a variable with the data being sent.

4.2 Verification of Real-time Simulation

The Unity simulation has been verified against the modelica simulation. All tests were done from the parked position. The steady state pressures is in close agreement, with some dif-

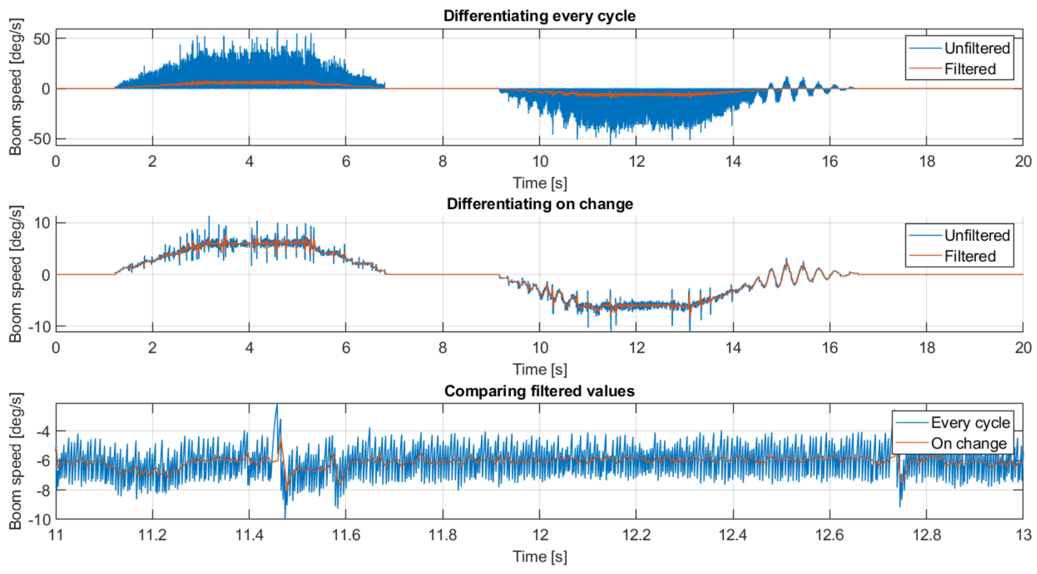


Figure 4.4: Comparison between naive and novel differentiation technique

ferences in transient behaviour.

Figure 4.5 shows the pressure at the start is approximately 70 bar, this is in agreement with hand calculations of the pressure generated due to the weight of the main boom, which tells us that the kinematic equations for the cylinders is correct.
Some of the differences can be explained by the fact that the bulk modulus in the modelica model is a function of density and pressure, and in Unity, the bulk modulus is assumed fixed. This fact is

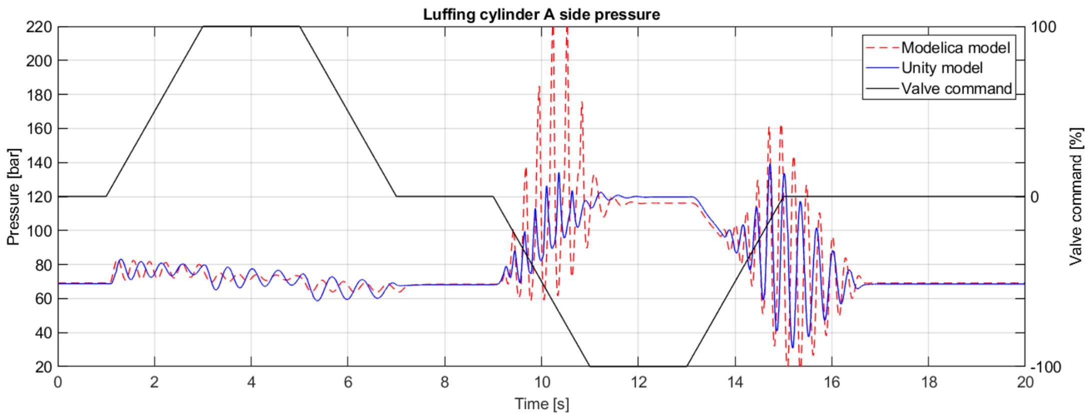


Figure 4.5: Luffing model verification

Figure 4.6 shows the ΔP for the slew system. The steady state pressure is quite close, which means the friction is similar in both simulations. This is expected because the friction constants is calculated from the pressure drop.
The initial pressure spike has a similar shape, but the damping seems to be quite a lot higher in the Unity model. The natural frequency of the slew system is similar in both cases.
Figure 4.7 is the same test, done on the telescope system. This test shows the largest deviation between the real-time model and the modelica model.
The deviation after 15s is due to the Unity simulation hitting the end stop.

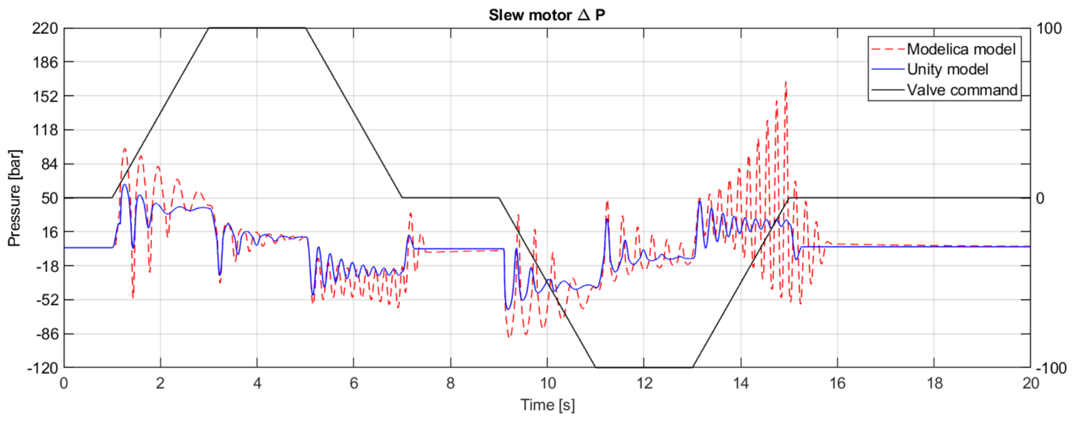


Figure 4.6: Slew model verification

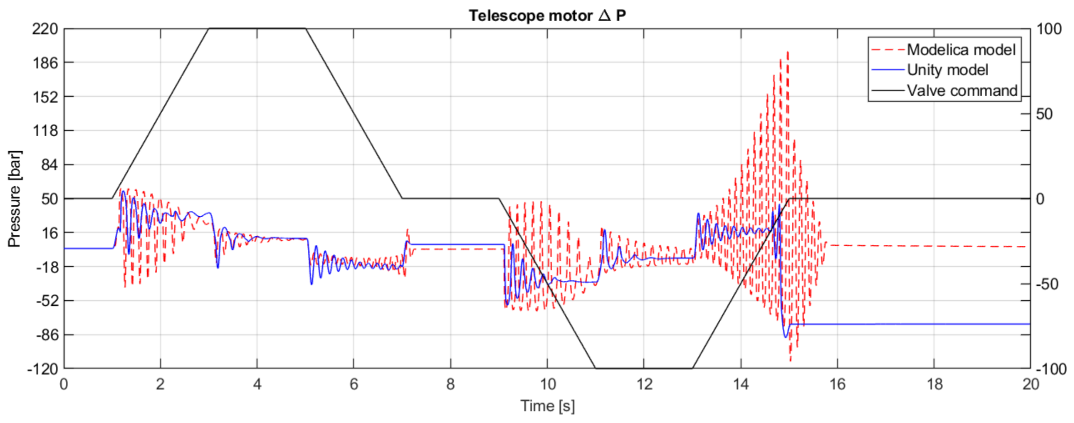


Figure 4.7: Telescope model verification

4.3 Active Damping

The luffing system has some problems with oscillations while driving slowly down. Pressure feedback was incorporated to combat this. The result of this is greatly reducing oscillations in the luffing system (figure 4.8). The test is done from parked position.

The efficacy of pressure feedback is improved by using a valve with a higher bandwidth, the oscillations is virtually eliminated with the 15 hz valve.
5 hz bandwidth is marginal, and the speed is slightly affected due to the oscillating behaviour at the valve input. This is in agreement with what Hagen observed in [7], requiring that the control valve be 3 times faster than the mechanical-hydraulic system, which is about 3 hz as can be seen from the plot.

The deadband compensation can be seen in action when the valve command is close to zero. The valve input oscillates between $\pm 1\%$.

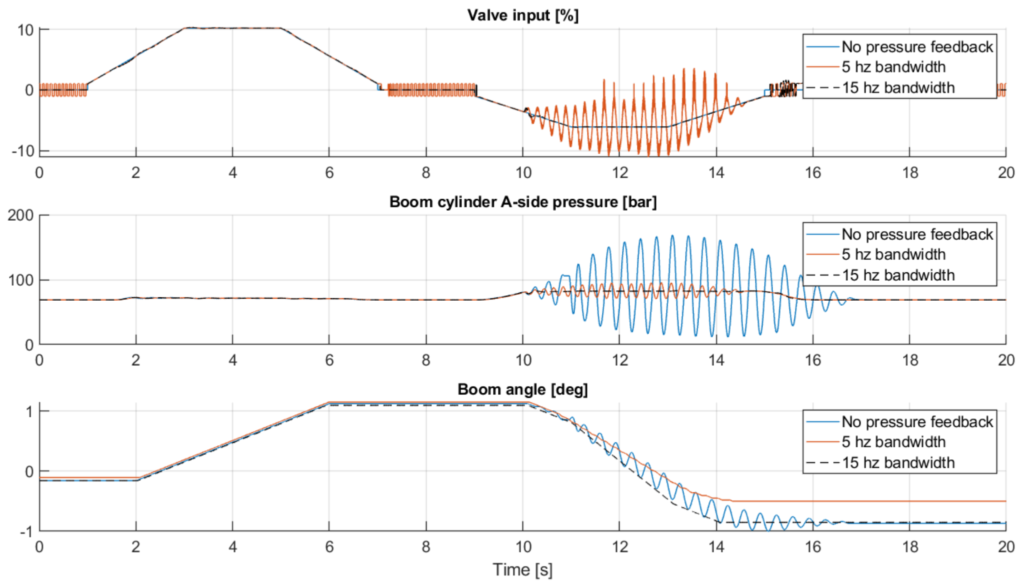


Figure 4.8: Pressure feedback with different valve bandwidths

4.4 Motion Control Performance

An operational scenario has been tested with two different wave conditions. The results of these tests is given in this section.

4.4.1 Active Motion Compensation

Rotation of vectors has not been implemented in TwinCAT yet, so the testing was done with the waves only affecting heave, surge and sway. A small offset was added to the frequency of surge and sway to prevent them being in phase with heave. Two wave conditions were tested. Control input to the gangway was done manually using a gamepad, and the video from the test in normal wave conditions can be found on YouTube [14].

Two scenarios with differing wave conditions were tested, the wave conditions defined as severe wave conditions is 3m heave, with a 10s period, surge and sway set to a quarter of heave.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.75sin(0.12\pi t) \\ 0.75sin(0.11\pi t) \\ 3sin(0.1\pi t) \end{bmatrix} \tag{4.1}$$

The wave conditions defined as normal wave conditions is, 1m heave, with a period of 20s. Surge and sway is still set to a quarter of heave.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.25sin(0.07\pi t) \\ 0.25sin(0.06\pi t) \\ sin(0.05\pi t) \end{bmatrix} \tag{4.2}$$

The scenarios start at about 20 seconds, and the gangway is driven into close proximity to the wind turbine. From 40 to 60 seconds, the gangway is held still in AMC mode before force control is activated, and the gangway is driven to contact the turbine, which can be seen from the contact force in figure 4.9 and 4.10. At 110 seconds, force control is deactivated, and the contact force starts drifting away from the setpoint. The gangway is then driven away from the turbine, and AMC mode is disabled at about 140 seconds. The last operation is driving the gangway back to the initial position.

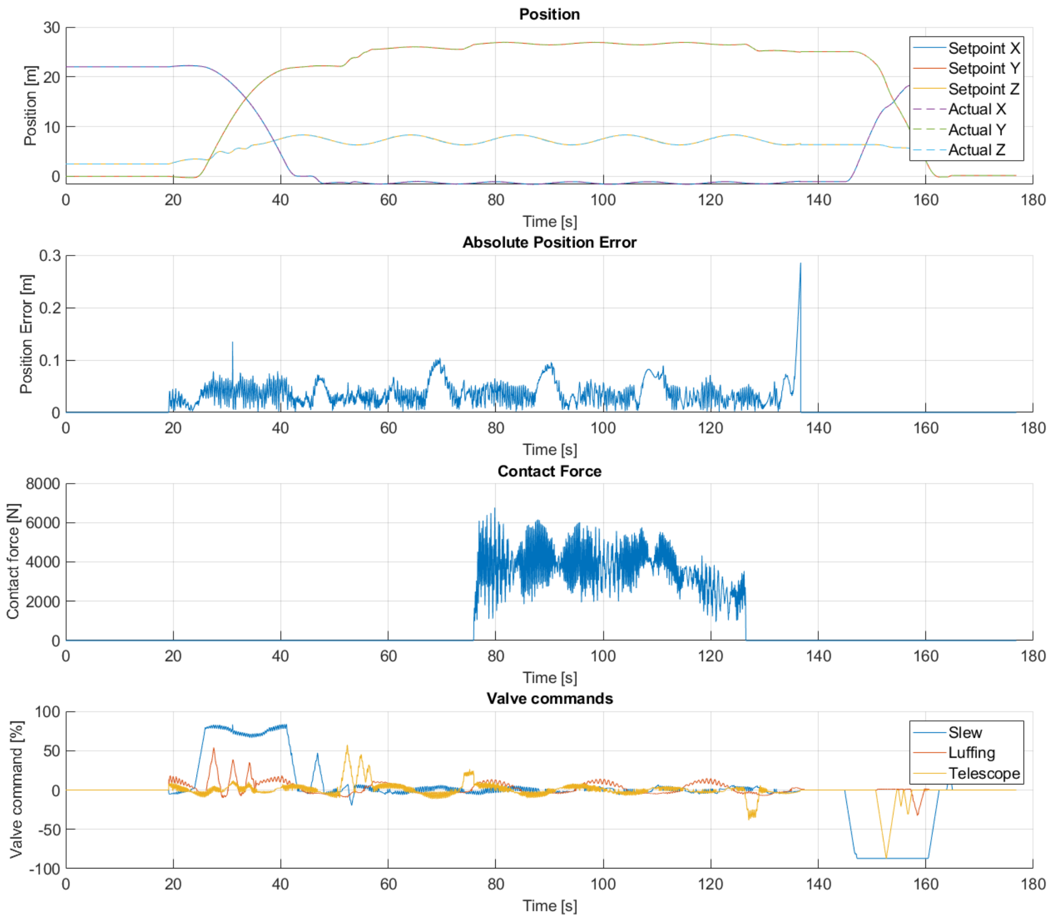


Figure 4.9: Operational cycle with normal waves

The error was calculated as the absolute distance between setpoint and current position (equation 2.34)

The required positioning accuracy for "hover mode" is $\pm 100mm$ according to DNVGL-ST-0358. [1]

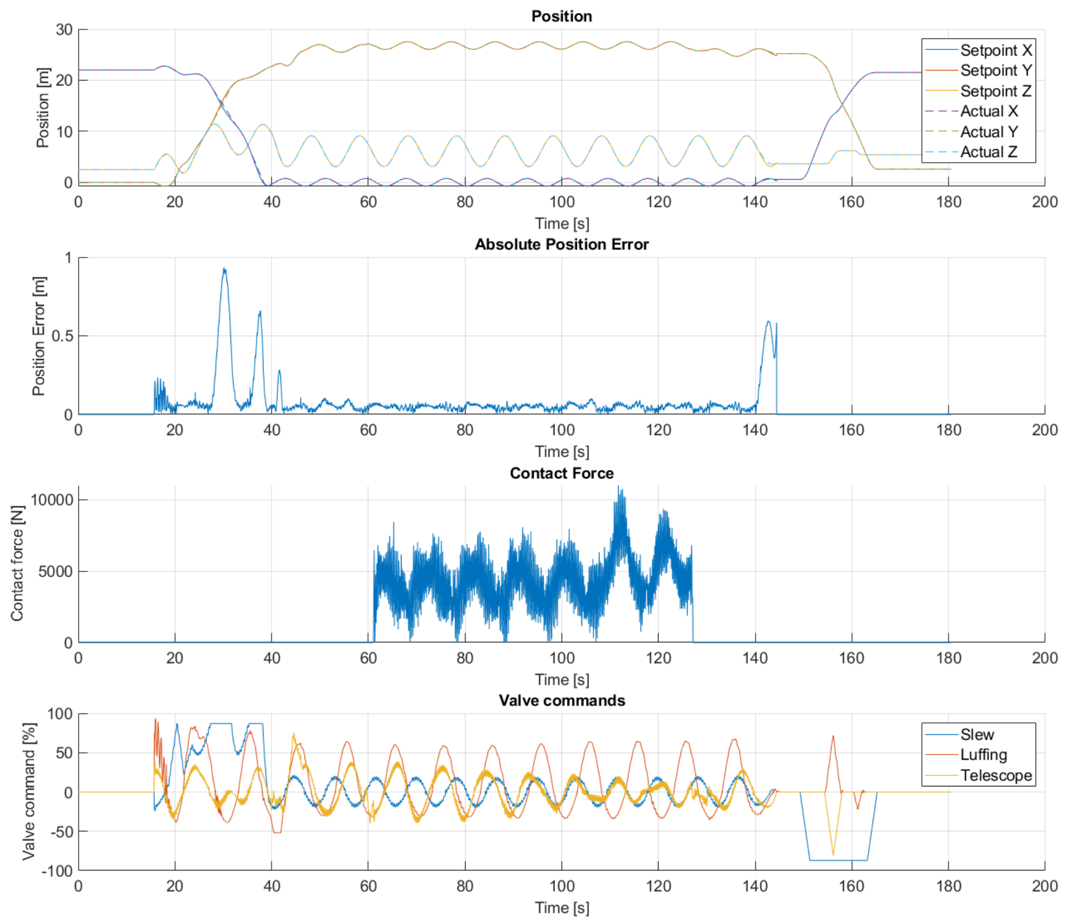


Figure 4.10: Operational cycle with severe waves

The required accuracy is achieved for both the normal and severe wave conditions when not driving the gangway using the joystick. The spikes in position accuracy at the end of the cycle is due to the controller output being ramped down when exiting AMC mode.

The utilization of the luffing system is close to the limit in the severe wave scenario, reaching 70% valve input at 135 seconds in figure 4.10. Both increasing the frequency of the waves and increasing the amplitude makes the severe wave scenario much more demanding than the normal waves.

4.4.2 Force control

In "bumper mode", the goal is to keep a constant force against the wind turbine. The force setpoint is set to 4000 N, same as in the paper by F. Yu. [17].
The contact force is held between 2000N and 6000N, and the position accuracy is not affected in any noticeable ammount in bumper mode.

Chapter 5

Conclusions

The simulation works well as a hardware-in-the-loop test setup. Testing and implementing a simple control system able to achieve 3D compensation with acceptable accuracy was greatly simplified. Furthermore, testing and tuning of a pressure feedback system were done on the simulated model.

Using a game engine for simulating a hydraulic system in near real-time is a novel way of doing hardware-in-the-loop simulations, enabling realistic simulators to be made. Simulating the complete hydraulic system introduces a more realistic test for the control system and allows for feedback of values such as pressures to be implemented for pressure feedback. Furthermore, the hydraulic sensor values are essential for testing the control system.
The developed hardware-in-the-loop setup works well to extend to other products and add more data collection to improve the validity as a digital twin.

Contribution

The developed script for moving and rotating parts made it very easy to implement other applications into the Unity simulator environment. Consequently, Red Rock Marines' new 3D compensated crane was added to the ship and controlled from the same PLC-based control system used on the actual crane in operation. However, no hydraulics was added to the simulation for simplicity; the joint velocities are controlled directly.

The developed discrete differentiation technique is a method that will find the sample rate on its own. This is especially useful whenever the sample rate of the control system is faster than what a digital sensor outputs.

Further work

The simulator has a lot of improvement potential, such as building a standalone Unity application with menus to change simulation parameters, communication ports. VR support is also an option that is available in Unity.

The project can be extended to simulate other hardware, such as cranes and davits, due to the flexibility of using Unity with C# scripting.

For further improvement on the "digital twin" part of the system, models for health and wear can be integrated into the simulation using data from the physical twin.

The friction force due to contact between gangway and wind turbine has not been implemented. When implementing this, the positioning accuracy while in "bumper mode" will increase due to the friction helping to "hold still" the boom tip.

Since the focus of this project has been on developing a real-time simulator environment for doing hardware-in-the-loop simulations, there is a lot of remaining research potential related to the control strategy and motion control algorithms. State-of-the-art model-based control techniques have the potential to improve the positioning accuracy significantly. Testing of novel control methods can easily be carried out using the digital twin developed in this project. Adding vector rotations to the controller is also a good point for improvement.

Appendix A

Task From MSc Catalogue

Prosjekt nr. 3

"Digital Twin of 3D Motion Compensated Gangway"

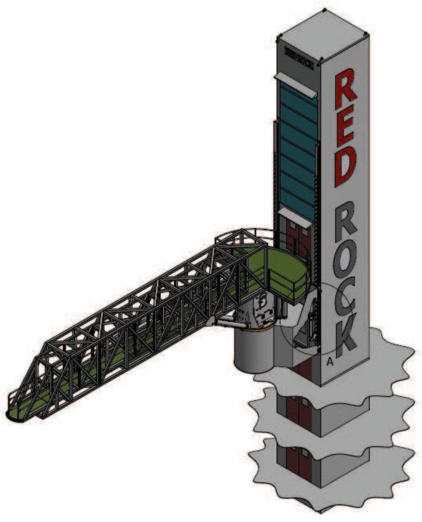
"Master"

Short Introduction

The goal of the project is to develop a digital twin of a novel 3D motion compensated gangway. Red Rock is the leading supplier of the next generation of digital lifting and handling systems for the offshore and marine markets. The 3D motion compensated gangway depicted below is Red Rock Marine's newest product and is state-of-the-art within the offshore wind industry's personnel transfer systems.

Keywords

- Modeling and Simulation
- Multibody Systems
- Hydraulics
- 3D Kinematics and Dynamics
- 3D Motion Compensation and Control
- Real-Time Control Systems



Project Description

The following objectives should be carried out:

1. Modeling of the 3D multibody system and hydraulic actuation systems based on provided designs.
2. Identification of state-of-the-art control techniques for 3D motion compensated gangways.
3. Development and design of control algorithms for controlling the hydraulic actuators.
4. Simulation of relevant motion compensation scenarios and testing of control design.
5. Verify that the model can run faster than real time.
6. Establish a virtual environment (e.g., Unity) to visually demonstrate a realistic operational scenario using the 3D motion compensated gangway's digital twin.

Additional Information

A software platform of choice can be used to achieve the above-mentioned simulation results, such as MATLAB/Simulink (Simscape) or OpenModelica. RedRock can provide Beckhoff TwinCAT system if the students want to test / prove their system in a Hardware-in-the-Loop (HIL) environment.

Contact information:

Full name	E-mail address	Phone number
Daniel Hagen (R&D Engineer)	daniel.hagen@redrock.no	92013462
Torfinn Løvåsen (CTO)	torfinn.lovasen@redrock.no	95118244

Appendix B

Gangway Specifications

1 MAIN DATA

1.1 Specification

Type:	RGCT340-20-30
Emergency lift-off	350 kg @ 30m
Gangway type	Type 2 (DNV-GL)
Movable gangway structure	Yes
Compensated gangway	Yes
Gangway mounted on tower with elevator	Yes
Vertical gangway movement	5m range from 20m-25m ASL
Gangway operating angle (luffing)	+/- 10º extreme amplitude of +/- 15º
Maximum outreach (Rmax)	30 m (horizontal)
Minimum outreach (Rmin)	20 m (horizontal)
Telescopic stroke	10m
Telescopic method	By winch
Compensation Speed, telescopic	Max. ~2.0 m/s
Gangway vertical movement luffing	Hydraulic Cylinders
Gangway vertical movement trolley	Winch
Winch mounted on telescopic part of gangway	1000kg (option for 2000kg)
Elevator capacity	2000 kg
Elevator movement distance	TBA Depends on final vessel design
Elevator stop positions	3 fixed + 1 dynamic (follows the gangway's position)
Width walkway	1.5m - 1.2m
Height Handrails	1.3m
Slewing angle	Max 200 degrees +/- 100 degrees from parking position
Weight (dry without load – incl tower)	~ 85T
Operational modes	Bumper & Hovering mode
Connection method to ship	Welding (tower) with modular sections
Tower height	20m depends on vessel integration/interface and yard supply

Appendix C

Scripts from Unity

C.1 TwinCAT Handler

```
using UnityEngine;
using TwinCAT.Ads;
using System;

public class TwinCAT_Handler : MonoBehaviour
{
  private AdsClient _tcClient;
  public string AMS_id = "1.1.1.1.1.1";
  public int ADS_port = 851;
  public String POU = "P_Unity";
  public String readVariable = "adsOutput";
  public String writeVariable = "adsInput";
  private uint hRead;
  private uint hWrite;

  void Awake()
  {
    _tcClient = new AdsClient();
    _tcClient.Connect(AMS_id, ADS_port);
    if (_tcClient.IsConnected)
    {
      Debug.Log("Twin CAT ADS port connected");
    }
    else
    {
      Debug.LogError("ADS Connection failed");
    }
  }
  void Start()
  {
    hRead = _tcClient.CreateVariableHandle(POU + "." + readVariable);
    hWrite = _tcClient.CreateVariableHandle(POU + "." + writeVariable);
  }

  public float[] ReadFloatArray(int length)
  {
    var value = new float[length];
    int[] args = {length};
    try
    {
      value = (float[])_tcClient.ReadAny(hRead, typeof(float[]), args);
    }
    catch
    {
      Debug.LogError("TC Error - reading ARRAY failed");
    }
    return value;
  }

  public bool WriteValue(object value)
  {
    try
    {
      _tcClient.WriteAny(hWrite, value);
      return true;
    }
    catch (AdsErrorException exc)
    {
      Debug.LogError("TC Write Error " + exc.Message);
    }
    return false;
  }
}
```

C.2 Script for Moving Rigidbodies

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class PositionController : MonoBehaviour
{
    public Transform[] transforms;
    public ControlType[] controlMode;
    public Axis[] axis;
    public Direction[] direction;

    // Initial position of parts
    private Vector3[] initialPosition;
    private Vector3[] initialAngle;

    // Variable to set positions
    public float[] position;

    // Enum to select control modes
    public enum ControlType
    {
        position,
        angle
    }
    public enum Axis
    {
        x,
        y,
        z
    }
    public enum Direction
    {
        positive = 1,
        negative = -1
    }

    // Start is called before the first frame update
    void Start()
    {
        // Create arrays of the correct length
        position = new float[transforms.Length];
        initialPosition = new Vector3[transforms.Length];
        initialAngle = new Vector3[transforms.Length];

        // Store initial positions
        for (int i = 0; i < transforms.Length; i++)
        {
            position[i] = 0f;
            initialPosition[i] = transforms[i].localPosition;
            initialAngle[i] = transforms[i].localEulerAngles;
        }
    }

    public void FixedUpdate()
    {
        // Update visual position of transforms here
        for (int i = 0; i < transforms.Length; i++)
        {
            if (controlMode[i] == ControlType.position)
            {
                if (axis[i] == Axis.x) {transforms[i].localPosition = initialPosition[i] + (int)direction[i] * new ...
                     Vector3(position[i], 0.0f, 0.0f);}
                if (axis[i] == Axis.y) {transforms[i].localPosition = initialPosition[i] + (int)direction[i] * new ...
                     Vector3(0.0f, position[i], 0.0f);}
                if (axis[i] == Axis.z) {transforms[i].localPosition = initialPosition[i] + (int)direction[i] * new ...
                     Vector3(0.0f, 0.0f, position[i]);}
            }
            else if (controlMode[i] == ControlType.angle)
            {
                if (axis[i] == Axis.x) {transforms[i].localEulerAngles = initialAngle[i] + (int)direction[i] * new ...
                     Vector3(position[i], 0.0f, 0.0f);}
                if (axis[i] == Axis.y) {transforms[i].localEulerAngles = initialAngle[i] + (int)direction[i] * new ...
                     Vector3(0.0f, position[i], 0.0f);}
                if (axis[i] == Axis.z) {transforms[i].localEulerAngles = initialAngle[i] + (int)direction[i] * new ...
                     Vector3(0.0f, 0.0f, position[i]);}
            }
        }
    }
}
```

C.3 Main Script

```
using System.Threading;
using System;
using UnityEngine;

// Enum to choose which object to control
public enum controlObject
{
    gangway,
    crane
}

public class Main : MonoBehaviour
{
    // Public variables
    [Header("Communication")]
    public long communicationTime;

    [Header("Simulation options")]
    public bool useTwinCat = false;
    public bool simulateDynamics = true;
    public bool graphOn = false;
    public bool localWaveData = true;
    public bool craneLoad = true;
    public controlObject controlObject;

    [Header("Transforms")]
    public Transform shipTransform;
    public Transform boomTip;
    public Transform hook;

    [Header("Simulation objects")]
    public TwinCAT_Handler _tcHandler;
    public Simulation sim;
    private int controlIndex;
    public AhcMode ahc;
    public GameObject ropeSolver;
    public JointController[] jointControllers;

    // Private variables
    private Vector3 initialShipPosition;
    private float[] adsInput;
    private float[] adsOutput;
    public float[] joystick;
    public float[] effort;
    private float[] waveData;
    public float heave, surge, sway;
    public float roll, pitch, yaw;
    private Vector3 wireVector;
    private Vector2 wireAngle;

    // Variables to be used in threaded communication
    private Thread _adsThread;
    private bool threadStart = false;

    // PLC read/write thread
    private void adsReadWrite()
    {
        // Do not start until unity has loaded and started
        while (!threadStart){}

        System.Diagnostics.Stopwatch _timer;
        _timer = new System.Diagnostics.Stopwatch();
        _timer.Start();

        while (true)
        {
            // Log the communication time and restart timer
            communicationTime = _timer.ElapsedMilliseconds;
            _timer.Restart();

            // Read input from PLC
            adsInput = new float[16];
            adsInput = _tcHandler.ReadFloatArray(adsInput.Length);

            // Translate input array
            Array.Copy(adsInput, 0, waveData, 0, 3);    // Surge, sway, heave
            waveData[3] = adsInput[3]*Mathf.Rad2Deg;    // Roll
            waveData[4] = adsInput[4]*Mathf.Rad2Deg;    // Pitch
            waveData[5] = adsInput[5]*Mathf.Rad2Deg;    // Yaw
            Array.Copy(adsInput, 6, effort, 0, 6);      // Effort is fixed
            sim.testIn = adsInput[15];                  // For testing latency

            // Create new output array
            float[] adsOutput = new float[16];

            // Switch between different objects
            switch (controlObject)
            {
                case controlObject.gangway:
                    // Send inputs to simulation
                    jointControllers[controlIndex].effort[3] = effort[3];   // Elevator
                    sim.slewEffort = effort[0];
                    sim.luffingEffort = effort[1];
                    sim.telescopeEffort = effort[2];

                    // Slight to improve chance of receiving update
                    Thread.Sleep(1);

                    // Construct output array
                    adsOutput[0] = -joystick[0];
                    adsOutput[1] = joystick[1];
                    adsOutput[2] = joystick[2];
                    adsOutput[3] = joystick[3];
                    adsOutput[4] = sim.slewAngle;
                    adsOutput[5] = sim.boomAngle;
                    adsOutput[6] = sim.telescopeLength;
                    adsOutput[7] = sim.slew.motor.A.p*1e-5f;
                    adsOutput[8] = sim.slew.motor.B.p*1e-5f;
                    adsOutput[9] = sim.luffing.cylinder.A.p*1e-5f;
                    adsOutput[10] = sim.luffing.cylinder.B.p*1e-5f;
                    adsOutput[11] = sim.telescope.motor.A.p*1e-5f;
                    adsOutput[12] = sim.telescope.motor.B.p*1e-5f;
                    adsOutput[13] = -sim.contactForce.x;
                    adsOutput[15] = sim.testOut;
                    break;
                case controlObject.crane:
                    // Set effort of joint controller
                    for (int i = 0; i <  jointControllers[controlIndex].effort.Length; i++)
                    {
                        jointControllers[controlIndex].effort[i] = effort[i];
                    }
                    // Construct output array
                    adsOutput[0] = -jointControllers[controlIndex].controller.position[0];          // Slew angle
                    adsOutput[1] = -jointControllers[controlIndex].controller.position[1];          // Boom angle
                    adsOutput[2] = -jointControllers[controlIndex].controller.position[2] - 190f;   // KnBoom angle
                    adsOutput[3] = jointControllers[controlIndex].controller.position[3];           // Wire position
                    adsOutput[4] = -jointControllers[controlIndex].controller.position[4] + 90f;           // Left 3D ...
                         Tool angle
                    adsOutput[5] = -jointControllers[controlIndex].controller.position[5] + 90f;           // Right ...
                         3D Tool angle
                    adsOutput[6] = wireAngle.x;
                    adsOutput[7] = wireAngle.y;
                    break;
            }

            // Send output to PLC
            _tcHandler.WriteValue(adsOutput);
        }
    }

    // Awake is called before start and can enable disable gameobjects
    void Awake()
    {
        // Enable/disable full simulation
        sim.gameObject.GetComponent<Simulation>().enabled = simulateDynamics;

        // Enable/disable load on crane
        ropeSolver.SetActive(craneLoad);
    }

    // Start is called before the first frame update
    void Start()
    {
        // Create new communication thread
        if (useTwinCat)
        {
            _adsThread = new Thread(adsReadWrite);
            _adsThread.Start();
        }

        // Get initial positon of Ship
        initialShipPosition = shipTransform.position;

        // Set up graphs
        if (graphOn)
        {
            DebugGUI.SetGraphProperties("Boom pressure A", "Boom pressure A", -1, 1, 0, new Color(1, 1, 0), true);
            DebugGUI.SetGraphProperties("Boom pressure B", "Boom pressure B", -1, 1, 0, new Color(0, 1, 0), true);
        }

        // Create arrays
        effort = new float[6];
        joystick = new float[4];
        waveData = new float[6];
    }

    // Update is called once per frame
    // GetButtonDown needs to be in Update to work reliably
    void Update()
    {
        // Allow communication thread to start
        threadStart = true;

        // Loop through moving different objects
        if (Input.GetButtonDown("Fire3"))
        {
            controlIndex++;
            if (controlIndex > jointControllers.Length - 1) {controlIndex = 0;}
            controlObject = (controlObject)controlIndex;
        }
    }

    void FixedUpdate()
    {
        // Read joystick input
        joystick[0] = Input.GetAxis("Horizontal");
        joystick[1] = Input.GetAxis("Vertical");
        joystick[2] = Input.GetAxis("HorizontalRight");
        joystick[3] = Input.GetAxis("VerticalRight");

        // Set controlIndex according to selected object enum
        controlIndex = (int)controlObject;

        // Set inputs of all joint controllers not controlled to zero
        for (int i = 0; i < jointControllers.Length; i++)
        {
            if (i != controlIndex)
            {
                for (int j = 0; j < jointControllers[i].effort.Length; j++)
                {
                    jointControllers[i].effort[j] = 0f;
                }
            }
        }

        // If not using twincat, the simulation is controlled via gamepad
        if (!useTwinCat)
        {
            // Send joystick values to joint controllers
            for (int i = 0; i < 4; i++)
            {
                jointControllers[controlIndex].effort[i] = joystick[i];
            }

            // If hydraulic simulation is active, gangway joint controller is controlled by sim
            if (simulateDynamics)
            {
                // Send inputs to simulation
                sim.elevatorEffort = jointControllers[0].effort[3];
                sim.slewEffort = jointControllers[0].effort[0];
                sim.luffingEffort = jointControllers[0].effort[1];
                sim.telescopeEffort = jointControllers[0].effort[2];
            }
        }

        // Overwrite joint positions if hydraulic sim is active
        if (simulateDynamics)
        {
            // Overwrite position of joints according to simulation
            jointControllers[0].controller.position[0] = sim.slewAngle;
            jointControllers[0].controller.position[1] = sim.boomAngle;
            jointControllers[0].controller.position[2] = sim.telescopeLength;
        }

        // Simulate wave motion
        if (localWaveData)
        {
            surge = Mathf.Sin(0.22f*Time.time)*1.2f;
            sway = Mathf.Sin(0.23f*Time.time)*1.25f;
            heave = Mathf.Sin(0.21f*Time.time)*1.5f;
            roll = Mathf.Sin(0.24f*Time.time)*5.0f;
            pitch = Mathf.Sin(0.25f*Time.time)*1.0f;
            yaw = Mathf.Sin(0.26f*Time.time)*1.0f;
        }
        else
        {
            // External wave data is jittery :(
            surge = waveData[0];
            sway = waveData[1];
            heave = waveData[2];
            roll = waveData[3];
            pitch = waveData[4];
            yaw = waveData[5];
        }

        // Find angle of wire rope
        // Vector from boom tip to hook
        wireVector = boomTip.position - hook.position;
        // Rotate vector with boomtip
        wireVector = Quaternion.AngleAxis(boomTip.eulerAngles.y, Vector3.up) * wireVector;
        // Calculate angle
        wireAngle = new Vector2(Mathf.Atan2(wireVector.z, wireVector.y), Mathf.Atan2(wireVector.x, wireVector.y));

        // Set ship position
        // This converts the right hand rule system to unity coordinates
        shipTransform.position = initialShipPosition + new Vector3(-surge, -heave, sway);
        shipTransform.localEulerAngles = new Vector3(roll, -yaw, pitch);

        // Local AHC mode on gangway
        ahc.update();


        ///////////////////////////////////////////////////////////////////////////
        ////////////////////////////////// Logging ////////////////////////////////
        ///////////////////////////////////////////////////////////////////////////

        // Log heave
        if (graphOn)
        {
            DebugGUI.Graph("Boom pressure A", sim.luffing.cylinder.A.p*1e-5f);
            DebugGUI.LogPersistent("Boom pressure A", sim.luffing.cylinder.A.p*1e-5f);
            DebugGUI.Graph("Boom pressure B", sim.luffing.cylinder.B.p*1e-5f);
            DebugGUI.LogPersistent("Boom pressure B", sim.luffing.cylinder.B.p*1e-5f);
        }

        if (simulateDynamics)
        {
            Grapher.Log(sim.slew.volumeA.p*1e-5f, "Pressure slew A", Color.red);
            Grapher.Log(sim.slew.volumeB.p*1e-5f, "Pressure slew B", Color.green);
            Grapher.Log(sim.slew.volumeDCVa.p*1e-5f, "Pressure DCV A", Color.blue);
            Grapher.Log(sim.slew.volumeDCVb.p*1e-5f, "Pressure DCV B", Color.yellow);
            //Grapher.Log(sim.slew.position, "Slew position", Color.white);

            Grapher.Log(sim.telescope.volumeA.p*1e-5f, "Pressure telescope A", Color.red);
            Grapher.Log(sim.telescope.volumeB.p*1e-5f, "Pressure telescope B", Color.green);
            //Grapher.Log(sim.telescope.volumeDCVa.p*1e-5f, "Pressure DCV A", Color.blue);
            //Grapher.Log(sim.telescope.volumeDCVb.p*1e-5f, "Pressure DCV B", Color.yellow);
            //Grapher.Log(sim.telescope.pcvVolume.p*1e-5f, "PCV pressure", Color.white);

            Grapher.Log(sim.luffing.cylinder.A.p*1e-5f, "Pressure cylinder A", Color.red);
            Grapher.Log(sim.luffing.cylinder.B.p*1e-5f, "Pressure cylinder B", Color.green);
            //Grapher.Log(sim.luffing.cylinder.cylinderLength, "Cylinder length", Color.white);
            //Grapher.Log(sim.boomTorque, "Boom torque", Color.white);

            Grapher.Log(sim.slew.motor.Tfric, "Friction torque", Color.white);
            Grapher.Log(sim.slew.motor.Jeq, "Equivalent inertia", Color.white);
            Grapher.Log(sim.slew.motor.Tload, "Load torque", Color.white);
        }
    }

    void OnDestroy()
    {
        if (_adsThread != null)
        {
            _adsThread.Abort();
        }
    }
}
```

C.4 Simulation Thread

```
using UnityEngine;
using System.Threading;

[System.Serializable]
// Inherits MonoBehaviour to be able to adjust parameters in Unity
public class Simulation : MonoBehaviour
{
    [Header("Transforms")]
    public Transform mruTransform;
    public Transform tipTransform;
    public Transform turbineTransform;

    [Header("Simulation setup")]
    public float dt = 1e-7f;
    public double simulationTime = 0.0D;
    public float realTime = 0.0f;
    public ulong iteration = 0;
    private ulong previousIteration = 0;
    private System.Diagnostics.Stopwatch _timer;

    [Header("Hydraulic data")]
    public float smallVolume = 0.001f;
    public float bulkModulus = 875e6f;
    public float hpuPressure = 250e5f;

    [Header("Collision")]
    public float springConstant = 1e8f;
    public float contactDamping = 1e7f;
    private Vector3 turbinePos;
    private Vector3 gangwayPos;
    public Vector3 tipVelocity;
    public Vector3 oldPos;
    private Quaternion gangwayRotation;
    private Quaternion tipRotation;
    public Vector3 contactForce;
    public Vector3 rGangway;
    public Vector3 rTipTurbine;
    public Vector3 rTipTurbineLocal;
    public float penetrationDepth;
    public float turbineRadius = 4f;
    public Vector3 tipPos;

    [Header("Inertial data")]
    // Inertia matrices are Ixx, Iyy, and Izz at the center of mass
    public float kingMass = 11999;
    public Vector3 kingCoM = new Vector3(0.16f, 0.319f, 1.354f);
    public Vector3 kingInertia = new Vector3(14728f, 21407f, 20405f);
    public float boomMass = 4114f;
    public Vector3 boomCoM = new Vector3(9.953f, 0.048f, 0.089f);
    public Vector3 boomInertia = new Vector3(2365f, 156321f, 157457f);
    public float telescopeMass = 2911f;
    public Vector3 telescopeCoM = new Vector3(10.852f, 0f, 0.12f);
    public Vector3 telescopeInertia = new Vector3(1142f, 105726f, 106386f);

    [Header("Speed")]
    public float elevatorSpeed = 0.5f;      // Meters per second
    public float slewSpeed = 0.1f;    // Radians per second
    public float luffingSpeed = 0.1f;     // Radians per second
    public float telescopeSpeed = 2.0f;      // Meters per second

    [Header("Limits")]
    public Limit elevatorLimit = new Limit(0.0f, 8.5f);

    [Header("Inputs")]
    public float elevatorEffort;
    public float slewEffort, luffingEffort, telescopeEffort;

    [Header("Outputs")]
    public float elevatorHeight = 0.0f;
    public float slewAngle = 0.0f, boomAngle = 0.0f, telescopeLength = 0.0f;

    // Simulation objects
    public Telescope telescope;
    public Slew slew;
    public Luffing luffing;

    // Private variables
    private CylinderKinematics cylinderkinematics;
    private Kinematics kinematics;

    // Gravity
    public Vector3 g;
    public Vector3 glocal;

    // Update thread
    private Thread _simThread;
    private void simUpdate()
    {
        // Stopwatch for realtime
        _timer = new System.Diagnostics.Stopwatch();
        _timer.Start();

        while (true)
        {
            update();
        }
    }

    // Start is called before the first frame update
    void Start()
    {
        // Calculate system sizing from the given speeds

        // Initialize system objects
        telescope = new Telescope();
        slew = new Slew();
        luffing = new Luffing();
        cylinderkinematics = new CylinderKinematics();
        kinematics = new Kinematics();

        // Initialize values
        oldPos = new Vector3(0f, 0f, 0f);
        slew.hpuPressure = hpuPressure;
        luffing.hpuPressure = hpuPressure;
        telescope.hpuPressure = hpuPressure;

        // Start simulation thread
        _simThread = new Thread(simUpdate);
        _simThread.Start();
    }

    void FixedUpdate()
    {
        // Update real time for display in unity
        realTime = (float)_timer.Elapsed.TotalSeconds;

        // g vector on gangway (z axis flipped)
        g = Quaternion.Inverse(mruTransform.rotation) * new Vector3(0, -9.81f, 0);

        // Save position of gangway in global coordinates
        turbinePos = turbineTransform.position;
        gangwayPos = mruTransform.position;
        gangwayRotation = mruTransform.rotation;
        tipRotation = tipTransform.rotation;

        // Set radius of turbine to the attached collider
        turbineRadius = turbineTransform.GetComponent<SphereCollider>().radius;
    }

    // Function to calculate contact force between tip and turbine
    public Vector3 calculateContactForce(float dt)
    {
        // Find local tip position from FK
        rGangway = kinematics.forwardKinematics(new float[] {slew.position, luffing.boomAngle, telescope.position});
        tipPos = rGangway;

        // Finds gangway tip position in global unity coordinates
        rGangway = gangwayPos + gangwayRotation * new Vector3(rGangway.x, rGangway.y, -rGangway.z);

        // Find vector between gangway tip and turbine
        rTipTurbine = turbinePos - rGangway;
        //rTipTurbine = new Vector3(rTipTurbine.x, 0f, rTipTurbine.z);   // Remove vertical axis since turbine is vertical

        // Convert to local boom coordinates
        rTipTurbineLocal = Quaternion.Inverse(tipRotation) * rTipTurbine;
        rTipTurbineLocal = new Vector3(rTipTurbineLocal.x, rTipTurbineLocal.y, -rTipTurbineLocal.z);

        // Calculate tip velocity
        tipVelocity = (rTipTurbineLocal - oldPos)/dt;

        // Store old position
        oldPos = rTipTurbineLocal;

        // Calculate penetration depth
        penetrationDepth = rTipTurbine.magnitude - turbineRadius;

        // Contact force
        Vector3 contactForce = new Vector3(0f, 0f, 0f);
        if (penetrationDepth < 0f)
        {
            contactForce = rTipTurbineLocal.normalized * penetrationDepth * springConstant + tipVelocity*contactDamping;
        }

        return contactForce;
    }

    public float testIn;
    public float testOut;
    public float boomTorque;

    public void update()
    {
        // Run simulation in sync with real time
        while (simulationTime < _timer.Elapsed.TotalSeconds)
        {
            // Calculate local direction of g vector on the boom
            glocal = new Vector3(g.x, g.y, -g.z);    // Convert to correct gangway coordinates by flipping z
            // Follow rotation of boom
            glocal = Quaternion.Euler(0, 0, slew.position*Mathf.Rad2Deg) * glocal;
            glocal = Quaternion.Euler(0, luffing.boomAngle*Mathf.Rad2Deg, 0) * glocal;

            // Calculate contact force between gangway and turbine
            contactForce = calculateContactForce(dt);

            // Calculate equivalent inertias and loads from the current position
            // Inertia of boom + telescope in Y axis
            // Jeq = J + m*r^2
            float boomInertiaY = boomInertia.y + boomMass*boomCoM.x*boomCoM.x
                            + telescopeInertia.y + telescopeMass*(telescopeCoM.x + ...
                                 telescope.position)*(telescopeCoM.x + telescope.position);
            luffing.cylinder.equivalentMass = cylinderkinematics.equivalentMass(boomInertiaY, luffing.boomAngle);

            // External load on boom cylinder
            boomTorque = boomMass*boomCoM.x*glocal.z + telescopeMass*(telescopeCoM.x + telescope.position)*glocal.z + ...
                 (20f + telescope.position)*contactForce.z;
            luffing.cylinder.Fload = cylinderkinematics.cylinderForce(boomTorque, luffing.boomAngle);

            // Inertia of king + boom + telescope in Z axis
            float kingInertiaZ = kingInertia.z + (boomInertia.z + telescopeInertia.z)*Mathf.Cos(luffing.boomAngle) + ...
                 (boomInertia.x + telescopeInertia.x)*Mathf.Sin(luffing.boomAngle)
                                    + ...
                                         boomMass*boomCoM.x*boomCoM.x*Mathf.Cos(luffing.boomAngle)*Mathf.Cos(luffing.boomAngle)
                                    + telescopeMass*(telescopeCoM.x + telescope.position)*(telescopeCoM.x + ...
                                         telescope.position)*Mathf.Cos(luffing.boomAngle)*Mathf.Cos(luffing.boomAngle);
            slew.inertia = kingInertiaZ;
            slew.Tload = (boomMass*boomCoM.x*glocal.y + telescopeMass*(telescopeCoM.x + telescope.position)*glocal.y ...
                 + (20f + telescope.position)*contactForce.y)*Mathf.Cos(luffing.boomAngle);

            // External load on telescope
            telescope.mass = telescopeMass;
            telescope.Fload = telescopeMass*glocal.x - contactForce.x;

            //Send values to simulation objects
            telescope.effort = telescopeEffort;
            telescope.boomAngle = boomAngle;
            slew.effort = slewEffort;
            luffing.effort = luffingEffort;

            // Update simulation objects
            telescope.update(dt);
            slew.update(dt);
            luffing.update(dt);

            // Update position values
            boomAngle = luffing.boomAngle*Mathf.Rad2Deg;
            telescopeLength = telescope.position;
            slewAngle = slew.position*Mathf.Rad2Deg;

            // Apply limit to positions
            elevatorLimit.constrain(ref elevatorHeight);

            // Update simulation time
            simulationTime += dt;
            iteration++;

            testOut = testIn;

            // Throw error if loop can't keep up
            if ((iteration - previousIteration) > 1e5)
            {
                Debug.LogError("Simulation out of sync, increasing stepsize");
                dt = 2.0f*dt;
                iteration = previousIteration;
                return;
            }
        }
        previousIteration = iteration;
    }

    // Kill thread when exiting
    void OnDestroy()
    {
        if (_simThread != null)
        {
            _simThread.Abort();
        }
    }
}
```

C.5 Luffing Simulation

```
using UnityEngine;
using System;
public class Luffing
{
    // Input/output
    public float velocity = 0f;
    public float boomAngle = 0f;
    public float effort = 0f;
    // Dimension data
    public float bulkModulus = 875e6f;
    public float hpuPressure;

    // Simulation objects
    public HydraulicCylinder cylinder;
    public DCV dcv;
    public Volume volumeDCVa;
    public CBV cbvA;
    public PressureCompensator pcv;
    public Volume pcvVolume;
    private CylinderKinematics kinematics;

    // Constructor sets dimensions etc
    public Luffing()
    {
        // Calculate required flow
        float Qnom = 650.0f;

        // Create simulation objects
        dcv = new DCV(Qnom, 5.0f);
        volumeDCVa = new Volume(bulkModulus, 0.001f);
        cylinder = new HydraulicCylinder(250.0e-3f, 175.0e-3f, 1700.0e-3f, 2285.0e-3f);
        cbvA = new CBV(Qnom, 16.0f, 200.0f, 4.0f, 50.0f);
        pcv = new PressureCompensator(Qnom, 5.0f, 5.0f);
        pcvVolume = new Volume(bulkModulus, 0.001f);

        // Cylinder kinematics
        kinematics = new CylinderKinematics();
    }
    public void update(float dt)
    {
        // Set system pressures
        pcv.A.p = hpuPressure;
        pcv.B.p = pcvVolume.p;
        pcv.LS.p = dcv.LS.p;
        dcv.A.p = volumeDCVa.p;
        dcv.B.p = cylinder.B.p;
        dcv.P.p = pcvVolume.p;
        dcv.T.p = 1.0e5f;
        cbvA.A.p = volumeDCVa.p;
        cbvA.B.p = cylinder.A.p;
        cbvA.setLS(cylinder.B.p);

        // Send effort value to valves
        dcv.alpha = effort;

        // Flow balance in volumes
        pcvVolume.Q = pcv.B.Q + dcv.P.Q;
        volumeDCVa.Q = dcv.A.Q + cbvA.A.Q;
        cylinder.A.Q = -(cbvA.B.Q);
        cylinder.B.Q = -(dcv.B.Q);

        // Boom angle
        // Convert from cylinder length to boom angle
        boomAngle = kinematics.boomAngle(cylinder.cylinderLength);

        // Update simulation objects
        // Flow is calculated from port pressures
        // New pressure is calculated in the volumes
        dcv.update(dt);
        volumeDCVa.update(dt);
        cylinder.update(dt);
        cbvA.update(dt);
        pcv.update(dt);
        pcvVolume.update(dt);
    }
}
```

C.6 Slew Simulation

```
using UnityEngine;

public class Slew
{
    // Input/output
    public float position = 0f;
    public float velocity = 0f;
    public float effort = 0.0f;
    public float Tload = 0.0f;

    // Dimension data
    public float inertia;
    public float bulkModulus = 875e6f;
    public float gearRatio = 624.0f;
    public float hpuPressure = 250e5f;

    // Simulation objects
    public DCV dcv;
    public Volume volumeDCVa;
    public Volume volumeDCVb;
    public Volume volumeA;
    public Volume volumeB;
    public HydraulicMotor motor;
    public CBV cbvA;
    public CBV cbvB;
    public PressureCompensator pcv;
    public Volume pcvVolume;
    public ReliefValve rvA;
    public ReliefValve rvB;

    // Constructor sets dimensions etc
    public Slew()
    {
        // Calculate required flow
        float Qnom = 150.0f;

        // Create simulation objects
        dcv = new DCV(Qnom, 5.0f);
        volumeA = new Volume(bulkModulus, 0.001f);
        volumeB = new Volume(bulkModulus, 0.001f);
        volumeDCVa = new Volume(bulkModulus, 0.001f);
        volumeDCVb = new Volume(bulkModulus, 0.001f);
        motor = new HydraulicMotor(200f, 0.9f, 0.9f);
        cbvA = new CBV(Qnom, 5.0f, 250.0f, 5.0f, 60f);
        cbvB = new CBV(Qnom, 5.0f, 250.0f, 5.0f, 60f);
        pcv = new PressureCompensator(Qnom, 5.0f, 5.0f);
        pcvVolume = new Volume(bulkModulus, 0.001f);
        rvA = new ReliefValve(Qnom, 5.0f, 250f);
        rvB = new ReliefValve(Qnom, 5.0f, 250f);

        // Set the friction of the motor
        // deltaPcreep(bar), deltaPmax(bar), motorSpeed(rad/s)
        motor.setFrictionForce(5.0f, 10.0f, 78f);
    }

    public void update(float dt)
    {
        // System pressures
        pcv.A.p = hpuPressure;
        pcv.B.p = pcvVolume.p;
        pcv.LS.p = dcv.LS.p;
        dcv.A.p = volumeDCVa.p;
        dcv.B.p = volumeDCVb.p;
        dcv.P.p = pcvVolume.p;
        dcv.T.p = 1.0e5f;
        motor.A.p = volumeA.p;
        motor.B.p = volumeB.p;
        cbvA.A.p = volumeDCVa.p;
        cbvA.B.p = volumeA.p;
        cbvB.A.p = volumeDCVb.p;
        cbvB.B.p = volumeB.p;
        cbvA.setLS(volumeDCVb.p);
        cbvB.setLS(volumeDCVa.p);
        rvA.A.p = volumeA.p;
        rvA.B.p = volumeB.p;
        rvB.A.p = volumeB.p;
        rvB.B.p = volumeA.p;

        // Send effort value to valves
        dcv.alpha = -effort;

        // Flow balance in volumes
        pcvVolume.Q = pcv.B.Q + dcv.P.Q;
        volumeDCVa.Q = dcv.A.Q + cbvA.A.Q;
        volumeDCVb.Q = dcv.B.Q + cbvB.A.Q;
        volumeA.Q = motor.A.Q + cbvA.B.Q + rvA.A.Q + rvB.B.Q;
        volumeB.Q = motor.B.Q + cbvB.B.Q + rvB.A.Q + rvA.B.Q;

        // Telescope position
        position = -motor.theta/gearRatio;
        velocity = -motor.thetaDot/gearRatio;

        // Calculate load
        motor.Tload = Tload/gearRatio;
        motor.Jeq = inertia/(gearRatio*gearRatio);

        // Update simulation objects (This is where the magic happens)
        dcv.update(dt);
        volumeA.update(dt);
        volumeB.update(dt);
        volumeDCVa.update(dt);
        volumeDCVb.update(dt);
        motor.update(dt);
        cbvA.update(dt);
        cbvB.update(dt);
        pcv.update(dt);
        pcvVolume.update(dt);
        rvA.update(dt);
        rvB.update(dt);
    }
}
```

C.7 Telescope Simulation

```
using UnityEngine;
using System;

public class Telescope
{
    // Input/output
    public float position = 0f;
    public float velocity = 0f;
    public float boomAngle = 0f;
    public float effort = 0.0f;
    public float Fload = 0.0f;
```

```
// Dimension data
public float mass = 2910.0f;
public float bulkModulus = 875e6f;
public float gearRatio = 13.6f;
public float winchRadius = 0.155f;
public float endstopSpringStiffness = 1e6f;
public float endstopDamping = 1e5f;
public float maxLength = 10.0f;
public float hpuPressure = 250e5f;

// Simulation objects
public DCV dcv;
public Volume volumeDCVa;
public Volume volumeDCVb;
public Volume volumeA;
public Volume volumeB;
public HydraulicMotor motor;
public CBV cbvA;
public CBV cbvB;
public PressureCompensator pcv;
public Volume pcvVolume;
public ReliefValve rvA;
public ReliefValve rvB;

// Constructor sets dimensions etc
public Telescope()
{
    // Calculate required flow
    float Qnom = 200.0f;

    // Create simulation objects
    dcv = new DCV(Qnom, 5.0f);
    volumeA = new Volume(bulkModulus, 0.001f);
    volumeB = new Volume(bulkModulus, 0.001f);
    volumeDCVa = new Volume(bulkModulus, 0.001f);
    volumeDCVb = new Volume(bulkModulus, 0.001f);
    motor = new HydraulicMotor(100f, 0.9f, 0.9f);
    cbvA = new CBV(Qnom, 5.0f, 250.0f, 5.0f, 60f);
    cbvB = new CBV(Qnom, 5.0f, 250.0f, 5.0f, 60f);
    pcv = new PressureCompensator(Qnom, 5.0f, 5.0f);
    pcvVolume = new Volume(bulkModulus, 0.001f);
    rvA = new ReliefValve(Qnom, 5.0f, 250f);
    rvB = new ReliefValve(Qnom, 5.0f, 250f);

    // Set the friction of the motor
    // deltaPcreep(bar), deltaPmax(bar), motorSpeed(rad/s)
    motor.setFrictionForce(5.0f, 10.0f, 209f);
}

public void update(float dt)
{
    // System pressures
    pcv.A.p = hpuPressure;
    pcv.B.p = pcvVolume.p;
    pcv.LS.p = dcv.LS.p;
    dcv.A.p = volumeDCVa.p;
    dcv.B.p = volumeDCVb.p;
    dcv.P.p = pcvVolume.p;
    dcv.T.p = 1.0e5f;
    motor.A.p = volumeA.p;
    motor.B.p = volumeB.p;
    cbvA.A.p = volumeDCVa.p;
    cbvA.B.p = volumeA.p;
    cbvB.A.p = volumeDCVb.p;
    cbvB.B.p = volumeB.p;
    cbvA.setLS(volumeDCVb.p);
    cbvB.setLS(volumeDCVa.p);
    rvA.A.p = volumeA.p;
    rvA.B.p = volumeB.p;
    rvB.A.p = volumeB.p;
    rvB.B.p = volumeA.p;

    // Send effort value to valves
    dcv.alpha = effort;

    // Flow balance in volumes
    pcvVolume.Q = pcv.B.Q + dcv.P.Q;
    volumeDCVa.Q = dcv.A.Q + cbvA.A.Q;
    volumeDCVb.Q = dcv.B.Q + cbvB.A.Q;
    volumeA.Q = motor.A.Q + cbvA.B.Q + rvA.A.Q + rvB.B.Q;
    volumeB.Q = motor.B.Q + cbvB.B.Q + rvB.A.Q + rvA.B.Q;

    // Telescope position
    position = motor.theta*winchRadius/gearRatio;
    velocity = motor.thetaDot*winchRadius/gearRatio;

    // End stops
    float endStopForce = 0f;
    if (position > maxLength)
    {
        endStopForce = endstopSpringStiffness*(position - maxLength) + endstopDamping*velocity;
    }
    else if (position < 0.0f)
    {
        endStopForce = endstopSpringStiffness*position + endstopDamping*velocity;
    }
    else
    {
        endStopForce = 0f;
    }

    // Calculate motor load
    motor.Tload = (Fload + endStopForce)*winchRadius/gearRatio;
    motor.Jeq = mass*winchRadius*winchRadius/(gearRatio*gearRatio);
```

```
        // Update simulation objects (This is where the magic happens)
        dcv.update(dt);
        volumeA.update(dt);
        volumeB.update(dt);
        volumeDCVa.update(dt);
        volumeDCVb.update(dt);
        motor.update(dt);
        cbvA.update(dt);
        cbvB.update(dt);
        pcv.update(dt);
        pcvVolume.update(dt);
        rvA.update(dt);
        rvB.update(dt);
    }
}
```

C.8 Orifice

```
using UnityEngine;

public class Orifice
{
    // Flow constant
    private float Kv;

    // Valve opening
    public float alpha = 1.0f;

    // Flow/pressure drop
    public float Q, dp;

    // Constructor to calculate Kv
    // Converts l/min and bar to SI units
    public Orifice(float Qnom = 100f, float deltaPnom = 100f)
    {
        this.Kv = (Qnom/60000f)/Mathf.Sqrt(deltaPnom*1e5f);
    }

    // Flow calculation
    public void calculateFlow(float dp)
    {
        alpha = Mathf.Clamp01(alpha);
        this.dp = dp;
        this.Q = alpha*Kv*Mathf.Sign(dp)*Mathf.Sqrt(Mathf.Abs(dp));
    }

    // Pressure drop
    public void calculateDeltaP(float Q)
    {
        this.Q = Q;
        this.dp = Mathf.Pow(Q/Kv,2);
    }
}
```

C.9 Volume

```
public class Volume
{
    // Variables
    public float beta, volume;
    public float p = 0f, Q = 0f;

    public Volume(float beta = 875e6f, float volume = 0.001f)
    {
        this.beta = beta;
        this.volume = volume;
    }

    // Function to calculate pressure from flow balance
    // pDot = beta/volume*(Qin - Qout)
    public void update(float dt)
    {
        float pDot = beta/volume*Q;
        p += pDot*dt;
    }
}
```

Appendix D

Matlab Script for System Sizing

```
clc; clear all; close all;
format long;

% System sizing
% Calculating pressures and flow from given size instead of calculating
% system sizing from loads. This way it is easier to verify the chosen
% components and change them later
iSlewRing = 8;              % Gear ratio between pinion and slew ring [-]
iSlew = iSlewRing*78.0;     % Total slew gear ratio [-]
iTelescope = 13.6;          % Gear ratio of telescope winch [-]
rTelescopeDrum = 0.155;     % Radius of telescope winchdrum [m]
dLuff = 0.175;              % Diameter of luffing cylinders [m]
vgSlew = 200;               % Displacement of slew motor [ccm]
vgTelescope = 100;          % Displacement of telescope winch motor [ccm]
Qslew = 150;                % Flow of slew valve [l/min]
Qluff = 650;                % Flow of luffing valve [l/min]
Qtelescope = 200;           % Flow of telescope valve [l/min]

% Constants
g = 9.81;               % Gravity [m/s^2]
daf = 1.3;              % Dynamic amplification factor
nmh = 0.9;              % Hydraulic mechanical efficiency [-]
nv = 0.9;               % Volumetric efficiency [-]

% External forces
Fx = 10000;             % Axial force on boom tip [N]
Fy = 5000;              % Side force on boom tip [N]
Fz = 25000;             % Vertical force on boom tip [N]

% Mass and dimensions of components
jKing = 20405;          % In Z axis [kgm^2]
mBoom = 4114;           % Mass of boom [kg]
jBoom = 156321;         % In Y axis (Z is almost equal) [kgm^2]
rBoom = 9.993;          % Along X axis [m]
mTelescope = 2911;      % Mass of telescope [kg]
jTelescope = 105725;    % In Y axis (Z is almost equal) [kgm^2]
rTelescope = 10.892;    % Distance to CoM [m]
rMin = 20;              % Minimum radius of gangway [m]
rMax = 30;              % Maximum radius of gangway [m]

% Required speed and acceleration
vMax = 2;               % Maximum speed in all axes [m/s]
tRamp = 2;              % Ramp time [s]
aMax = vMax/tRamp;      % Maximum acceleration in all axes [m/s^2]
heelAngle = 5;          % Maximum heel angle [deg]
boomAngleMax = 20;      % Maximum boom angle [deg]

% Required joint velocity and acceleration
pos = FK([0 0 0]);      % Position to calculate joint velocity
dt = 0.0000001;         % For numerically calculating joint speed
qd = (IK(pos + [vMax*dt vMax*dt vMax*dt]') - IK(pos))/dt;
qdd = qd * aMax/vMax;   % Maximum joint acceleration

% Forces in system
Jslew = jKing + jBoom + mBoom*rBoom^2 + jTelescope ...
        + mTelescope*(rTelescope + rMax - rMin)^2;
Tslew = qdd(2)*Jslew;
Tslew = Tslew + mBoom*rBoom*g*sind(heelAngle) ...
        + mTelescope*(rTelescope + rMax - rMin)*g*sind(heelAngle);
Tslew = Tslew + rMax*Fy;
Tslew = Tslew*daf/nmh;
TslewPinion = Tslew/iSlewRing;  % Torque at slew pinion

Jboom = jBoom + mBoom*rBoom^2 ...
        + jTelescope + mTelescope*(rTelescope + rMax - rMin)^2;
Tboom = qdd(1)*Jboom;
Tboom = Tboom + mBoom*rBoom*g + mTelescope*(rTelescope + rMax - rMin)*g;
Tboom = Tboom + rMax*Fz;
Fluff = cylinderForce(Tboom, 0);
Fluff = Fluff*daf;

Ftelescope = qdd(3)*mTelescope + mTelescope*g*sind(boomAngleMax) + Fx;
Ttelescope = Ftelescope*rTelescopeDrum;
Ttelescope = Ttelescope*daf/nmh;

% Resulting pressure
Pslew = Tslew/iSlew/(vgSlew/2/pi/100^3) * 1e-5
Aluff =  (0.25*pi*(dLuff)^2)*2;
```

```
Pluff = Fluff / Aluff * 1e-5
Ptelescope = Ttelescope/iTelescope/(vgTelescope/2/pi/100^3) * 1e-5

% Resulting speed
nSlew = (Qslew/60000)/(vgSlew/2/pi/(100)^3)*60/2/pi;
Vslew = (Qslew/60000)/(vgSlew/2/pi/(100)^3)/iSlew*nv
Vluff = (boomAngle(cylinderLength(0) + Qluff/Aluff/60000*dt) ...
        - boomAngle(cylinderLength(0)))/dt
nTelescope = (Qtelescope/60000)/(vgTelescope/2/pi/(100)^3)*60/2/pi;
Vtelescope = (Qtelescope/60000)/(vgTelescope/2/pi/(100)^3) ...
             /iTelescope*rTelescopeDrum*nv

% Speed in XYZ, at home position
Vx = Vtelescope
Vy = rMin*Vslew
Vz = rMin*Vluff

% Total flow
Qtot = Qluff + Qslew + Qtelescope

% Calculate needed valve span
slewSpan = vMax/Vy
luffSpan = vMax/Vz
telescopeSpan = vMax/Vx
```

Bibliography

[1] DNV GL AS. *Offshore gangways.* DNV-RP-A204. 2017.

[2] DNV GL AS. *Qualification and assurance of digital twins.* DNV-ST-308. 2020.

[3] Beckhoff Automation. *TE1000 | TwinCAT 3 ADS.* 2021. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_ads_intro/index.html&id=7262890787652929099 (visited on 05/28/2021).

[4] M. Bak. “Model based design of electro-hydraulic motion control systems for offshore pipe handling equipment.” In: 2014.

[5] J. Denavit and R. S. Hartenberg. “A kinematic notation for lower-pair mechanisms based on matrices.” In: *Trans. ASME E, Journal of Applied Mechanics* 22 (June 1955), pp. 215–221.

[6] Thor Fossen. *Handbook of marine craft hydrodynamics and motion control.* Chichester, West Sussex, U.K. Hoboken N.J: Wiley, 2011. ISBN: 978-1-119-99149-6.

[7] Daniel Hagen, Damiano Padovani, and Martin Choux. “Design and Implementation of Pressure Feedback for Load-Carrying Applications with Position Control.” In: May 2019, pp. 520–534. URL: https://trepo.tuni.fi/handle/10024/117427.

[8] M.R. Hansen and T.O. Andersen. “Controlling a Negative Loaded Hydraulic Cylinder using Pressure Feedback.” In: *Modelling, Identification, and Control.* ACTAPRESS, 2010. DOI: 10.2316/p.2010.675-116. URL: https://doi.org/10.2316/p.2010.675-116.

[9] MICROSOFT. *.NET architectural components.* 2020. URL: https://docs.microsoft.com/en-us/dotnet/standard/components#applicable-standards (visited on 05/18/2021).

[10] Chris Paredis. *OpenHydraulics.* https://github.com/cparedis/OpenHydraulics. 2013. (Visited on 05/28/2021).

[11] Henrik C. Pedersen and Torben O. Andersen. “Pressure Feedback in Fluid Power Systems—Active Damping Explained and Exemplified.” In: *IEEE Transactions on Control Systems Technology* 26.1 (Jan. 2018), pp. 102–113. DOI: 10.1109/tcst.2017.2650680. URL: https://doi.org/10.1109/tcst.2017.2650680.

[12] J. Reinikka. *Production line simulation made with Unity and controlled by TwinCAT.* Seinäjoki, 2019. URL: https://www.theseus.fi/handle/10024/168595.

[13] Alvise Rigo, Alexander Spyridakis, and Daniel Raho. “Atomic Instruction Translation towards a Multi-threaded QEMU.” In: Europe: European Council for Modelling and Simulation, 2016. ISBN: 978-0-9932440-2-5.

[14] Harald Sangvik. *Digital Twin of 3d Motion Compensated Gangway.* Youtube. 2021. URL: https://youtu.be/dUICb9b3xEo (visited on 05/28/2021).

[15] J. K. Sørensen. “Reduction of Oscillations in Hydraulically Actuated Knuckle Boom Cranes.” PhD thesis. Grimstad, 2016.

[16] Volker Waurich and Jürgen Weber. “Interactive FMU-Based Visualization for an Early Design Experience.” In: *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017.* Linköping University Electronic Press, July 2017. DOI: 10.3384/ecp17132879. URL: https://doi.org/10.3384/ecp17132879.

[17] F. Yu. “Modeling, Simulation and Control of Motion Compensated Gangway in Offshore Operations.” MA thesis. Aalesund, 2017.