Towards to the development of a virtual environment for operators training for the ROV KAXAN from CIDESI

Thesis

TO ACHIEVE THE ACADEMIC DEGREE OF

Master of Science in Mechatronics

BY

Adrian Rivera Resendiz

Santiago de Queretaro, Qro., Mexico, February 2013

Contents

1 Introduction **11**

1.1 Problem Definition 11

1.2 Description of the project 12

1.3 Objectives 12

1.4 Structure of the thesis 13

2 Theoretical Fundamentals **14**

2.1 Marine coordinate system 14

2.2 Linear and angular velocities' transformations 15

2.3 Cinematic and Dynamic Model 17

2.3.1 The matrix M 17

2.3.2 The matrix C 19

2.3.3 The matrix D 20

2.3.4 The matrix G 21

2.3.5 Matrices simplifications 22

2.3.6 The vector τ 23

2.4 Sea Currents 24

2.5 CIDESI's KAXAN ROV 25

2.6 State of the art 26

3 Methodology **27**

3.1 Set-up and considerations for the visual environment 27

3.2 The graphic models 28

3.2.1 Creation of the models 28

3.2.2 Lighting 30

3.2.3 Sea bottom & Textures 31

3.2.4 Water surface & Blending 33

3.2.5 Fog 34

3.2.6 3D models in Wavefront format 36

3.2.7 Object loader Class . . . 38
3.3 Joystick functions . . . 42
3.4 Solution of equations (Mass-Spring-Damper) . . . 44
3.4.1 Matlab vs C# vs Real . . . 49
3.4.2 Results of P Controller . . . 53
3.4.3 Results of PI Controller . . . 56
3.4.4 Results of PID Controller . . . 59
3.4.5 Conclusions of the Mass-Spring-Damper's equations . . . 62
3.5 Thrusters . . . 62
3.5.1 Thruster 520 . . . 62
3.5.2 Thruster 540 . . . 67
3.6 Training platform (The simulator) . . . 69
3.6.1 GUI - Graphical user interface . . . 70
3.6.2 Matrices' functions class . . . 72
3.6.3 Equation's solution . . . 74
3.6.4 PID Controllers' code . . . 75
3.6.5 Normal mode and Test mode . . . 76
3.6.6 Matlab's results vs C#'s . . . 76

4 Depth PID controller **83**
4.1 Selecting K_p . . . 83
4.2 Selecting K_d . . . 85
4.3 Selecting K_i . . . 87
4.4 New controller vs. Old controller . . . 88

5 Orientation PID controller **91**
5.1 Selection K_p - Orientation controller . . . 91
5.2 Selection K_i - Orientation controller . . . 93
5.3 Selection K_d - Orientation controller . . . 94

6 Conclusions & Results **95**

6.1 Final results . . . 95
6.2 Conclusions . . . 95
6.3 Future work . . . 95

A Flow Charts **97**

A.1 Mass-Spring-Damper on C# . . . 97
A.2 Simulator on C# . . . 98

B Thrusters datasheets **99**

B.1 Model 520 . . . 99
B.2 Model 540 . . . 100

C Joystick programming **101**

C.1 Joystick structure code . . . 101
C.2 Joystick reading code . . . 101

D Matrix Operations Class **103**

D.1 Addition-Subtraction . . . 103
D.2 Multiplication of matrices . . . 104
D.3 Matrix $\times k$ . . . 104
D.4 Inverse using *Gauss-Jordan Elimination* . . . 105
D.5 Transpose . . . 107
D.6 Integration . . . 107
D.7 Matrix printing . . . 108
D.8 Matrix printing to a file . . . 108
D.9 Reading matrix from file . . . 109

E Dynamic model's matrices **110**

E.1 Matrix C_{RB} . . . 110
E.2 Matrix C_A . . . 110
E.3 Matrix D . . . 111
E.4 Matrix g . . . 111

E.5 Matrix J_1 . . . 111
E.6 Matrix J_2 . . . 112
E.7 Matrix J . . . 112
E.8 Vector $\dot{\nu}$ . . . 113

References **114**

Glossary **114**

List of Figures

1.1 KAXAN ROV . . . 11
1.2 Project schematic . . . 12
2.1 Body-fixed and earth-fixed references frames . . . 15
2.2 Thrusters on the ROV . . . 23
2.3 Diagram of ROV's computer system . . . 26
3.1 OpenTK control inside a C# Form . . . 28
3.2 ROV on Solidworks . . . 29
3.3 Oil Platform on Solidworks . . . 29
3.4 OpenGL and Texture coordinates . . . 32
3.5 Bottom of the see . . . 34
3.6 Sea surface . . . 35
3.7 Fog effect . . . 35
3.8 ROV on Blender . . . 36
3.9 Round figures using 340 polygons . . . 36
3.10 Round figures using 124 polygons, after reduction with Blender . . . 37
3.11 Dualshock 3 Sintaxis Joystick . . . 43
3.12 Axes labels . . . 44
3.13 Mass Spring Damper model in Matlab . . . 46
3.14 Mass Spring Damper model in C# . . . 46
3.15 Position . . . 50

3.16 Velocity 51
3.17 Acceleration 51
3.18 Matlab's control model 52
3.19 C#'s control model 52
3.20 Position P Controller 53
3.21 Velocity P Controller 54
3.22 Acceleration P Controller 54
3.23 Force P Controller 55
3.24 Position PI Controller 56
3.25 Velocity PI Controller 57
3.26 Acceleration PI Controller 57
3.27 Force PI Controller 58
3.28 Position PID Controller 59
3.29 Velocity PID Controller 60
3.30 Acceleration PID Controller 60
3.31 Force PID Controller 61
3.32 1 to 4 degrees polynomial (Forward mode 520) 63
3.33 5 to 8 degrees polynomial (Forward mode 520) 63
3.34 Modelled function for the 520 forward mode 64
3.35 1 to 4 degrees polynomial (Backward mode 520) 65
3.36 5 to 8 degrees polynomial (Backward mode 520) 66
3.37 Modelled function for the 520 backward mode 67
3.38 Modelled functions for the 540 69
3.39 Training platform's GUI 70
3.40 Model area 70
3.41 Sea current area 71
3.42 Depth PID area 71
3.43 ψ PID area 71
3.44 Visual environment selection 72

3.45 Status area . . . 72
3.46 Start/Stop area . . . 72
3.47 Simulink model for a constant control voltage . . . 76
3.48 Acceleration at constant force . . . 77
3.49 Velocity at constant force . . . 77
3.50 Position at constant force . . . 78
3.51 Simulink model for the Depth controller . . . 78
3.52 Acceleration using the Depth controller . . . 78
3.53 Velocity using the Depth controller . . . 79
3.54 Position using the Depth controller . . . 79
3.55 Simulink model for the Orientation controller . . . 80
3.56 Acceleration using the Orientation controller . . . 80
3.57 Velocity using the Orientation controller . . . 80
3.58 Position using the Orientation controller . . . 81
3.59 Acceleration on Normal mode . . . 81
3.60 Velocity on Normal mode . . . 82
3.61 Position on Normal mode . . . 82
4.1 K_p Rise time, Settling time & Overshot . . . 84
4.2 $K_p = 65$ . . . 85
4.3 K_d Rise time, Settling time & Overshot . . . 86
4.4 $K_p = 65$, $K_d = 30$ . . . 86
4.5 K_i Rise time, Settling time & Overshot . . . 87
4.6 $K_i = 0.5$ & $K_i = 1$ . . . 88
4.7 New vs. Old, Rise time . . . 89
4.8 New vs. Old, Settling time . . . 89
4.9 New vs. Old, Overshot . . . 90
4.10 Old controller, $K_p = 100\, K_i = 0\, K_d = 10$ . . . 90
5.1 K_p Undershot & Stationary error . . . 92
5.2 $K_p = 250$ . . . 92

5.3 K_i Undershot & No-Error 93
5.4 K_i responses 93
5.5 K_d Undershot & No-Error 94
5.6 $K_p = 250$, $K_i = 300$, $K_d = 50$ 94
A.1 Flow chart of Mass Spring Damper program on C# 97
A.2 Flow chart of the Trainings Platform 98

List of Tables

2.1 Marine vehicles' notation (SNAME notation) 14
2.2 Motion vectors 15
2.3 I_0 components 19
3.1 Position 50
3.2 Velocity 51
3.3 Acceleration 52
3.4 Position P Controller 53
3.5 Velocity P Controller 54
3.6 Acceleration P Controller 55
3.7 Force P Controller 55
3.8 Position PI Controller 56
3.9 Velocity PI Controller 57
3.10 Acceleration PI Controller 58
3.11 Force PI Controller 58
3.12 Position PID Controller 59
3.13 Velocity PID Controller 60
3.14 Acceleration PID Controller 61
3.15 Force PID Controller 61
3.16 Graphic's values from the 520 forward mode 62
3.17 Errors in $n - degree$ polynomials 64
3.18 Graphic's values from the 520 backwards mode 65

3.19 Graphic's values from the 540 forward mode 67
3.20 Graphic's values from the 540 backward mode 69
4.1 Responses for the P controller 83
4.2 Responses for the PD controller 85
4.3 Responses for the PID controller 87
4.4 Old controller vs. New controller 88
5.1 Responses for the P controller 92
5.2 Responses for the PI controller 93
5.3 Responses for the PID controller 94

Acknowledgements

First of all I would like to thank my parents because they have given me their support during my whole life including this master period.

I will also like to thank my brothers; Angel and Arturo for helping me with my presentation, to Monica for helping me in every single aspect during these years, to Gibran, Enrique and David for giving me their ideas and helping me to improve my work.

To the professors Tomas Salgado and Luis Govinda Garcia for guiding me during all the project-developing process.

And finally to professor Klaus-Peter Kämper and all the teachers in the FH-Aachen and the CIDESI, for sharing their knowledge with me.

Abstract

An ROV is a remotely operated vehicle, and learning how to drive an ROV is not an easy task, there are a lot factors that complicates the learning process, like: sea currents, low visibility while the depth is increasing, objects that cannot be seen by the main camera, and a lot more. Because an ROV is expensive, is not easy to practice how to drive one.

The CIDESI has one ROV named KAXAN, but there is no training platform in which an operator can learn how to drive it. The objective of this project is to establish the foundations on the creation of a complete training platform.

A first version of this simulator was created, the program includes the dynamics of the ROV, therefore the movements on the simulator are similar to the movements of the KAXAN in real life.

Furthermore a PID controller for the orientation and the depth of the ROV was designed.

The training platform was programmed on C#, and OpenGL was used to draw the ROV and the virtual environment.

1 Introduction

The acronym ROV in marine robotics is known as *Remote Operated Vehicle*, which refers to an underwater ship where the operator controls it remotely. There is also the AUV which are *Autonomous Underwater Vehicle*, the main difference between an ROV and the AUV, is that the AUV doesn't need an operator.

The KAXAN ROV is an unmanned submarine vehicle designed for visual exploration of shallow marine structures such as oil stations and hydroelectric centrals.

The KAXAN has been created and developed by the CIDESI, and it is remotely controlled using an umbilical cable that goes from the surface down to the ROV.

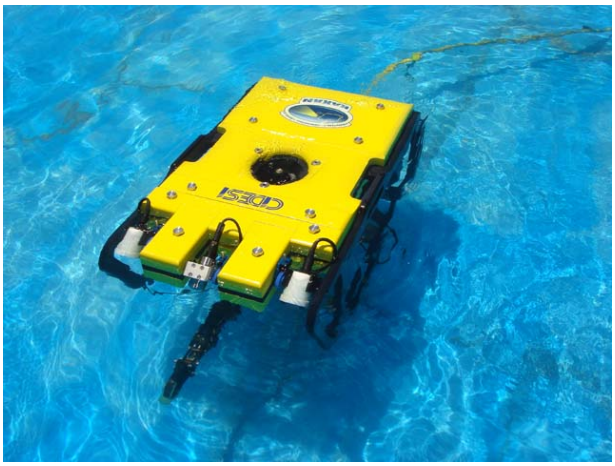


Figure 1.1: KAXAN ROV

Further information about the KAXAN can be found on section 2.5.

1.1 Problem Definition

Driving an ROV in the real world is not an easy task, because a marine environment is dark and in most cases the visibility is good just for a few meters. The operator must trust in the navigation instruments; depth sensor, compass among others.

The KAXAN is an expensive ROV, it includes a lot of specialized systems like: buoyancy foam, underwater connectors, cameras, a depth gauge, lights, thrusters, an underwater robotic arm, sensors, etc., so the user operating the ROV must control it safely in order to protect it.

Also the ROV operates near very expensive equipment like: oil tubes, hydroelectric facilities.

Currently there is not a platform in which an operator can drive the vehicle without the danger of damaging it or destroying it, so the only way in which an operator can learn how to drive the ROV is through direct operation of the vehicle under a controlled environment, like a pool.

This environment does not include real phenomenon like sea currents or the difference in density between salty water and sweet water.

1.2 Description of the project

Create the bases of a training platform for the KAXAN in which the operator will be able to learn how to drive the ROV. The program should simulate the ROV's real behaviour; effects like: inertia, Coriolis forces, sea currents, and others, must be considered.

The simulator's input device must match the one used on the KAXAN.

This simulator must use the mathematical model for the KAXAN, and the results are going to be compared with Matlab's model.

The simulator will include a graphical interface where the operator will watch an ROV's CAD model and drive it through different environments.

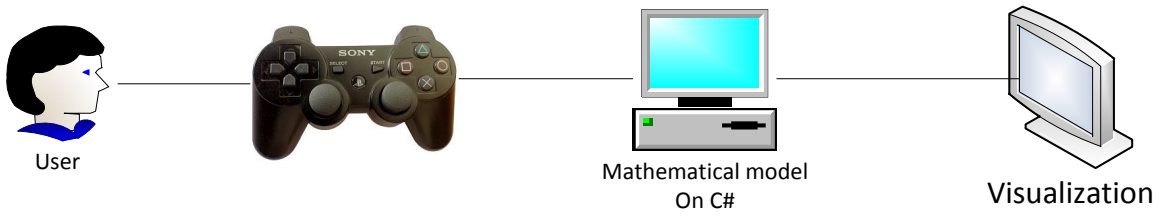


Figure 1.2: Project schematic

Also for the academic purposes of this project the depth and orientation's controller must be designed and implemented (preferably a PID control).

This simulator must be able to be run in a domestic PC, not an industrial computer, the design must be oriented to save as many computer system's resources as possible.

This project is the first step in the development of a complex training platform for possible KAXAN's operators.

1.3 Objectives

The project's main objective is:

- Establish the bases for the creation of a training platform for the KAXAN ROV, in which the users are able to get used to the ROV's manoeuvrability

The specific objectives for this project are:

- Comprehension of the KAXAN ROV's mathematical model
- Testing and familiarizing with ROV's simulator on Matlab/Simulink
- Program and solve ROV's equations using numerical methods on C#
- Development of a visual model of the ship
- Implementation of a joystick to control the ROV's model

- Add an automatic control law for the depth of the ROV
- Add an automatic control law for the direction of the ROV
- Development of a virtual submarine environment, e.g. oil platform
- Redaction of the final report

1.4 Structure of the thesis

In chapter 2 the theory about the mathematical modelling of marine vehicles is discussed.

In chapter 3 are the specifics about how the program was created, which initial considerations were made before programming, and which tests were made to validate the results.

Chapters 4 and 5 talk about the design, considerations, tests and results of the PID controllers used in this project.

And finally in chapter 6 are the conclusions and achieved goals of this project, and the future work that expands upon this project.

2 Theoretical Fundamentals

Mathematical modelling of marine vehicles is a complex task because several factors must be taken in consideration, for example:

- The vehicle's shape
- If the vehicle is immerse, high speed or a surface ship.
- Sea currents
- The ocean swell

The basic principles about marine vehicles modelling can be found on Fossen's books, [5] and [4].

Even though modelling the KAXAN ROV's mathematical model is not the goal of this project, it is important to study about how the marine vehicles are modelled and how to explain the different phenomenons at which the ROV is submitted. The model used in this project was already calculated at the institute.

2.1 Marine coordinate system

To determinate the position and orientation of a marine vehicle is required to know six independent coordinates, therefore is a 6 DOF (degrees of freedom) system, the first 3 degrees are translational coordinates and the other 3 are rotational coordinates (Table 2.1).

DOF	Name	Description	Forces and moments	Linear and angular vel.	Positions and Euler angles
1	Surge	Motion in x-direction	X	u	x
2	Sway	Motion in y-direction	Y	v	y
3	Heave	Motion in z-direction	Z	w	z
4	Roll	Rotation about the x-axis	K	p	ϕ
5	Pitch	Rotation about the y-axis	M	q	θ
6	Yaw	Rotation about the z-axis	N	r	ψ

Table 2.1: Marine vehicles' notation (SNAME notation)

When analysing a marine vehicle's movement is convenient to define 2 different coordinate systems, the first one attached to the vehicle, and the second one fixed to some point on space.

The first coordinate system is known as *body-fixed* and it's formed by: X_0 or longitudinal axis directed from aft to fore Y_0 or transverse axis directed to starboard, and finally Z_0 or normal axis directed from top to bottom. Generally the origin of *body-fixed* axis coincides with the vehicle's gravity center, but it can be placed on any convenient point.

The second coordinate system is know as *earth-fixed*, the figure 2.1 shows both frames.

By convention the position and orientation are expressed relative to the *earth-fixed* frame, and the linear and angular velocities referred to the *body-fixed* frame.

Based on SNAME notation (Table 2.1) and the considerations above the vehicle's general motion can be described with the vectors in table 2.2.

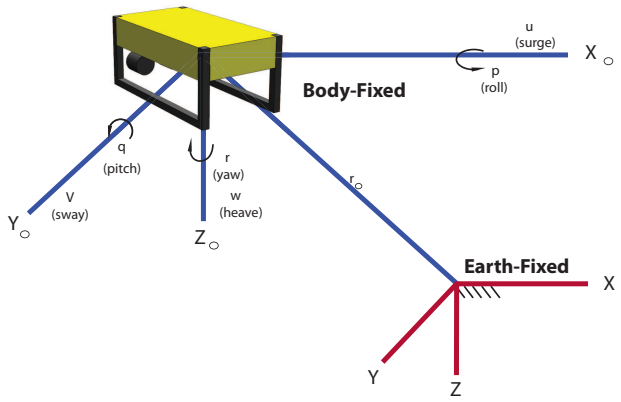


Figure 2.1: Body-fixed and earth-fixed references frames

$$\eta = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} \quad \eta_1 = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \eta_2 = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}$$

$$\nu = \begin{bmatrix} \nu_1 \\ \nu_2 \end{bmatrix} \quad \nu_1 = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad \nu_2 = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\tau = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} \quad \tau_1 = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad \tau_2 = \begin{bmatrix} K \\ M \\ N \end{bmatrix}$$

Table 2.2: Motion vectors

The vector η represents the vehicle's position referred to *earth-fixed* frame, the vector ν is the velocity vector referred to the *body-fixed* frame, and finally τ is the force vector also referred to *body-fixed* frame.

2.2 Linear and angular velocities' transformations

Even thought the velocity is referred to the *body-fixed* frame, is mandatory to make the transformation of this velocity vector to *earth-fixed* system.

Since both frames are not pointing in the same direction, some rotations must be applied, for example the matrices' transformation due to rotation in X, Y and Z is given by:

$$C_{x,\phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \tag{2.1}$$

$$C_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.2}$$

$$C_{z,\psi} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

If the transposes of 2.3, 2.2 and 2.1 are multiplied a new matrix is obtained:

$$J_1(\eta_2) = C_{z,\psi}^T \, C_{y,\theta}^T \, C_{x,\phi}^T$$

$$J_1(\eta_2) = \begin{bmatrix} \cos\psi\,\cos\theta & \cos\psi\,\sin\phi\,\sin\theta - \cos\phi\,\sin\psi & \sin\phi\,\sin\psi + \cos\phi\,\cos\psi\,\sin\theta \\ \cos\theta\,\sin\psi & \cos\phi\,\cos\psi + \sin\phi\,\sin\psi\,\sin\theta & \cos\phi\,\sin\psi\,\sin\theta - \cos\psi\,\sin\phi \\ -\sin\theta & \cos\theta\,\sin\phi & \cos\phi\,\cos\theta \end{bmatrix} \tag{2.4}$$

$J_1(\eta_2)$ (Eq. 2.4) is the matrix capable of transforming the data on X_0, Y_0 and Z_0 from *body-fixed* frame to the *earth-fixed* frame.

Now the transformation from vector ν_1 to $\dot{\eta}_1$ is defined by:

$$\dot{\eta}_1 = J_1(\eta_2)\,\nu_1 \tag{2.5}$$

Let's define $\dot{\eta}_2$ as $\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$, and:

$$\dot{\eta}_2 = J_2(\eta_2)\,\nu_2 \tag{2.6}$$

The problem is that $J_2(\eta_2)$ is not defined, but if vector ν_2 is cleared, then:

$$\nu_2 = J_2^{-1}(\eta_2)\,\dot{\eta}_2 = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + C_{x,\phi} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + C_{x,\phi}C_{y,\theta} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \dot{\phi} - \dot{\psi}\,\sin\theta \\ \dot{\theta}\,\cos\phi + \dot{\psi}\,\cos\theta\,\sin\phi \\ \dot{\psi}\,\cos\phi\,\cos\theta - \dot{\theta}\,\sin\phi \end{bmatrix} \tag{2.7}$$

From equation 2.7, $J_2^{-1}(\eta_2)$ can be deduced:

$$J_2^{-1}(\eta_2) = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \cos\theta\,\sin\phi \\ 0 & -\sin\phi & \cos\phi\,\cos\theta \end{bmatrix} \therefore$$

$$J_2(\eta_2) = \begin{bmatrix} 1 & \sin\phi\,\tan\theta & \cos\phi\,\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix} \tag{2.8}$$

Combining 2.4 and 2.8 the matrix $J(\eta_2)$ can be written as:

$$\dot{\eta}_1 = J_1(\eta_2)\,\nu_1$$

$$\dot{\eta}_2 = J_2(\eta_2)\,\nu_2$$

$$J = \begin{bmatrix} J_1 & 0_{3\times3} \\ 0_{3\times3} & J_2 \end{bmatrix}$$

$$J(\eta_2) = \begin{bmatrix} c\,\psi\,c\,\theta & c\,\psi\,s\,\phi\,s\,\theta - c\,\phi\,s\,\psi & s\,\phi\,s\,\psi + c\,\phi\,c\,\psi\,s\,\theta & 0 & 0 & 0 \\ c\,\theta\,s\,\psi & c\,\phi\,c\,\psi + s\,\phi\,s\,\psi\,s\,\theta & c\,\phi\,s\,\psi\,s\,\theta - c\,\psi\,a\,\phi & 0 & 0 & 0 \\ -s\,\theta & c\,\theta\,s\,\phi & c\,\phi\,c\,\theta & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & s\,\phi\,t\,\theta & c\,\phi\,t\,\theta \\ 0 & 0 & 0 & 0 & c\,\phi & -s\,\phi \\ 0 & 0 & 0 & 0 & \dfrac{s\,\phi}{c\,\theta} & \dfrac{c\,\phi}{c\,\theta} \end{bmatrix}$$

Whit $c\,\blacksquare = cos(\blacksquare)$, $s\,\blacksquare = sin(\blacksquare)$ and $t\,\blacksquare = tan(\blacksquare)$

$$\dot{\eta} = J(\eta_2)\,\nu \tag{2.9}$$

The vehicle's movement vector relative to the *earth-fixed* coordinate system, is given by the velocity vector 2.9, where $J(\eta_2)$ is a transformation matrix referred to the Euler angles (roll, pitch and yaw).

If a closer look to $J_2(\eta_2)$, at $\theta \approx \pm 90°$ a singularity is presented, therefore this mode of operation must be avoided, but the KAXAN is a very stable ROV, and this vertical position is never reached by the KAXAN.

2.3 Cinematic and Dynamic Model

The ROV's dynamics can be expressed with the non-linear equation:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) = \tau \tag{2.10}$$

where:

- M = inertia matrix (including added mass)
- $C(\nu)$ = Coriolis and centripetal matrix (including added mass)
- D = damping matrix
- g = gravitational forces and moments' vector
- τ = forces and moments' vector

If we reduce the equation 2.10 to:

$$M\,\dot{\nu} = \tau$$

we get a equation similar to

$$m\,a = f$$

which is the second law of Newton. This function relates the mass, the acceleration and the force into one equation. If no force is acting on the system then it remains with a constant speed or at body rest.

If we also take in consideration the Euler's Axioms we can relate both forces and moments referred to the body's center of mass:

$$\dot{p}_c = f_c$$
$$\dot{h}_c = m_c$$
$$\dot{p}_c = M\,\dot{\nu}_c$$
$$\dot{h}_c = I\,\dot{\omega}_c$$

where $\dot{p}_c$ represent the forces on the body, $\dot{h}_c$ the moments, M the mass, I the matrix, $\dot{\nu}_c$ the linear acceleration and $\dot{\omega}_c$ the angular acceleration, these equations usage is usually known as *vectorial mechanics*.

2.3.1 The matrix M

The matrix M is formed by two different matrices,

$$M = M_{RB} + M_A$$

where M_{RB} is the rigid-body inertia matrix, and $M - A$ is the added inertia matrix, this M_A should be understood as the pressure-induced forces and moments due to a forced motion of the body proportional to the acceleration of the body. M_{RB} and M_A matrices are defined bellow[1].

$$M_{RB} = \begin{bmatrix} mI_{3\times3} & -mS(r_G) \\ mS(r_G) & I_0 \end{bmatrix} = \begin{bmatrix} m & 0 & 0 & 0 & mz_G & -my_G \\ 0 & m & 0 & -mz_G & 0 & mx_G \\ 0 & 0 & m & my_G & mx_G & 0 \\ 0 & -mz_G & my_G & I_x & -I_{xy} & -I_{xz} \\ mz_G & 0 & -mx_G & -I_{yx} & I_y & -I_{yz} \\ -my_G & mx_G & 0 & -I_{zx} & -I_{zy} & I_z \end{bmatrix} \tag{2.11}$$

$$M_A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = -\begin{bmatrix} X_{\dot{u}} & X_{\dot{v}} & X_{\dot{w}} & X_{\dot{p}} & X_{\dot{q}} & X_{\dot{r}} \\ Y_{\dot{u}} & Y_{\dot{v}} & Y_{\dot{w}} & Y_{\dot{p}} & Y_{\dot{q}} & Y_{\dot{r}} \\ Z_{\dot{u}} & Z_{\dot{v}} & Z_{\dot{w}} & Z_{\dot{p}} & Z_{\dot{q}} & Z_{\dot{r}} \\ K_{\dot{u}} & K_{\dot{v}} & K_{\dot{w}} & K_{\dot{p}} & K_{\dot{q}} & K_{\dot{r}} \\ M_{\dot{u}} & M_{\dot{v}} & M_{\dot{w}} & M_{\dot{p}} & M_{\dot{q}} & M_{\dot{r}} \\ N_{\dot{u}} & N_{\dot{v}} & N_{\dot{w}} & N_{\dot{p}} & N_{\dot{q}} & N_{\dot{r}} \end{bmatrix}$$

where m is the mass, x_G, y_G and z_G are the coordinates of the center of mass referred to the *body-fixed* frame, the I components the inertia tensors with respect to the origin of the body frame, and the M_A components are the added mass force along one axis due to a certain acceleration in one direction. S is a *Skew-symmetry* matrix, where $S = -S^T$ and it is defined as:

$$S(\lambda) = \begin{bmatrix} 0 & -\lambda_3 & \lambda_2 \\ \lambda_3 & 0 & -\lambda_1 \\ -\lambda_2 & \lambda_1 & 0 \end{bmatrix} \tag{2.12}$$

To calculate M_{RB} is necessary to calculate the center of mass, which is given by the formula

$$r_G = \frac{1}{m}\int_V r\,\rho_A dV$$

where:

- r_G is the vector of position referred to *body-fixed* coordinate system
- r is the position vector referred to *earth-fixed* coordinate system
- ρ_A is the density of the vehicle's mass
- m is the body mass
- V is the displaced fluid's volume

In practice is easier to calculate the center of mass with an specialized software, like Solidworks.

The inertia tensor (I_0) is the equivalent to the mass in the case of unidimensional movement, but in tridimentional movement, the opposition to rotation movement appears in several directions. The inertia matrix is defined by

$$I_0 = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix}$$

[1] [4, p. 26,33]

and each component is expressed like it is shown in table 2.3

$$\begin{array}{ll} I_x = \int_v (y^2+z^2)\,\rho_A\,dV & I_{xy} = \int_v xy\,\rho_A\,dV = \int_v yx\,\rho_A\,dV = I_{yx} \\ I_y = \int_v (x^2+z^2)\,\rho_A\,dV & I_{xz} = \int_v xz\,\rho_A\,dV = \int_v zx\,\rho_A\,dV = I_{zx} \\ I_z = \int_v (x^2+y^2)\,\rho_A\,dV & I_{yz} = \int_v yz\,\rho_A\,dV = \int_v zy\,\rho_A\,dV = I_{zy} \end{array}$$

Table 2.3: I_0 components

therefore we can say that $I_0 = I_0^T > 0$

Depending on the shape of the vehicle's submerged part, different formulas can be used to calculate the matrix coefficients M_A, but for irregular or complex shapes these values must be calculated by experimentation or with specialized software, in the case of the KAXAN ROV an specialized software was used.

For vehicles completely submerged M_A will be always strictly positive, that means $M_A > 0$. If $M_A > 0$ then $M_A = M_A^T > 0$ is a good approximation[2], then:

$$M_A = -\begin{bmatrix} X_{\dot{u}} & X_{\dot{v}} & X_{\dot{w}} & X_{\dot{p}} & X_{\dot{q}} & X_{\dot{r}} \\ X_{\dot{v}} & Y_{\dot{v}} & Y_{\dot{w}} & Y_{\dot{p}} & Y_{\dot{q}} & Y_{\dot{r}} \\ X_{\dot{w}} & Y_{\dot{w}} & Z_{\dot{w}} & Z_{\dot{p}} & Z_{\dot{q}} & Z_{\dot{r}} \\ X_{\dot{p}} & Y_{\dot{p}} & Z_{\dot{p}} & K_{\dot{p}} & K_{\dot{q}} & K_{\dot{r}} \\ X_{\dot{q}} & Y_{\dot{q}} & Z_{\dot{q}} & K_{\dot{q}} & M_{\dot{q}} & M_{\dot{r}} \\ X_{\dot{r}} & Y_{\dot{r}} & Z_{\dot{r}} & K_{\dot{r}} & M_{\dot{r}} & N_{\dot{r}} \end{bmatrix} \tag{2.13}$$

After defining M_{RB} and M_A now we can proceed to define the value of M:

$$M = M_{RB} + M_A$$

$$M = \begin{bmatrix} m - X_{\dot{u}} & -X_{\dot{v}} & -X_{\dot{w}} & -X_{\dot{p}} & mz_G - X_{\dot{q}} & -X_{\dot{r}} - my_G \\ -X_{\dot{v}} & m - Y_{\dot{v}} & -Y_{\dot{w}} & -Y_{\dot{p}} - mz_G & -Y_{\dot{q}} & mx_G - Y_{\dot{r}} \\ -X_{\dot{w}} & -Y_{\dot{w}} & m - Z_{\dot{w}} & my_G - Z_{\dot{p}} & mx_G - Z_{\dot{q}} & -Z_{\dot{r}} \\ -X_{\dot{p}} & -Y_{\dot{p}} - mz_G & my_G - Z_{\dot{p}} & I_x - K_{\dot{p}} & -I_{xy} - K_{\dot{q}} & -I_{xz} - K_{\dot{r}} \\ mz_G - X_{\dot{q}} & -Y_{\dot{q}} & -Z_{\dot{q}} - mx_G & -I_{yx} - K_{\dot{q}} & I_y - M_{\dot{q}} & -I_{yz} - M_{\dot{r}} \\ -X_{\dot{r}} - my_G & mx_G - Y_{\dot{r}} & -Z_{\dot{r}} & -I_{zx} - K_{\dot{r}} & -I_{zy} - M_{\dot{r}} & I_z - N_{\dot{r}} \end{bmatrix} \tag{2.14}$$

2.3.2 The matrix C

The matrix C is the Coriolis and Centripetal matrix, this C matrix is also formed by $C_{RB}(\nu) + C_A(\nu)$, where $C_{RB}(\nu)$ is the rigid-body matrix of Coriolis and centripetal forces and $C_A(\nu)$ the added mass Coriolis matrix. Matrix $C_{RB}(\nu)$ can be defined[3] as:

$$C_{RB}(\nu) = \begin{bmatrix} 0_{3\times 3} & -mS(\nu_1) - mS(S(\nu_2)r_G) \\ -mS(\nu_1) - mS(S(\nu_2)r_G) & mS(S(\nu_1)r_G) - S(I_0\nu_2) \end{bmatrix}$$

$$C_{RB}(\nu) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -m(y_G q + z_G r) & m(y_G p + w) & m(z_G p - v) \\ m(x_G q - w) & -m(z_G r + x_G p) & m(z_G q + u) \\ m(x_G r + v) & m(y_G r - u) & -m(x_G p + y_G q)) \end{bmatrix}$$

[2][4, p. 34]
[3][4, p. 28]

$$\begin{bmatrix} m(y_G q + z_G r) & -m(x_G q - w) & -m(x_G r + v) \\ -m(y_G p + w) & m(z_G r + x_G p) & -m(y_G r - u) \\ -m(z_G p - v) & -m(z_G q + u) & m(x_G q + y_G q) \\ 0 & -I_{yz} q - I_{xz} p - I_z r & I_{yz} r + I_{xy} p - I_y q \\ I_{yz} q + I_{xz} p - I_z r & 0 & -I_{xz} r - I_{xy} q - I_x p \\ -I_{yz} r - I_{xy} p - I_y q & I_{xz} r + I_{xy} q - I_x p & 0 \end{bmatrix} \tag{2.15}$$

Fossen writes in his book that if a rigid-body is moving through an ideal fluid, then the hydrodynamic Coriolis and centripetal matrix $C_A(\nu)$ can always be parametrized such that $C_A(\nu)$ is skew-symmetrical, $C_A(\nu) = -C_A^T(\nu)$[4], by defining:

$$C_A(\nu) = \begin{bmatrix} 0_{3\times 3} & -S(A_{11}\nu_1 + A_{12}\nu_2) \\ -S(A_{11}\nu_1 + A_{12}\nu_2) & -S(A_{21}\nu_1 + A_{22}\nu_2) \end{bmatrix}$$

$$C_A(\nu) = \begin{bmatrix} 0 & 0 & 0 & 0 & -a_3 & a_2 \\ 0 & 0 & 0 & a_3 & 0 & -a_1 \\ 0 & 0 & 0 & -a_2 & a_1 & 0 \\ 0 & -a_3 & a_2 & 0 & -b_3 & b_2 \\ a_3 & 0 & -a_1 & b_3 & 0 & -b_1 \\ -a_2 & a_1 & 0 & -b_2 & b_1 & 0 \end{bmatrix}$$

whit

$$\begin{aligned} a_1 &= X_{\dot{u}} u + X_{\dot{v}} v + X_{\dot{w}} w + X_{\dot{p}} p + X_{\dot{q}} q + X_{\dot{r}} r \\ a_2 &= X_{\dot{v}} u + Y_{\dot{v}} v + Y_{\dot{w}} w + Y_{\dot{p}} p + Y_{\dot{q}} q + Y_{\dot{r}} r \\ a_3 &= X_{\dot{w}} u + Y_{\dot{w}} v + Z_{\dot{w}} w + Z_{\dot{p}} p + Z_{\dot{q}} q + Z_{\dot{r}} r \\ b_1 &= X_{\dot{p}} u + Y_{\dot{p}} v + Z_{\dot{p}} w + K_{\dot{p}} p + K_{\dot{q}} q + K_{\dot{r}} r \\ b_2 &= X_{\dot{q}} u + Y_{\dot{q}} v + Z_{\dot{q}} w + K_{\dot{q}} p + M_{\dot{q}} q + M_{\dot{r}} r \\ b_3 &= X_{\dot{r}} u + Y_{\dot{r}} v + Z_{\dot{r}} w + K_{\dot{r}} p + M_{\dot{r}} q + N_{\dot{r}} r \end{aligned}$$

2.3.3 The matrix D

The matrix D is known as the hydrodynamic damping, this effect can be caused by several reasons:

- $D_P(\nu)$ radiation-induced potential damping due to forced body oscillations
- $D_S(\nu)$ linear skin friction due to laminar boundary layers and quadratic skin friction due to turbulent boundary layers
- $D_W(\nu)$ wave drift damping
- $D_M(\nu)$ damping due to vortex shedding

The damping matrix $D(\nu)$ can be written as a sum of all these components:

$$D(\nu) = D_P(\nu) + D_S(\nu) + D_W(\nu) + D_M(\nu)$$

If a rigid-body is moving through an ideal fluid then the hydrodynamic damping is real, non-symmetrical and strictly positive, therefore:

$$D(\nu) > 0$$

[4] [4, p. 36]

Potential damping $D_P(\nu)$ is also known as radiation-induced damping, and it is product of the oscillation produced by the waves. For underwater vehicles operating at great depths the potential damping can be neglected, but for a surface vehicle the potential damping effect may be significant. The radiation induced forces and moments can be written as:

$$\tau_R = -A(\omega)\ddot{\eta} - B(\omega)\dot{\eta} - C\,\eta$$

where $A = -M_A$ is the added inertia matrix, $B = -D_P$ represents a linear potential damping, C is the linearized restoring forces and moments, and finally ω is the wave circular frequency.

Skin friction is important when considering a low-frequency motion of the vehicle, due to laminar boundary layer theory. This is usually referred as a quadratic or non-linear skin friction.

Wave drift damping can be interpreted as added resistance for surface vessels advancing in waves. For higher sea states, the wave drift damping is the most important contribution to surge, due to the fact that this damping forces are proportional to the square of the significant wave height. Wave drift damping in sway and yaw is small relative to eddy making damping (vortex shedding).

Damping Due to Vortex Shedding is a force produced by frictional forces caused by the movement of one object through a viscous fluid. This viscous damping force can be modelled as:

$$f(U) = -\frac{1}{2}\rho C_D(Rn) A|U|U$$

where U is the velocity of the vehicle, A is the cross-sectional area, $C_D(Rn)$ is the drag-coefficient based on the area and ρ is the density of the water. The drag coefficient $C_D(Rn)$ depends on:

$$Rn = \frac{U\,D}{\nu}$$

which is known as the *Reynolds number*, where D is the characteristic length of the body and ν the coefficient of kinematic viscosity ($\nu = 1.56 \cdot 10^{-6}$ for salt water at 5°C salinity 3.5%). Finally the quadratic drag in 6 DOF is expressed as:

$$D_M(\nu)\nu = \begin{bmatrix} |\nu|^T D_1 \nu \\ |\nu|^T D_2 \nu \\ |\nu|^T D_3 \nu \\ |\nu|^T D_4 \nu \\ |\nu|^T D_5 \nu \\ |\nu|^T D_6 \nu \end{bmatrix}$$

It's important to mention that C_D and A will be different for each matrix' element.

2.3.4 The matrix G

The restoring forces and moments are represented by the matrix g, these forces are product of the buoyant and gravitational forces. The gravitational force f_c acts at the center of gravity of the body (r_G) and the buoyant force f_B acts through the center of buoyancy $r_B = [x_B, y_B, z_B]^T$.

To calculate these forces on an underwater vehicle, first we must take in consideration the mass m of the object to calculate the gravitational force W on it, this submerged weight is defined as: $W = mg$,where g is the density, and using the equation 2.4, f_G can be defined as:

$$f_G(\eta_2) = J_1^{-1}(\eta_2) \begin{bmatrix} 0 \\ 0 \\ W \end{bmatrix}$$

The buoyancy force is defined as: $B = \rho g \bigtriangledown$, where $\bigtriangledown$ is the volume of fluid displaced by the body and ρ the density of the water, finally f_B is defined as:

$$f_B(\eta_2) = J_1^{-1}(\eta_2) \begin{bmatrix} 0 \\ 0 \\ B \end{bmatrix}$$

With f_G and f_B the matrix $g(\eta)$ can be written as:

$$g(\eta) = -\begin{bmatrix} f_G(\eta) + f_B(\eta) \\ r_G \times f_G(\eta) + r_B \times f_B(\eta) \end{bmatrix}$$

$$g(\eta) = \begin{bmatrix} (W-B)\sin\theta \\ -(W-B)\cos\theta\sin\phi \\ -(W-B)\cos\theta\cos\phi \\ -(y_G W - y_B B)\cos\theta\cos\phi + (z_G W - z_B B)\cos\theta\sin\phi \\ (z_G W - z_B B)\sin\theta - (x_G W - x_B B)\cos\theta\cos\phi \\ -(x_G W - x_B B)\cos\theta\sin\phi - (y_G W - y_B B)\sin\theta \end{bmatrix} \tag{2.16}$$

In the case on a neutrally buoyant underwater vehicle, like the KAXAN, $W = B$, therefore:

$$g(\eta) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\overline{BG}_y W\cos\theta\cos\phi + \overline{BG}_z W\cos\theta\sin\phi \\ \overline{BG}_z W\sin\theta + \overline{BG}_x W\cos\theta\cos\phi \\ -\overline{BG}_x W\cos\theta\sin\phi - \overline{BG}_y W\sin\theta \end{bmatrix}$$

where $\overline{BG} = [x_G - x_B, y_G - y_B, z_G - z_B]^T$.

2.3.5 Matrices simplifications

In the past sections all the values of *M, C, D and g* have been explained and these value were written as a general expression of each matrix, but under certain circumstances these matrices can be simplified, therefore their complexity will be reduced and this will reduce the use of CPU's resources, making the program faster and lighter.

As said before in section 2.1 the center of the *body-fixed* frame can be placed at convenience, if this origin is placed in the same point as the center of mass r_G then the matrix I_0 is simplified[5] to be:

$$I_0 = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

therefore

$$M_{RB} = diag\{m, m, m, I_x, I_y, I_z\}$$

Now let's simplify matrices M_A and $C_A(\nu)$. In general the motion of an underwater vehicle moving in 6 DOF at high speed is highly non-linear and coupled, but generally an ROV doesn't move at high speed, only at low speed, also if the vehicle has 3 planes of symmetry is possible to neglect the effect from the

[5] [4, p. 28]

off-diagonal elements in the added mass matrix M_A, therefore the matrix $C_A(\nu)$ is automatically also simplified:

$$M_A = -diag\{X_{\dot{u}}, Y_{\dot{v}}, Z_{\dot{w}}, K_{\dot{p}}, M_{\dot{q}}, N_{\dot{r}}\}$$

$$C_A(\nu) = \begin{bmatrix} 0 & 0 & 0 & 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v \\ 0 & 0 & 0 & Z_{\dot{w}}w & 0 & -X_{\dot{u}}u \\ 0 & 0 & 0 & -Y_{\dot{v}}v & X_{\dot{u}}u & 0 \\ 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v & 0 & -N_{\dot{r}}r & M_{\dot{q}}q \\ Z_{\dot{w}}w & 0 & -X_{\dot{u}}u & N_{\dot{r}}r & 0 & -K_{\dot{p}}p \\ -Y_{\dot{v}}v & X_{\dot{u}}u & 0 & -M_{\dot{q}}q & K_{\dot{p}}p & 0 \end{bmatrix}$$

The same reasons used to simplify M_A applied to $D(\nu)$, where the damping matrix will only contain the diagonal structure with the linear and quadratic damping terms on the diagonal:

$$\begin{aligned} D(\nu) &= -diag\{X_u, Y_v, Z_w, K_p, M_q, N_r\} - diag\{X_{u|u|}|u|, Y_{v|v|}|v|, Z_{w|w|}|w|, K_{p|p|}|p|, M_{q|q|}|q|, N_{r|r|}|r|\} \\ &= -diag\{X_u + X_{u|u|}|u|, Y_v + Y_{v|v|}|v|, Z_w + Z_{w|w|}|w|, K_p + K_{p|p|}|p|, M_q + M_{q|q|}|q|, N_r + N_{r|r|}|r|\} \end{aligned}$$

2.3.6 The vector τ

The forces and moments applied to the ROV come from the thrusters installed on it, these thrusters are the active forces on the system and they produce the values on vector τ. The thrusters are placed as shown on the figure 2.2.

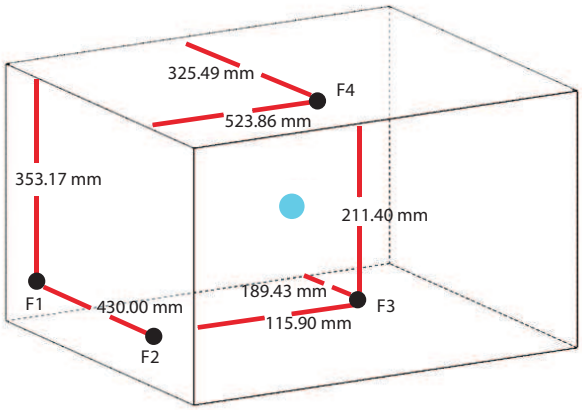


Figure 2.2: Thrusters on the ROV

There are 4 thrusters placed on the ROV:

- F1 - Back Left thruster
- F2 - Back Right thruster
- F3 - Side thruster
- F4 - Vertical thruster

The vector τ is given by:

$$\tau = \begin{bmatrix} X \\ Y \\ Z \\ K \\ M \\ N \end{bmatrix} = B \begin{bmatrix} F1 \\ F2 \\ F3 \\ F4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -D_{3z} & D_{4y} \\ D_{1z} & D_{2z} & 0 & -D_{4x} \\ -D_{1y} & -D_{2y} & D_{3x} & 0 \end{bmatrix} \begin{bmatrix} F1 \\ F2 \\ F3 \\ F4 \end{bmatrix} \tag{2.17}$$

where B is a matrix of 6×4 which distributes the action of the thrusters on the different components of τ (X,Y,Z,K,M *and* N). B is formed by D_{ij} which is the distance between each thruster to the center of mass r_G.

After defining the vector τ, all the values of $M\dot{\nu}+C(\nu)\nu+D(\nu)\nu+g(\eta) = \tau$ (Eq. 2.10) has been explained.

2.4 Sea Currents

One important effect of the ocean that affects every marine vehicle are the sea currents. The sea currents have been used for centuries as routes through the ocean. The sea currents are caused by different effects like: gravity, wind friction, water density variations in different parts of the ocean and thermohaline currents which are produces by heat exchange at the sea surface together with salinity changes.

To model ocean currents and their effects on vessels motion, a new model must be applied:

$$M_{RB}\dot{\nu} + C_{RB}(\nu)\nu + g(\eta) + M_A\dot{\nu} + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r = \tau$$

where $M_{RB}\dot{\nu} + C_{RB}(\nu)\nu + g(\eta)$ are the rigid-body terms, $M_A\dot{\nu} + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r$ the hydrodynamic terms, and $\nu_r = \nu - \nu_c$ the current velocity vector, which is assumed to slowly-varing, producing a $\dot{\nu}_c \approx 0$.

Finally the result is:

$$M\dot{\nu} + C_{RB}(\nu)\nu + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r + g(\eta) = \tau \tag{2.18}$$

Current speed and direction is determined by vector ν_c, but the speed is noted as V_c, while its direction relative to the moving vessel is expressed in terms of the angle of attack α and the sideslip angle β.

Assuming an irrotational fluid, the 3D current model is given by:

$$\begin{bmatrix} u_c^E \\ v_c^E \\ w_c^E \end{bmatrix} = R_{y,\alpha_c}^T \, R_{z,-\beta_c}^T \begin{bmatrix} V_c \\ 0 \\ 0 \end{bmatrix}$$

with:

$$R_{y,\alpha_c}^T = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$R_{z,-\beta_c}^T = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

this results in:

$$\begin{aligned} u_c^E &= V_c \cos\alpha_c \cos\beta_c \\ v_c^E &= V_c \sin\beta_c \\ w_c^E &= V_c \sin\alpha_c \cos\beta_c \end{aligned}$$

Now we have the vector ν_c^E which is the current velocity vector referred to the *earth-fixed* frame. If we apply a rotation with the transformation matrix $J(\eta)$ the velocity vector is obtained:

$$\nu_c = J^{-1}(\eta)\nu_c^E \tag{2.19}$$

It is important to remark that the equation 2.18 is the formula used in all the calculations of the simulator.

2.5 CIDESI's KAXAN ROV

The center for engineering and industrial development also known as CIDESI, belongs to the national council of science and technology (CONACYT, Mexico). The CIDESI has two main missions: First, generating value in companies and enhance their competitiveness by the development and application of knowledge both relevant and pertinent. The second is promoting Mexican technology development.

In the Mexican oil and power industries there are great potential applications of underwater vehicles, mainly on the inspection, maintenance and repair of underwater structures, specially in deep waters not easily accessible to humans.

Because of these reasons CIDESI works in the development of an ROV suitable for the necessities of Mexican's oil and power industries. The current version of this ROV is a *Shallow Water ROV* named *KAXAN*, which at Mexican Maya language means *the seeker*. The article [14] presents the considerations, characteristics and specifics about the design of the KAXAN ROV.

Some characteristics about the KAXAN are:

- Weight: 108kg
- Dimensions: length 1.1m × width 0.65m × height 0.5m
- Maximum operation depth: 120m
- Sensors: Depth sensor and compass
- Thrusters: 1 vertical, 1 lateral and 2 at the back
- 2 underwater lamps
- Cameras: 1 on the front
- Electric manipulator: 3 functions manipulator, only for sample recollection, maximum sample weight:5kg

The ROV's elements and subsystems were designed and drawn using Solidworks, other critical elements, for example the electronic container, were analysed using the software of finite element: Ansys.

Using specialized software the hydrostatic robot parameters were determinated; weight, buoyancy, gravity center, flotation center and meta-centric height.

The ROV's electronic architecture consists of two computers, the first one located at the surface and the second inside the ROV's electronic container, these computers communicate by an Ethernet protocol using an umbilical cable with one optic fibre and two optic fibre's multiplexer. The surface computer includes a screen where the user can see the images taken by the ROV's camera, with this computer the user is able to move the vehicle using a joystick. The electronic architecture is monitored and controlled by an application programmed in Labview, showing the system GUI.

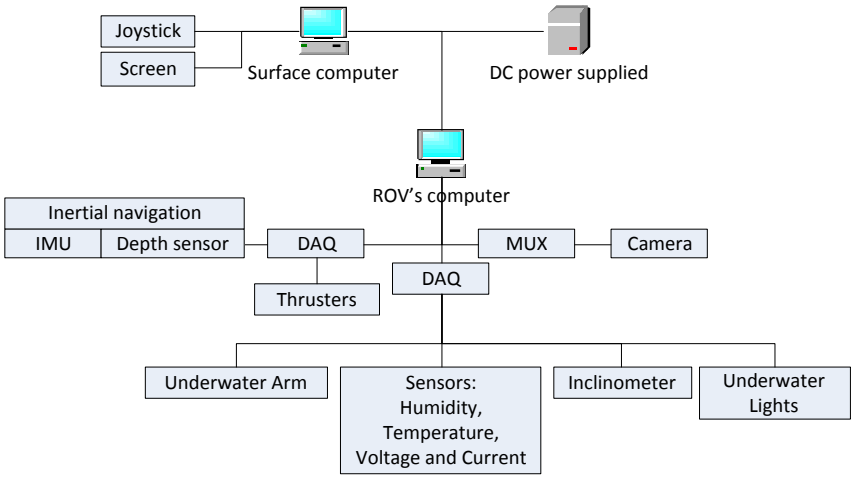


Figure 2.3: Diagram of ROV's computer system

The ROV counts with an specialized vision system capable of identifying patrons and doing object's measurements, with a precision of 95% in a range from 0.2m to 0.45m during laboratory experiments, this kind of equipment is very helpful during underwater inspections.

2.6 State of the art

There exists a big number of delicate and expensive equipment which must be used carefully, there also exist equipment which needs very special environmental conditions to work with, and to reproduce these effects in a controlled environment is not suitable in every case, due to these reasons the use of simulators is a necessity in several applications.

A lot of simulation platforms have been implemented both for fun or with scientific purposes.

A well known game that acts as a simulator is a flight simulator from Microsoft which recreates several different flying conditions, and the user can fly through different environments using a rich number of aircrafts. If the user counts with a joystick in his computer, the sensation of manoeuvring the airplane with a shaft can be simulated.

A more scientific example is a training platform for a large tactical communication equipment [1], where the real object is designed to work under war conditions, as one can image these environmental conditions are not easily reproduced, therefore a training platform is required to teach the operators how to interact with the machine and the different weather conditions and special phenomenons.

A simulation platform for a ship's propulsion plant has also been created [15], in this simulator the non-linear mathematical model of the propulsion systems has been programmed. The engine is a large low speed, two-stroke diesel motor, the mathematical model also includes the shaft, a fixed pitch propeller and ship hull dynamics. The navigation condition of the main propulsion of the ship can be properly simulated and a good visualization of the engine is gained by the system. This system can adopted both, training facility under laboratory conditions and a platform of intelligent control algorithm for marine main diesel propulsion plant.

3 Methodology

3.1 Set-up and considerations for the visual environment

The objective of this thesis is to program a simulator for the KAXAN ROV, capable to behave similarly to the real submarine.

Some important aspects of the program must be:

- The program has to be a real time simulation.
- The input interface must be similar to the input interface of the KAXAN
- A visual output of the robot must be provided to the user
- The user must have the ability to simulate different sea currents

The compiler selected to program this simulator was Visual Studio C# (C Sharp). C#'s language was selected instead of other languages because:

- It's intended to be simple, modern, general-purpose and object-oriented
- It has source code portability, especially for those programmers already familiar with C and C++
- C#'s applications are intended to be economical regarding processing power and memory requirements, but the language was not intended to compete directly on performance and size with C or assembly language

It is easier and faster to program in C# rather than C++, the language is more user-friendly than traditional C++, even though a program in C++ runs faster than a program in C#, a light program (in terms of size) like this, didn't present any speed during its operation. Another important aspect is that C# is object-oriented, therefore the code and the classes created for this program are suitable for its reuse in other projects.

For the creation of the virtual environment, the ROV's model was drawn using the libraries of OpenGL. OpenGL stands for Open Graphics Library and it is an API for rendering graphics, usually in 3D. Since graphic cards usually include an Open GL implementation, and the Open GL specification is not platform-specific, a lot of the graphic process is done directly by the graphic card.

Different plug-ins (or libraries) exist to bind OpenGL's functions and the forms created by C#, one of these libraries is OpenTK. OpenTK allows C#'s users to access OpenGL, OpenAL and OpenCL, and includes a lot of helpful functions like: mathematical functions, fonts, etc. It has been used in scientific visualizations, modelling software and other projects. It also works directly with the graphic card so a lot of computing work has been made by the graphic hardware, instead of the processor, making the application nicer and faster.

To work with OpenTK, it is necessary to include the library into the project and then add a control window to the form.

OpenTK offers support for different input devices, however the information about said support is scarce so the TaoFrame was used instead. TaoFrame is an older library than OpenTK, but it offers support for different joystick devices and the there's plenty information on how to do it.

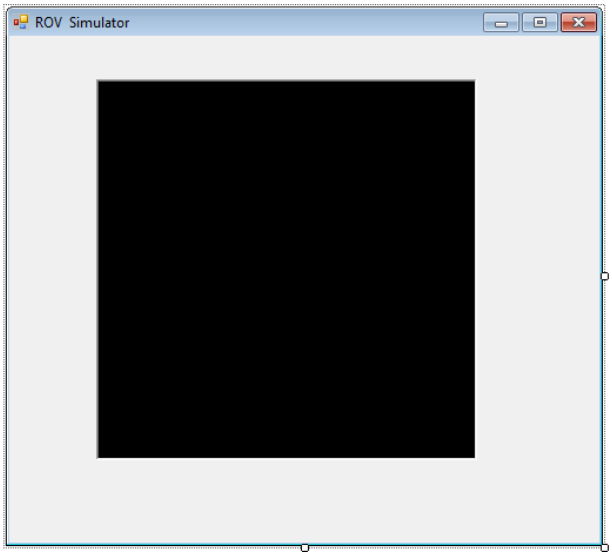


Figure 3.1: OpenTK control inside a C# Form

3.2 The graphic models

To draw a graphic model in OpenGL, one must specify the position of each vertex for each polygon, to add shadows and reflections to the drawings, one must include the normal vector for each vertex or polygon of the model. Modifying these models directly in OpenGL implies that every time that a vertex needs to be changed, it must be changed in all of the polygons that include this vertex. Doing this is easy for a simple model (few polygons) but when the model is more complex, has round edges or is very detailed, doing this manually, represents a real problem, so instead of drawing complex models directly with OpenGL, other specialized programs for CAD or 3D modelling can be used. These programs include several tools to draw and add effects to the model.

The disadvantage in drawing the models in another program is that the resulting file must be transformed to OpenGL's functions, however the model can be modified when needed with the specialized editor and reconverted for use in the simulator.

3.2.1 Creation of the models

The KAXAN is already built and all the designs were made using the CAD program: Solidworks, therefore for the purposes of this project Solidworks was also chosen.

The models drawn for this simulator were:

- KAXAN
  - Top section
  - Bottom section
- Oil Platform
  - Buildings
  - Platform

– Supports

As it can been seen, the models were split in parts because each part has its own specific color, and each part is loaded in the program as a complete object, with the "Object loader class" (it will be discussed in the following sections) it is not possible to specify a color (or any other property) for just a section of the object, it must be applied to the whole object.

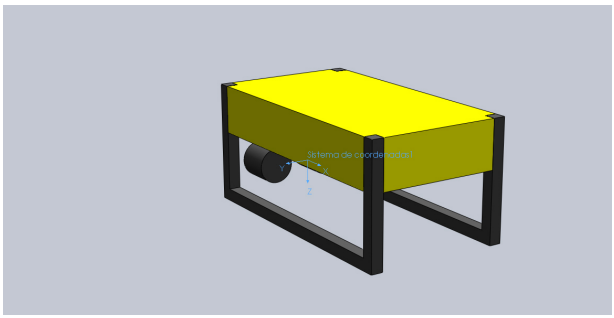


Figure 3.2: ROV on Solidworks

The original model of the ROV (Fig. 3.2) is very complex and has a lot of round shapes, therefore a simpler model was created, all of the parts were neglected with the exception of the structures and the thrusters in the back of the ROV. The thrusters were represented with two round cylinders in the back part of the ROV. When creating the model it is recommended to add extra axes of reference, but these axes must correspond with the axes of the *body-fixed* frame (Z axis pointing down). This is helpful in order to give the correct orientation to the model.

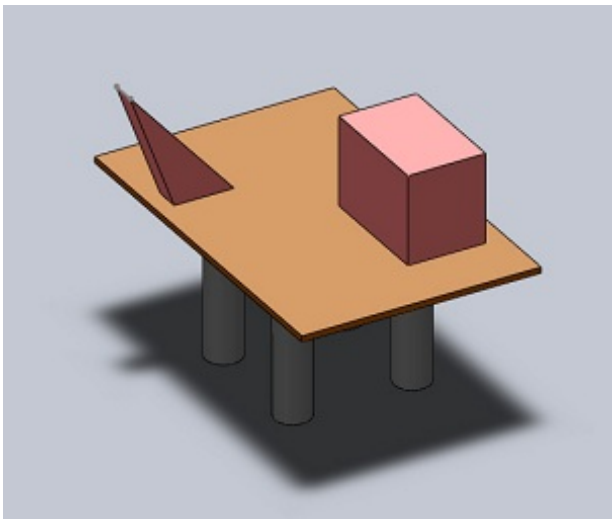


Figure 3.3: Oil Platform on Solidworks

The oil platform (Fig. 3.3) is also a very simple representation of the real object, and it works just to illustrate how a real platform would look from the perspective of the ROV. It is important to remark that the model is just a visual object, and it does not interfere neither the dynamics of the robot nor the currents.

A visual improvement to any program is the addition of shadows. The shadows create the effect of depth in a model.

To draw the shadows is necessary to:

- Include the information about the normal vector of each vertex
- Enable the lighting capabilities of OpenGL
- In order to use the colors of the model is necessary to define the behaviour of the materials

Calculating by hand the values of the normal vectors is easy but time consuming. The good thing about some CAD programs is that they do these calculations automatically. How to implement these vectors in the program will be defined later.

To enable the lighting capabilities it is required to define the light sources acting on the scene. There are three different components in light sources: Ambient, Diffuse and Specular, because of the complexity of these components, the differences between them are not going to be explained in this paper. OpenGL can set until 8 different light sources at the same time. The following code shows how to configure a light in the scene.

```
//Set the ambient, diffuse and specular components of the light
GL.Light(LightName.Light0, LightParameter.Ambient, Vector4.One);
GL.Light(LightName.Light0, LightParameter.Diffuse, Vector4.One);
GL.Light(LightName.Light0, LightParameter.Specular, Vector4.One);
GL.Light(LightName.Light0, LightParameter.Position,
 new Vector4(1, 1, -1, 0));
GL.Enable(EnableCap.Light0);
```

3.2.2 Lighting

The light configured is set as *Light0*, it is a white light and includes the specular, the diffuse and the ambient component. One important parameter is the position of the light, the position contains 4 values, the first 3 values are the X, Y and Z components, and the fourth has an special behaviour, it tells OpengGL whether the first three values are a position or a vector. In this example the fourth value is equal to 0, therefore the other coordinates are the components of the light vector.

This was the code used to create a light source, but enabling the lighting in the scene is still needed:

```
GL.Enable(EnableCap.Light0);
GL.Enable(EnableCap.Lighting);
```

EnableCap.Light0 turns on the Light0 and *EnableCap.Lighting* enables the lighting capability.

To calculate the light reflected on, OpenGL can take in consideration the color of the model. For example if the model is red, then all the light reflected will be red, because all the other colors are absorbed by the model. The materials have also ambient, specular and diffuse components, and they specify how the different light components will be reflected by the model. In addition the materials have a shining component, which defines how focused the specular light is, being 0 unfocused, the focus value goes form 1 to 128 being 128 a surface very notable and shiny. The code used to specify the materials properties is shown below:

```
Vector4 mat_amb = new Vector4(.2f, .2f, .2f, 1);
```

```
Vector4 mat_diff = new Vector4(.8f, .8f, .8f, 1);
Vector4 mat_spec = new Vector4(.8f, .8f, .8f, 1);
//Set the specular, diffuse, ambient
//and shininess components
//of the material's light response
GL.Material(MaterialFace.Front,
 MaterialParameter.Specular, mat_spec);
GL.Material(MaterialFace.Front,
 MaterialParameter.Diffuse, mat_diff);
GL.Material(MaterialFace.Front,
 MaterialParameter.Ambient, mat_amb);
GL.Material(MaterialFace.Front,
 MaterialParameter.Shininess, 25);
GL.ColorMaterial(MaterialFace.Front,
 ColorMaterialParameter.Ambient);
GL.ColorMaterial(MaterialFace.Front,
 ColorMaterialParameter.Diffuse);
//Enable material's response to the light
GL.Enable(EnableCap.ColorMaterial);
```

It is required to specify as well which material parameters track the current color, *GL.ColorMaterial()*is the function in charge to do this, if these properties are not specified then the color of the model is not taken in consideration by OpenGL.

3.2.3 Sea bottom & Textures

As said before simpler models can be drawn directly in OpenGL, this is the case for the models created for the bottom of the sea and the water surface.

To create the bottom of the sea a quad was draw, this was made by using simple equations of OpenGL, the following code will exemplify how to draw it:

```
//Creation of a polygon
GL.Begin(BeginMode.Polygon);
  GL.Vertex3(-0.2f, 0.2f, 0.0f);
  GL.Vertex3(0.2f, 0.2f, 0.0f);
  GL.Vertex3(0.2f, -0.2f, 0.0f);
  GL.Vertex3(-0.2f, -0.2f, 0.0f);
GL.End();
```

To add a texture different considerations must be made, the first one is the coordinate system, this is because the coordinate system of the textures is different that the original OpenGL coordinate system.

Therefore when the texture coordinates are bind with the polygon, in normal conditions, the polygon coordinate (0,0) must not interfere with the texture coordinate (0,0). The following example show how to apply bind a polygon vertex with a texture corner (or other position):

```
GL.BindTexture(TextureTarget.Texture2D, 0);
GL.Enable(EnableCap.Texture2D);
//Binding of texture coordinates with the polygons vertices
```

(a) OpenGL coodinates (b) Texture coodinates

Figure 3.4: OpenGL and Texture coordinates

```
GL.Begin(BeginMode.Polygon);
   GL.TexCoord2(0.0, 0.0);
   GL.Vertex3(-0.2f, 0.2f, 0.0f);
   GL.TexCoord2(1.0, 0.0);
   GL.Vertex3(0.2f, 0.2f, 0.0f);
   GL.TexCoord2(1.0, 1.0);
   GL.Vertex3(0.2f, -0.2f, 0.0f);
   GL.TexCoord2(0.0, 1.0);
   GL.Vertex3(-0.2f, -0.2f, 0.0f);
GL.End();
```

But before the texture can be bind with the polygon it is necessary to load the texture, it can be saved in a .bmp or .png or .jpg file and then it must be load into our program memory, assigned to a positon and then it can be used. The following code shows how to load the texture information in our program:

```
GL.BindTexture(TextureTarget.Texture2D, 0);
//Load a new image
Bitmap bmp = new Bitmap("sand.bmp");
//Extract image data
BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width,
 bmp.Height),ImageLockMode.ReadOnly,
 System.Drawing.Imaging.PixelFormat.Format32bppArgb);
//Bind texture to the image
GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba,
 bmp_data.Width, bmp_data.Height, 0,
 OpenTK.Graphics.OpenGL.PixelFormat.Bgra, PixelType.UnsignedByte,
 bmp_data.Scan0);
bmp.UnlockBits(bmp_data);
//Established texture parameters
GL.TexParameter(TextureTarget.Texture2D,
 TextureParameterName.TextureMinFilter,
 (int)TextureMinFilter.Linear);
GL.TexParameter(TextureTarget.Texture2D,
 TextureParameterName.TextureMagFilter,
 (int)TextureMagFilter.Linear);
//Generates mipmap for different sizes of polygons
GL.Ext.GenerateMipmap(GenerateMipmapTarget.Texture2D);
```

When more than one texture is being used (like in our case) it is important to use the sentence:

```
GL.BindTexture(TextureTarget.Texture2D, x);
```

where the x is a number that specifies in which position the texture is going to be saved (there are a limited number of positions that can be used, so one must be careful), the instruction is first used to bind the following texture to the current environment, and the it must be used when the texture is applied to the shapes.

To create the effect of a wide ocean bottom it is necessary reproduce this polygon n-times in X and Y direction, a nested for instruction and dX and dY can do the work:

```
//Creation of several polygons with the same texture coordinates
GL.Enable(EnableCap.Texture2D);
for (float i = -10; i < 10; i++)
 {
  for (float j = -10; j < 10; j++)
   {
    GL.Begin(BeginMode.Polygon);
    GL.TexCoord2(0.0, 0.0);
    GL.Vertex3(-0.2f + (0.4*i), 0.2f + (0.4*j), 0.0f);
    GL.TexCoord2(1.0, 0.0);
    GL.Vertex3(0.2f + (0.4*i), 0.2f + (0.4*j), 0.0f);
    GL.TexCoord2(1.0, 1.0);
    GL.Vertex3(0.2f + (0.4*i), -0.2f + (0.4*j), 0.0f);
    GL.TexCoord2(0.0, 1.0);
    GL.Vertex3(-0.2f + (0.4*i), -0.2f + (0.4*j), 0.0f);
    GL.End();
   }
 }
```

The final product will be an array of polygons producing an environment similar to an ocean bottom (Fig. 3.5)

3.2.4 Water surface & Blending

To create the sea surface it is important the same code can be used, with 2 small differences:

```
GL.BindTexture(TextureTarget.Texture2D, 1);
Bitmap bmp = new Bitmap("sea.bmp");
```

The name of the texture has changed to *sea.bmp* and since more than one texture is going to be used, this new texture must be bind to a different position than the bottom texture.

Other important difference is that a second polygon must be included in the model, but this polygon must face the opposite direction, by doing this the water surface can be watched from below.

Other visual difference between the bottom of the sea and the sea surface is that the surface is semi translucent, this effect can be added to our simulator with just a few more instructions:

```
//Enable blending capabilities
```

Figure 3.5: Bottom of the see

```
GL.Color4(1, 1, 1, 0.9f);
GL.Enable(EnableCap.Blend);
//Blending function
GL.BlendFunc(BlendingFactorSrc.SrcAlpha,
 BlendingFactorDest.OneMinusSrcAlpha);
GL.CallList(surface);
GL.Disable(EnableCap.Blend);
```

The blending capability of OpenGL allows the model to blend with the others, to accomplish this instead of using an RGB color, an RGBA color must be specified, the difference between an RGB and an RGBA color is the opacity channel. Depending on the opacity and the Blending function (GL.BlendFunc) different effects can be achieved.

Finally a water surface model was implemented (Fig. 3.6).

3.2.5 Fog

Other nice effect is the Fog effect. With this effect all the objects placed after a certain distance will be covered with fog. In OpenGL different properties of the fog can be programmed, like: the behaviour, the distance, the color, the density, etc.

```
GL.Enable(EnableCap.Fog);
float[] color = { 0f, 0.0f, .3f, 1f };
//Fog behaviour linear
GL.Fog(FogParameter.FogMode,(int)FogMode.Linear);
GL.Fog(FogParameter.FogColor, color);
//Density of the fog
GL.Fog(FogParameter.FogDensity, 0.35f);
```

Figure 3.6: Sea surface

```
GL.Hint(HintTarget.FogHint, HintMode.Nicest);
//Where the fog starts
GL.Fog(FogParameter.FogStart, 25.0f);
//Where it ends
GL.Fog(FogParameter.FogEnd, 50.0f);
```

The code above set the characteristics of the fog used in our model, and in the following image (Fig. 3.7), the final result can be seen.

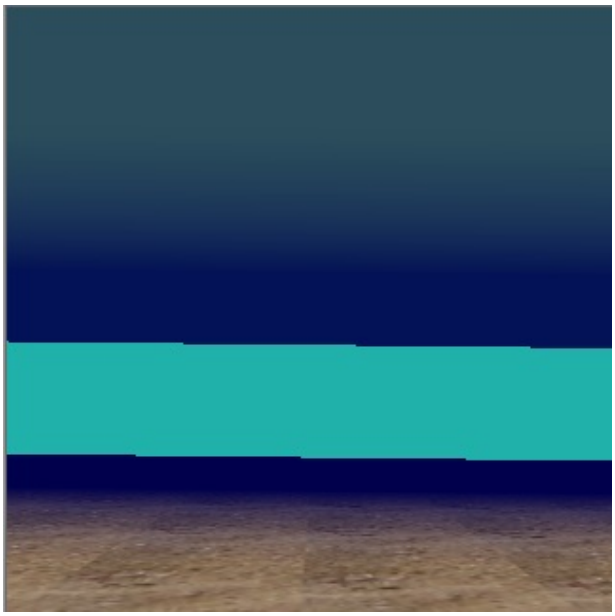


Figure 3.7: Fog effect

3.2.6 3D models in Wavefront format

Several formats for 3D model objects exist, almost each CAD editor or 3D model creator has its own extension, but portability is necessary and there are other common extensions to share 3D models. One commonly used extension is .obj (Wavefront .obj file). This extension can be used for several different programs, and the specifications of theses files are well known so it's easy to know how to implement them in a new program like this.

The disadvantage is that KAXAN model was made using Solidworks, and Solidworks doesn't work with .obj files, then another program had to be used to transform the Solidworks' files to Wavefront's files. For this purpose the 3D model software: Blender, was selected. Solidworks can save the model like an STL file (.stl) and then Blender can import this file, change it if necessary (in case of round shapes for example) and export the model as Wavefront.

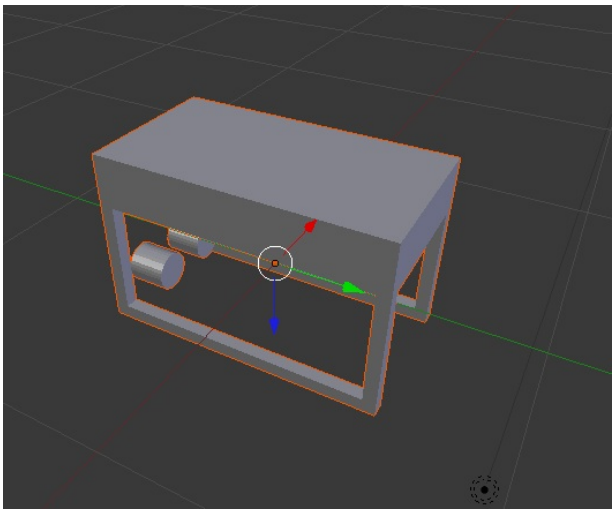


Figure 3.8: ROV on Blender

Round shapes have the problem that are drawn using several polygons, using Blender we can reduce the number of polygons in the model by joining vertex that are close to each other. If the number of polygons is reduced, then the quality of the image decreases, but also the processing work of this graphics diminishes, we must not forget that not only the KAXAN model must be drawn, also all the virtual environment surrounding it.

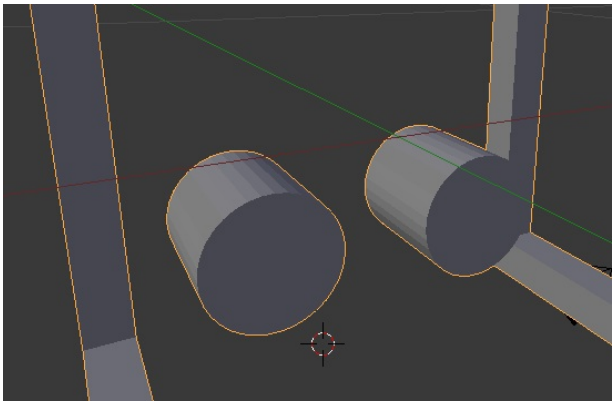


Figure 3.9: Round figures using 340 polygons

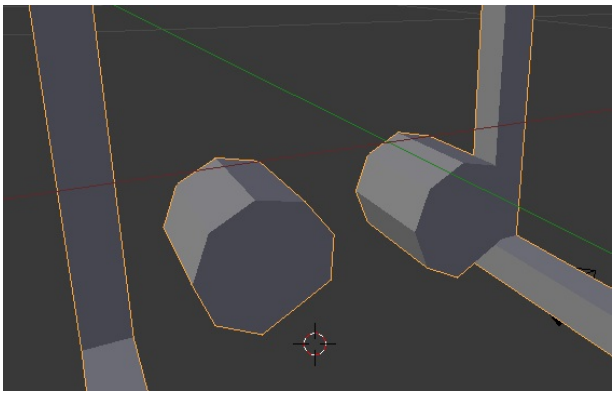


Figure 3.10: Round figures using 124 polygons, after reduction with Blender

A Wavefront file is made of the following sections:

Vertex section: Each vertex specified in the file is precede by a *'v '*, it is important not to forget the space after the 'v', and then the x, y and z coordinates. After each vertex definition the next 'v ' will be placed (if there is any other vertex left). For example *v 0.0 0.0 1.0v 1.0 0.0 0.0*, defines 2 vertex, one in the coordinates (0,0,1) and the second in (1,0,0).

Texture coordinates section: This section begins with a *'vt '*, followed by the u and v coordinates, these u and v coordinates are points of the texture that will be bound to some vertex in the polygon. For example *vt 0.0 1.0vt 1.0 0.0*, defines 2 texture coordinates, the first coordinate is the (0,1) of the bitmap and the second the (1,0).

Normals section: Each normal specified in the file is preceded by a *'vn '*, and then the x, y and z component of the vector, as in the Vertex section after each vertex definition a new *'vn '* will precede. For example *vn 0.7 0.7 0.0vn 0.0 1.0 0.0*, defines 2 normals, the first pointing to (0.7,0.7,0) and the second to (0,1,0).

Faces section: The faces definitions begin with an *'f '* and then the three or more indexes. Each index contains the information about the vertex, the texture coordinates and the normal vector, and each index can be written in 3 different ways.

1. The first one is just the information about the vertex, for example *f 1 2 3f 2 3 4 5*, defines 2 faces, the first one is made by 3 different vertex, in this case *'1'* that means the first defined vertex in the vertex section, *'2'* the second defined vertex and so on, so the first face is a triangle made by the first, the second and the third defined vertex. The second face is made with the second, the third, the fourth and the fifth defined vertex, so it will be a four sided polygon like an rectangle.
2. The second specifies the vertex and the texture coordinate and these two are separated by a slash *'/'*. For example *f 1/1 2/2 3/3f 2/1 3/2 4/3 5/4*, the first face has the same shape as in the last example, but in this example the first vertex is bound with the first texture coordinate and so on.
3. The third specifies the vertex, the texture coordinate and the normal vector, these three are separated by a slash '/', here is important to say that the texture coordinate can be omitted. For example *f 1//1 2//2 3//3f 2//1 3//2 4//3 5//4* the first face has the same shape as in the last example, but all the texture coordinates were omitted and instead a new normal vector was bound to each vertex. This is actually the specification used in the simulator.

There are more specifications in the Wavefront files that can be used to select textures or materials, but they were not considered in this project due to timing and complexity.

3.2.7 Object loader Class

As said before the models created in a Wavefront format need to be transformed to OpenGL functions, that is why for the purposes of this project a Class was created, so the model can be easily changed in a CAD application and then easily loaded in the simulator without making changes to the code of the program. This class can be found in the files of the program as *load_obj.cs*.

This Class is divided in 3 sections:

- Object Structure
- Decoding
- Printing

Object Structure is a section where a new structure is defined, every instance of the structure will hold all the information about the model like: its vertex, its normals and the faces (or polygons) forming the model.

```
public struct new_object
{
        public ArrayList vertex;
        public ArrayList normals;
        public ArrayList faces
}
```

The structure holds 3 *ArrayList*, these lists are an easy way to store and retrieve information in C#. To create an instance of the object it is necessary to add the class to the project and then the following code to our main file:

```
load_obj.new_object rov;
```

Now a new instance of the structure call *rov* has been created.

Decoding is the section where the .obj file is being read and then save into the structure defined before.

The first part of the function is the declaration part:

```
public new_object decoder(string file_name)
{
        new_object obj = new new_object();
        obj.vertex = new ArrayList();
        obj.normals = new ArrayList();
        obj.faces = new ArrayList();
        string file_str = "";
```

It is worth mentioning that the function returns a structure of type new_object and its only input parameter is a string containing the path and name of the .obj file. Inside of the structure a new empty

instance of our structure is declared. Since the structure is formed by an Arraylist (that is also a class), the instances for the vertex, normals, and faces must be initialized.

After the initialization it is necessary to check if the file exists, if not a dummy file call *cube.jpg* will be loaded

```
if (File.Exists(file_name) == true)
{
        StreamReader file = new StreamReader(file_name);
        file_str = file.ReadToEnd();
        file.Close();
}
else
{
        StreamReader file = new StreamReader("cube.obj");
        file_str = file.ReadToEnd();
        file.Close();
}
```

As it was explained in the Wavefront section (3.2.6) the .obj file used in this application contains three parts: the vertex part, the normals and the faces. The vertex part and the normals part work in a very similar way, therefore only the vertex part will be explained.

Every vertex triad is preceded by a *'v '* (a *'vn '* in the case of the normals) so the function must look for every *'v '* inside the document. After finding the chain (*'v '*), the following 3 numbers will be the coordinates of the vertex, the function will take this substring and then divided by their comas, the first number will be the the x coordinate, then the y and finally the z coordinate. These coordinates are saved into a *Vector3d* structure and finally this structure is added to the *obj.vertex* (*obj.normals* in the case of the normals) array.

```
int i = 0;
int index_nextv = 0;
do
{
        if (index_nextv == 0)
        {
                //Search where the next vector begins
                int index = file_str.IndexOf("v ", i);
                i = index + 2;
        }
        else
        {
                i = index_nextv;
                i += 2;
        }
        string[] num = new string[3];
        string sub = "";
        index_nextv = file_str.IndexOf("v ", i);
        if (index_nextv > 0)
        {
                sub = file_str.Substring(i, index_nextv - i);
        }
```

```
        else
        {
                int nextvn = file_str.IndexOf("vn ", i);
                sub = file_str.Substring(i, nextvn - i);
        }
        num = sub.Split(' ');
        double x, y, z;
        x = double.Parse(num[0]);
        y = double.Parse(num[1]);
        z = double.Parse(num[2]);
        Vector3d v = new Vector3d(x, y, z);
        obj.vertex.Add(v);
} while (index_nextv > 0);
```

The difference between the faces part and the vertex part is that in the faces case, they are constructed for at least 3 vertex but they can be formed by 4 or more, and for every corner of the polygon it exists also a normal vector bound to it. After finding the *'f '* preceding the corners information, this subchain is divided one more time at every blank space, and each section of this is also split at every slash, the first element of the slash is the vertex index, the second is the texture coordinate which will be empty in our models, and the third one will be the normal index. The normal index and the vertex index are saved as text into a string variable separated by a coma, then every other pair of normal and vertex indexes are added to the string. Finally this string is added to the *obj.faces* array.

```
do
{
        if (index_nextf == 0)
        {
                int index = file_str.IndexOf("f ", i);
                i = index + 2;
        }
        else
        {
                i = index_nextf;
                i += 2;
        }
        string num = "";
        string sub = "";
        index_nextf = file_str.IndexOf("f ", i);
        if (index_nextf > 0)
        {
                sub = file_str.Substring(i, index_nextf - i);
        }
        else
        {
                sub = file_str.Substring(i);
        }
        string[] face_parts;
        face_parts = sub.Split(' ');
        for (int k = 0; k < face_parts.Length; k++)
        {
                string[] vertex;
                vertex = face_parts[k].Split('/');
```

```
                num += vertex[2] + "," + vertex[0] + ",";
        }
        num = num.Substring(0, num.Length - 1);
        obj.faces.Add(num);
} while (index_nextf > 0);
```

This ends the decoder section, now all the information about the model is stored in the target structure.

Printing is the section where the model is actually transformed into OpenGL instructions. The function in charge of this is called *print*, and its declaration is:

```
public string print(new_object obj, BeginMode mode, bool text,
        double scale)
{
        string str = "";
```

The *print* function returns a string, and its input parameters are:

- The model contained in a *new_object* structure
- The OpenGL printing mode (BeginMode.Polygon draws a polygon, BeginMode.Lines draws only the edges of the faces, BeginMode.Points draws only dots)
- A bool value, if *false* the function will return an empty string but will execute the OpenGL instructions to draw the model, if *true* the function will return a string containing the functions but it will not execute any OpenGL instructions.
- And finally, if necessary, a double value used to scale the model (but not the normal vectors), being 1 the default size of the model, 2 the double, and 0.5 the half.

The first section of the function is to declare the drawing mode that OpenGL must use:

```
if (text)
        str += "\r\nGL.Begin(BeginMode." + mode.ToString() + ");";
else
        GL.Begin(mode);
```

Now it's time to draw the polygons. First, remember that the faces were saved in a string format in the next sequence: *f=vn1,v1,vn2,v2,vn3,v3...,vnk,vk*,where *vn1* is the index of the normal vector and *v1* is the index of the position of the first corner.

The program reads every enter of the *obj.faces* array, after reading it, the function splits this string in every coma, and runs a loop with an increment of two to read the pairs of objects in the array. Then selects the normal from the *obj.normals* and the position from the *obj.vertex*.

```
for (int j = 0; j < segment.Length - 1; j += 2)
{
        int num_normal = int.Parse(segment[j]);
        int num_vertex = int.Parse(segment[j + 1]);
        num_normal--;
```

```
        num_vertex--;
        if (text)
                str += "\r\nv=new Vector3d " + obj.normals[num_normal].ToString()
                        +";\r\nGL.Normal3(v);";
        else
                GL.Normal3((Vector3d)obj.normals[num_normal]);
        v = (Vector3d)obj.vertex[num_vertex];
        v *= scale;
        if (text)
                str += "\r\nv=new Vector3d " + v + ";\r\nGL.Vertex3(v);";
        else
                GL.Vertex3(v);
}
```

The last part is the closing of the drawing functions of OpenGL.

```
if (text)
        str += "\r\nGL.End();\r\n";
else
        GL.End();
}
return str;
```

To create an instance of the decoder and printer function with the name *loader*, the following code must be added:

```
load_obj loader = new load_obj();
```

Here is an implementation of this class used to load the models of the ROV inside the program:

```
//Creates an OpenGL List
//This is useful when certain polygons
//require a lot of operations to calculate the vertices
//with the list these calculations
//are only made once
GL.NewList(rov_model, ListMode.Compile);
rov = loader.decoder("models/rov_p1.obj");
GL.Color3(Color.Yellow);
loader.print(rov, BeginMode.Polygon, false, 1);
rov = loader.decoder("models/rov_p2.obj");
GL.Color3(Color.Black);
loader.print(rov, BeginMode.Polygon, false, 1);
GL.EndList();
```

3.3 Joystick functions

To control the KAXAN ROV two different input devices exist: the first one is a SONY Dualshock 3 Sintaxis joystick (Fig. 3.11), and the second one is a joystick designed in CIDESI specially for the ROV. In this project the Sintaxis joystick was selected because it is more portable than the custom-made joystick

and it offers more plugging options (USB cable or wireless via Bluetooth). The driver used in this project was downloaded from the MotioninJoy website[10], with this driver it is possible to connect the Sintaxis joystick with a windows operating system using USB or Bluetooth. Using this driver it is possible to configure the joystick in 3 different options: Playstation control, Xbox 360 control, or custom. In this implementation the joystick was configured as an Xbox 360 control, in the case of not having a Sintaxis joystick it can be replaced with a Xbox 360 control. With this configuration the joystick has: 16 Buttons, 5 Axes and a Hat switch (POV).

Figure 3.11: Dualshock 3 Sintaxis Joystick

As said before in the considerations section (3.1) the TaoFramework library is used to retrieve information about the joystick, there are two main ways to achieve this purpose. The first one consists in having a *thread* in our program which is constantly reading the flags of the joystick. These flags are activated when a button is pressed or an axis is moved. The second way to retrieve the information of the joystick is to read the input information at request, not in a continue way as in the first option. The second option was selected because it reduces complexity in the program and because it was not necessary to read continuously the state of the joystick, it can be read every time the scene is redrawn (every 0.16 milliseconds).

First the program initiates the joystick invocations, and then searches all the joysticks connected to the computer, if at least one is connected the program selects the first one.

```
Sdl.SDL_Init(Sdl.SDL_INIT_JOYSTICK);
int num_joysticks = Sdl.SDL_NumJoysticks();
IntPtr Joystick;
if (num_joysticks > 0)
{
        Joystick = Sdl.SDL_JoystickOpen(0);
}
```

A joystick structure was created to save the characteristics of the joystick, for example, the returned value by the joystick at its maximum axis position and also at its minimum axis position (Appendix C.1).

To read the current information of the joystick an *update* is invoked, then each value of interest is taken and processed. The x axis controls the thruster that moves the ROV sideways and also is the one that makes the KAXAN turn along its Z axis. The y axis of the Joystick controls the voltage in the thrusters that moves the ROV back and forth, these thrusters are located in the back and inferior sides of the ROV. The z axis changes the desired Z. The depth control is the one in charge of supplying the necessary voltage to move the ROV to this position. This desired Z cannot be less than 0 so software constraint

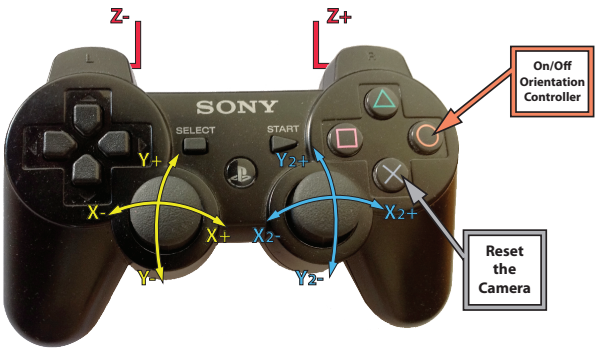


Figure 3.12: Axes labels

was included. The *x2* axis rotates the camera along the center of the object, this rotation has a radius of 3m from the center of the model. and finally the *y2* axis moves the camera along the Z axis of the environment. The *cross* button of the joystick is used to reset the camera to its initial position from the center of the object and the *circle* button is used to enable and disable the direction control. It worth mentioning that the camera on the Orthogonal view is controlled by *x2* and *y2* axis, but not the camera on the Perspective view. The code can be seen in appendix C.2.

At the end of the program is important to *close* the use of the joystick, it is just necessary to add this line:

```
Sdl.SDL_JoystickClose(Joystick);
```

3.4 Solution of equations (Mass-Spring-Damper)

The dynamic behaviour of the ROV can be described with the equation:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) = \tau$$

If we want to know the η vector is necessary to clear the $\dot{\nu}$ vector from 2.10 and solve the differential equation, make the corresponding transformations to earth fixed coordinates and we will obtain η vector.

$$\dot{\nu} = M^{-1}(\tau - (C(\nu)\nu + D(\nu)\nu + g(\eta))) \tag{3.1}$$

$$\dot{\eta} = J(\eta_2)\,\nu$$

$$\eta = \int \dot{\eta}\, dt$$

The main problem is, that the solution of the differential equation 3.1 is really complex (let's remember than *M, C and D* are matrices of 6×6), therefore solving this equation on real time for every change in the ROV's environment (like sea currents) is not a practical option. Therefore other methods to solve this equation must be used.

Instead of searching for the solution to this problem, we can try looking for the solution of a simpler but similar problem, for example a Mass Spring Damper system.

The Mass Spring Damper system behaves as the following equation:

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = f(t) \tag{3.2}$$

Now let's solve the equation:

$$\ddot{x}(t) = \frac{f(t) - c\dot{x}(t) - kx(t)}{m} \tag{3.3}$$

Let's assume that the object starts moving from rest, therefore *x(0)=0, $\dot{x}$(0)=0* and let's give values to $m = 1Kg$, $c = 10\frac{Ns}{m}$, $k = 20\frac{N}{m}$ and $f = 50N$, then

$$\begin{aligned}
\ddot{x}(t_0) &= \frac{f(0) - c\dot{x}(0) - kx(0)}{m} \\
\ddot{x}(t_0) &= \frac{50 - 10 * 0 - 20 * 0}{1} \\
\ddot{x}(t_0) &= 50
\end{aligned}$$

Now we want to know the position at 0.01s:

$$\begin{aligned}
t_0 &= 0 \\
t_1 &= 0.01 \\
\ddot{x}(t_1) &\approx \frac{f(t) - c\dot{x}(t_0) - kx(t_0)}{m} \\
&\approx \frac{50 - 10 * 0 - 20 * 0}{1} \\
&\approx 50 \\
\dot{x}(t_1) &= \int_{t_0}^{t_1} \ddot{x}(t)dt \approx 0.5 \\
x(t_1) &= \int_{t_0}^{t_1} \dot{x}(t)dt \approx 0.0025
\end{aligned}$$

The obtained $\ddot{x}(t_1)$ value is just an approximation, because to get the real value (or at least a very similar one) is necessary to solve the equation 3.3 with the values of $\dot{x}(t_1)$ and $x(t_1)$, but to get $\dot{x}(t_1)$ and $x(t_1)$ it's necessary the value of $\ddot{x}(t_1)$, therefore the proposal is to make a recursive algorithm that recalculates the value of $\ddot{x}(t_1)$, $\dot{x}(t_1)$ and $x(t_1)$ until they converge to a similar value or a certain number of loops has been reached.

For example if $\ddot{x}(t_1)$ is recalculated with the values obtained by the first run, a new set of numbers will be returned:

$$\begin{aligned}
\ddot{x}(t_1) &\approx \frac{f(t) - c\dot{x}(t_1) - kx(t_1)}{m} \\
&\approx \frac{50 - 10 * .5 - 20 * .0025}{1} \\
&\approx 44.95 \\
\dot{x}(t_1) &= \int_{t_0}^{t_1} \ddot{x}(t)dt \approx 0.4748 \\
x(t_1) &= \int_{t_0}^{t_1} \dot{x}(t)dt \approx 0.0024
\end{aligned}$$

If this process is repeated a few more times a set of converged values are obtained:

$$\ddot{x}(t_1) = 45.1928$$
$$\dot{x}(t_1) = 0.4746$$
$$x(t_1) = 0.0024$$

The intention is to create a simulator on C# to calculate the ROV's movement, then a program in C# to calculate the Mass Spring Damper behaviour must be also created (Fig. 3.14). To compare the results a *Matlab Simulink* model was also created (Fig. 3.13).

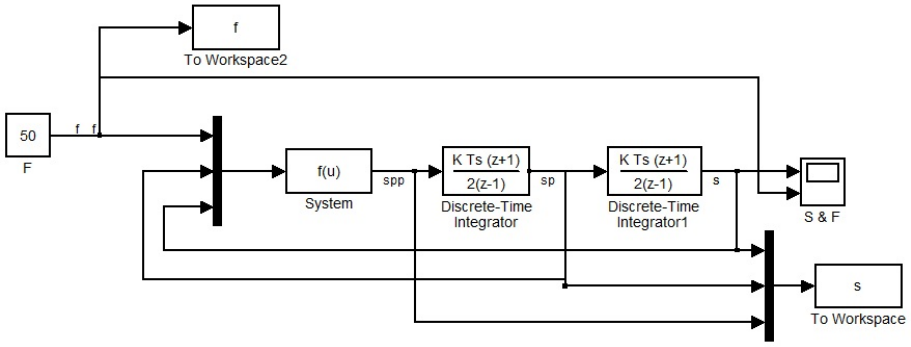


Figure 3.13: Mass Spring Damper model in Matlab

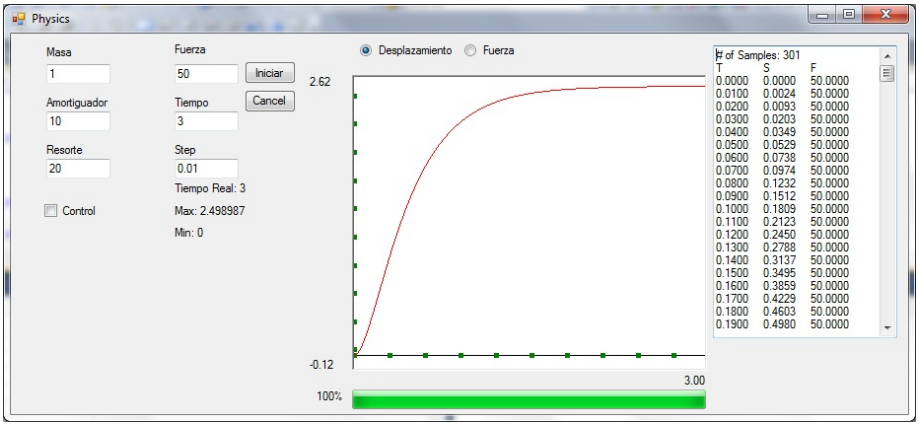


Figure 3.14: Mass Spring Damper model in C#

In Matlab is not required to include the recursive algorithm since Matlab does it automatically. Another characteristic of Matlab is that it can simulate as continue mode or as discrete mode, the discrete mode has been selected so the C# application can be modified to match the Matlab's model.

To solve the equations it is necessary to integrate and to differentiate (differentiation is necessary for the PID controller) and since we are working with a discrete process, numerical methods are needed to do both operations.

There are several types of numerical methods to integrate, here two of the more common methods are going to be explained. The first one is the *trapezoidal* method where:

$$\int_a^b f(x)dx \approx (b-a) * \frac{f(a)+f(b)}{2}$$

The advantages of this method is that only two points are needed to know the value of the integral and if the step of time is very short then a very good approximation is obtained. For big periods of time, a property of the define integral can be used:

$$\int_0^{a_2} f(x)dx = \int_0^{a_1} f(x)dx + \int_{a_1}^{a_2} f(x)dx \therefore$$

$$\text{if } A_1 = \int_0^{a_1} f(x)dx$$

$$A_2 = \int_0^{a_2} f(x)dx = A_1 + \int_{a_1}^{a_2} f(x)dx$$

$$\vdots$$

$$A_n = \int_0^{a_n} f(x)dx = A_{n-1} + \int_{a_{n-1}}^{a_n} f(x)dx$$

The second method is the *Simpson's rule*, where the function $f(x)$ is approximated by a quadratic polynomial in the interval $[a, b]$.

$$\int_a^b f(x)dx \approx \int_a^b P(x)dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

The problem with the Simpson's rule is that the function's value at the middle point $f(\frac{a+b}{2})$ is needed.

The first method was selected because it contains fewer operations and it is easier to calculate, the results (shown later in this section) proved that this method is accurate enough for the problem's needs.

The code of the numerical method is shown below:

```
private double integration(ref ArrayList func,ref ArrayList func1, double time)
{
        double integrate = 0;
        double f0, f1;
        int count = func.Count;
        if (count != 1)
        {
                f0 = (double)func[count - 2];
                f1 = (double)func[count - 1];
                //Trapezoidal Method (f1+f0)*dt/2
                integrate = (time) * (f1 + f0) / 2;
                //Add the value of the integral to the old values,
                //this is equal to the integration from 0 to the current value
                integrate += (double)func1[count - 2];
        }
        else
                //If there is only one value on the array,
                //the result of the integration is f(0)*dt/2
                return (double)func[0]*time/2;

        return integrate;
}
```

To continue with the numerical methods let's take a look to the differentiation method. Let's say that

$$f(x) = p_n(x) + R_n(x)$$

where p is a Newton's polynomial of degree n, then the first approximation of the differentiation is:

$$\frac{df(x)}{dx} \approx \frac{dp_n(x)}{dx}$$

or in general

$$\frac{d^n f(x)}{d^n x} \approx \frac{d^n p_n(x)}{d^n x}$$

where the error is defined as:

$$\frac{d^n R_n(x)}{d^n x}$$

With a polynomial of degree 1 the result is:

$$f(x) \approx p_1(x) = f[x_0] + (x - x_0)f[x_0, x_1] = f(x_0) + (x - x_0)\frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$\frac{df(x)}{dx} \approx \frac{dp_1(x)}{dx} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{3.4}$$

This equation 3.4 is the most used numerical differentiation method.

Now let's use a second degree polynomial:

$$\begin{aligned}
f(x) \approx p_2(x) &= f[x_0] + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] \\
\frac{df(x)}{dx} \approx \frac{dp_2(x)}{dx} &= f[x_0, x_1] + (2x - x_0 - x_1)f[x_0, x_1, x_2] \\
&= \frac{f(x_1) - f(x_0)}{x_1 - x_0} + (2x - x_0 - x_1)\frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{(x_2 - x_0)} \\
&= \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\
&\quad + (2x - x_0 - x_1)\frac{(x_1 - x_0)(f(x_2) - f(x_1)) - (x_2 - x_1)(f(x_1) - f(x_0))}{(x_2 - x_0)(x_2 - x_1)(x_1 - x_0)} \\
&= \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\
&\quad + (2x - x_0 - x_1)\frac{(x_1 - x_0)f(x_2) - (x_2 - x_0)f(x_1) + (x_2 - x_1)f(x_0)}{(x_2 - x_0)(x_2 - x_1)(x_1 - x_0)} \\
&= \frac{((2x - x_0 - x_1) - (x_2 - x_0)f(x_0)}{(x_2 - x_0)(x_1 - x_0)} \\
&\quad + \frac{(-(2x - x_0 - x_1) + (x_2 - x_1)f(x_1)}{(x_2 - x_1)(x_1 - x_0)} \\
&\quad + \frac{(2x - x_0 - x_1)f(x_2)}{(x_2 - x_0)(x_2 - x_1)} \\
\therefore \frac{df(x)}{dx} &\approx \frac{(2x - x_1 - x_2)f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{(2x - x_0 - x_2)f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{(2x - x_0 - x_1)f(x_2)}{(x_2 - x_0)(x_2 - x_1)}
\end{aligned} \tag{3.5}$$

If $x = x_2$ is substituted in 3.5 the function changes to:

$$f'(x_2) \approx \frac{(x_2 - x_1)f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{(x_2 - x_0)f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{(2x_2 - x_0 - x_1)f(x_2)}{(x_2 - x_0)(x_2 - x_1)} \tag{3.6}$$

This is the representation of equation 3.6 in C# code:

```
private double diff(ref ArrayList func, ref ArrayList time, double d_time)
{
        double diff = 0;
        int count = func.Count;
        double t0,t1,t2,f0,f1,f2;
        if (count == 1)
        {
                //This function is the same as the function in Matlab
                /*
                f0 = (double)func[0];
                diff = f0 / d_time;
                return diff;
                /* */
                //But when there is only one element we cannot be sure of the Derivate to
                return 0;
        }
        //If only two elements are present, the polynomial of 1 degree is used
        else if (count <= 2)
        {
                f0= (double)func[0];
                f1= (double)func[1];
                diff = (f1 - f0) / (d_time);
        }
        //If more than two elements are present, the polynomial of 2 degrees is used
        else
        {
                t0 = (double)time[count - 3];
                t1 = (double)time[count - 2];
                t2 = (double)time[count - 1];
                f0 = (double)func[count - 3];
                f1 = (double)func[count - 2];
                f2 = (double)func[count - 1];
                double p1 =((t2 - t1) * (f0) / ((t0 - t1) * (t0 - t2)));
                double p2=((t2 - t0) * (f1) / ((t1 - t0) * (t1 - t2)) );
                double p3=(((2 * t2) - t1 - t0) * (f2) / ((t2 - t1) * (t2 - t0)));
                diff = p1 + p2 + p3;
        }
return diff;
}
```

3.4.1 Matlab vs C# vs Real

It's important to remark that the C# solution's model works in discrete mode, with a given $\triangle t$ equals to a step given by the user (default value of 0.01 seconds), therefore to make a good comparison with the Matlab's model, this must be configured to work in discrete mode with the same step time and and using

the same integration method. Unfortunately Matlab uses different methods to differentiate than the one used in the C#'s model.

Because 3.2 is not a big and complex model, the solution of the differential equation 3.3 (using the given values of the constants) can be found:

$$\ddot{x}(t) = \frac{50 - 10\dot{x}(t) - 20x(t)}{1}$$

$$x(t) = \frac{5}{2} - \frac{5^{1/2}(5^{1/2} - 5)}{4e^{t(5^{1/2}+5)}} - \frac{5^{1/2}e^{t(5^{1/2}-5)}(5^{1/2} + 5)}{4} \tag{3.7}$$

$$\dot{x}(t) = \frac{5^{1/2}(5^{1/2} - 5)(5^{1/2} + 5)}{4e^{t(5^{1/2}+5)}} - \frac{5^{1/2}e^{t(5^{1/2}-5)}(5^{1/2} - 5)(5^{1/2} + 5)}{4} \tag{3.8}$$

$$\ddot{x}(t) = -\frac{5^{1/2}(5^{1/2} - 5)(5^{1/2} + 5)^2}{4e^{t(5^{1/2}+5)}} - \frac{5^{1/2}e^{t(5^{1/2}-5)}(5^{1/2} - 5)^2(5^{1/2} + 5)}{4} \tag{3.9}$$

Now with equations 3.7, 3.8 and 3.9 the results from Matlab and C# can be compared to the equation model in continues mode instead of discrete behaviour.

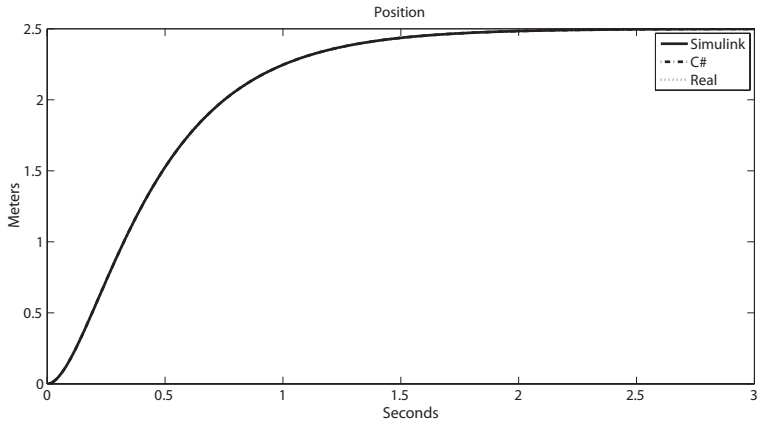


Figure 3.15: Position

Time	Simulink	C#	Real
0	0	0	0
0.01	0.00238	0.00238	0.002418
0.02	0.009288	0.009288	0.009359
0.03	0.020281	0.020281	0.020379
0.04	0.034948	0.034948	0.035069
0.05	0.052912	0.052912	0.053051
0.06	0.073825	0.073825	0.073979
0.07	0.097367	0.097367	0.097532
0.08	0.12324	0.12324	0.12341
0.09	0.15118	0.15118	0.15136
0.1	0.18093	0.18093	0.18111

Table 3.1: Position

As it can be seen in figures: 3.15, 3.16 and 3.17, the 3 graphics are very similar between each other, and comparing the results between 0 and 0.1 seconds on tables: 3.1, 3.2 and 3.3 the values of Simulink and

C# are the same, there is only a little difference no bigger than 1% between Simulink/C# and the Real value.

As it was defined in the problem's objectives a PID controller must be included in the simulator, therefore

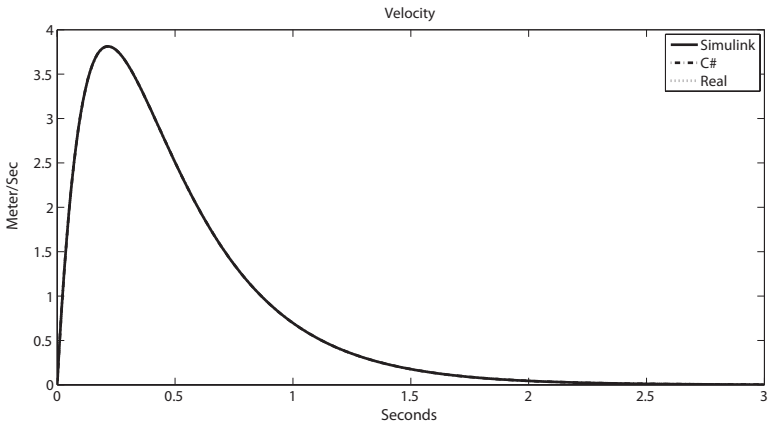


Figure 3.16: Velocity

Time	Simulink	C#	Real
0	0	0	0
0.01	0.47596	0.47596	0.47565
0.02	0.90571	0.90571	0.90514
0.03	1.2928	1.2928	1.292
0.04	1.6406	1.6406	1.6396
0.05	1.9522	1.9522	1.9511
0.06	2.2304	2.2304	2.2291
0.07	2.4779	2.4779	2.4765
0.08	2.6971	2.6971	2.6956
0.09	2.8903	2.8903	2.8887
0.1	3.0596	3.0596	3.058

Table 3.2: Velocity

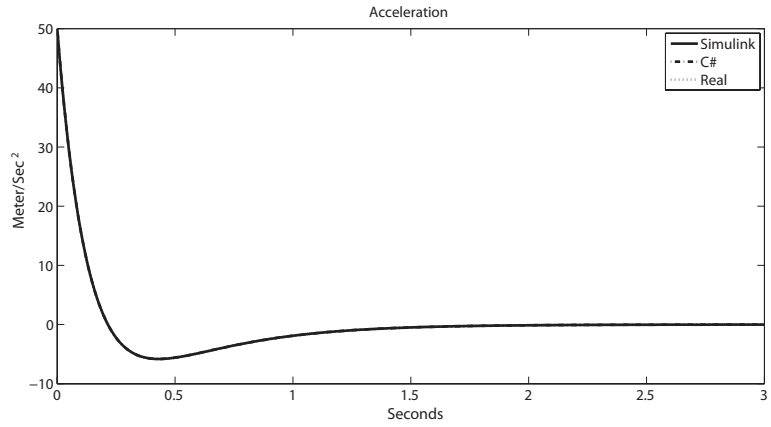


Figure 3.17: Acceleration

Time	Simulink	C#	Real
0	50	50	50
0.01	45.193	45.193	45.195
0.02	40.757	40.757	40.761
0.03	36.666	36.666	36.672
0.04	32.895	32.895	32.902
0.05	29.42	29.42	29.428
0.06	26.219	26.219	26.229
0.07	23.274	23.274	23.284
0.08	20.565	20.565	20.576
0.09	18.074	18.074	18.086
0.1	15.786	15.786	15.798

Table 3.3: Acceleration

a PID controller was implemented in this test. The first step was to program a proportional (P) controller.

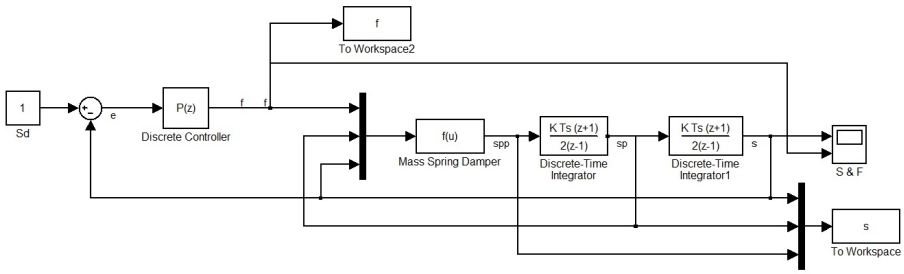


Figure 3.18: Matlab's control model

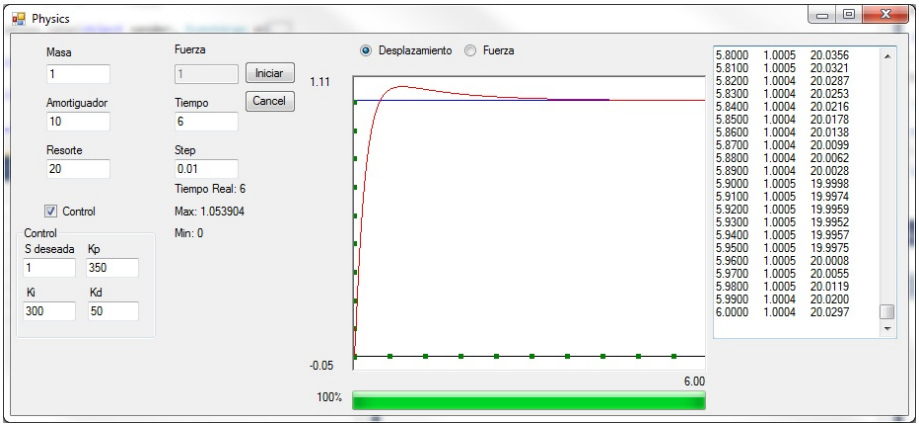


Figure 3.19: C#'s control model

In figures 3.18 and 3.19 both control models can be seen. This figures represent the P, PI and PID controller, the only difference between each controller is that for P controller both K_i and K_d equals 0, and in PI controller K_d equals 0.

Like in section 3.4.1 the differential equations that includes the P, PI and PID controller can be calculated:

$$F(t) = K_p * \varepsilon(t) + K_i \int \varepsilon(t)dt + K_d \frac{d\varepsilon(t)}{dt} \tag{3.10}$$

$$\text{The error is defined as: } \varepsilon(t) = x_d - x(t) \tag{3.11}$$

Replacing 3.10 and 3.11 in 3.3 results in:

$$\ddot{x}(t) = \frac{\left(K_p(x_d - x(t)) + K_i \int (x_d - x(t))dt + K_d \frac{d(x_d - x(t))}{dt}\right) - c\dot{x}(t) - kx(t)}{m} \tag{3.12}$$

The solution of 3.12 results in a very long equation that is not worth writing in this paper.

In the next sections the results of each type or controller (P, PI and PID) are going to be explained, but before continuing the initial conditions must be established:

$$x_d = 1$$
$$Kp = 350$$
$$Ki = 300$$
$$Kd = 50$$

3.4.2 Results of P Controller

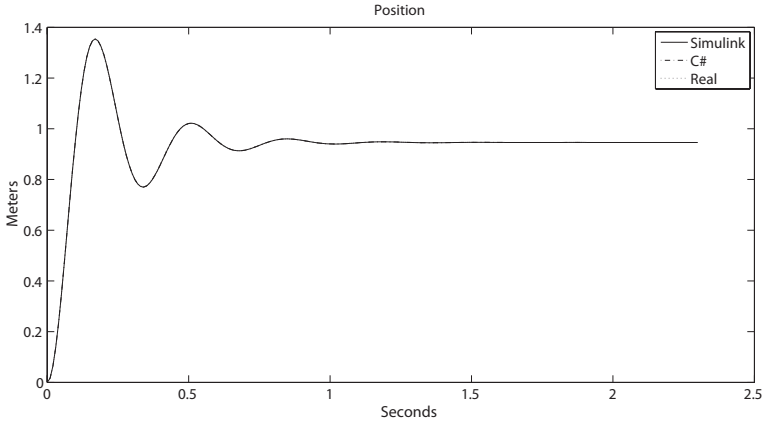


Figure 3.20: Position P Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	0.0165	0.0165	0.0169	0	0.0004	0.0004
0.02	0.0639	0.0639	0.0648	0	0.0009	0.0009
0.03	0.1377	0.1377	0.139	0	0.0013	0.0013
0.04	0.2327	0.2327	0.2345	0	0.0018	0.0018
0.05	0.3437	0.3437	0.3459	0	0.0022	0.0022
0.06	0.4652	0.4652	0.4678	0	0.0026	0.0026
0.07	0.5921	0.5921	0.5948	0	0.0027	0.0027
0.08	0.7193	0.7193	0.7221	0	0.0028	0.0028
0.09	0.8424	0.8424	0.8451	0	0.0027	0.0027
0.1	0.9576	0.9576	0.96	0	0.0024	0.0024

Table 3.4: Position P Controller

The results of the P controller show very similar values between the models of Simulink and C#, and just a little difference between them and the value of the real equations.

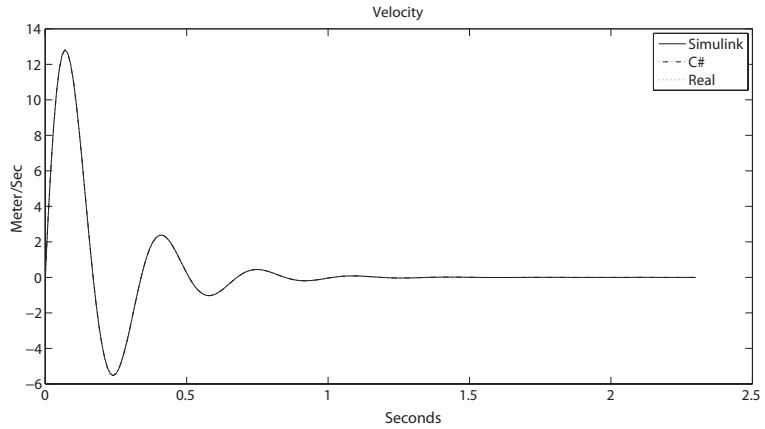


Figure 3.21: Velocity P Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	3.3042	3.3042	3.3102	0	0.006	0.006
0.02	6.1811	6.1811	6.1892	0	0.0081	0.0081
0.03	8.5705	8.5705	8.577	0	0.0065	0.0065
0.04	10.4349	10.4349	10.4364	0	0.0015	0.0015
0.05	11.7588	11.7588	11.7526	0	-0.0062	-0.0062
0.06	12.547	12.547	12.531	0	-0.016	-0.016
0.07	12.8225	12.8225	12.7955	0	-0.027	-0.027
0.08	12.6241	12.6241	12.5856	0	-0.0385	-0.0385
0.09	12.0035	12.0035	11.954	0	-0.0495	-0.0495
0.1	11.0221	11.0221	10.9628	0	-0.0593	-0.0593

Table 3.5: Velocity P Controller

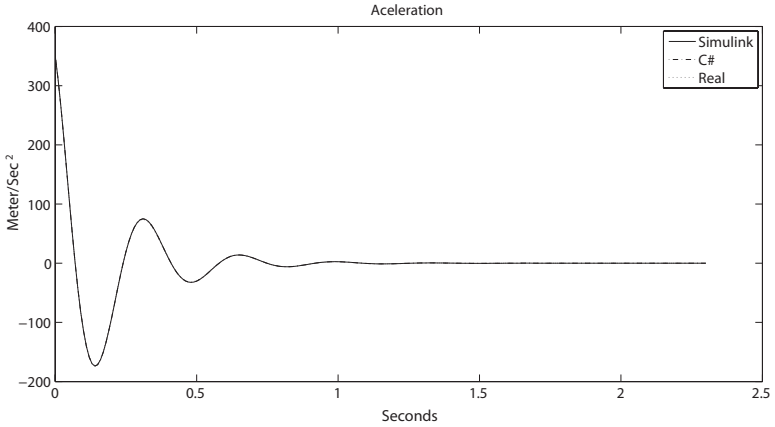


Figure 3.22: Acceleration P Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	350	350	350	0	0	0
0.01	310.845	310.845	310.653	0	-0.1921	-0.1921
0.02	264.528	264.528	264.146	0	-0.3829	-0.3829
0.03	213.344	213.344	212.792	0	-0.5524	-0.5524
0.04	159.54	159.54	158.856	0	-0.6843	-0.6843
0.05	105.243	105.243	104.477	0	-0.7662	-0.7662
0.06	52.3953	52.3953	51.6049	0	-0.7904	-0.7904
0.07	2.7066	2.7066	1.953	0	-0.7536	-0.7536
0.08	-42.3856	-42.3856	-43.0423	0	-0.6567	-0.6567
0.09	-81.7403	-81.7403	-82.245	0	-0.5047	-0.5047
0.1	-114.525	-114.525	-114.83	0	-0.3055	-0.3055

Table 3.6: Acceleration P Controller

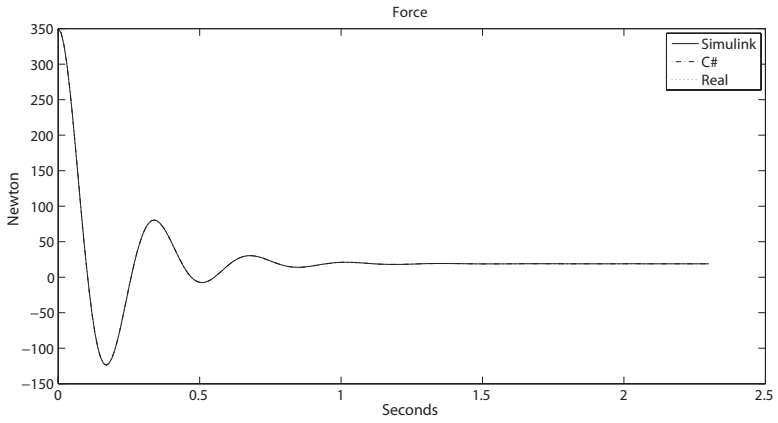


Figure 3.23: Force P Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	350	350	350	0	0	0
0.01	344.218	344.218	344.092	0	-0.1253	-0.1253
0.02	327.618	327.618	327.333	0	-0.2857	-0.2857
0.03	301.803	301.803	301.342	0	-0.4611	-0.4611
0.04	268.544	268.544	267.911	0	-0.6327	-0.6327
0.05	229.705	229.705	228.922	0	-0.7832	-0.7832
0.06	187.17	187.17	186.271	0	-0.8987	-0.8987
0.07	142.773	142.773	141.805	0	-0.9684	-0.9684
0.08	98.2416	98.2416	97.2565	0	-0.9851	-0.9851
0.09	55.1433	55.1433	54.1978	0	-0.9455	-0.9455
0.1	14.8485	14.8485	13.9983	0	-0.8502	-0.8502

Table 3.7: Force P Controller

3.4.3 Results of PI Controller

The results of the PI controller are now showing a small (less than 1%) but notable difference. Let's see the results of the position (Table 3.8) here the Simulink's model is closer to the Real model than the C#'s model, at least on the first ten results, but if we take a look to the table 3.11 it can be appreciated that the first value of the force in the Simulink's model is 1.5 Newtons higher than the other two values, in this table all the values from the C#'s model are closer to the real model than the Simulink's model.

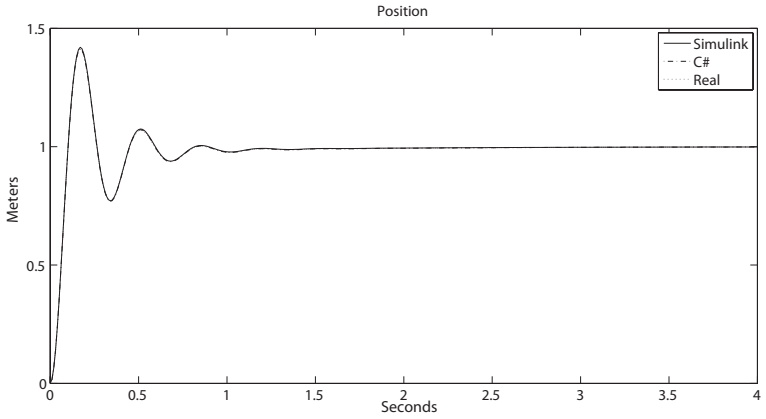


Figure 3.24: Position PI Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	0.01666	0.01659	0.01693	7.1E-05	0.00027	0.00034
0.02	0.06463	0.06436	0.06514	0.00027	0.00051	0.00078
0.03	0.13955	0.13896	0.14024	0.00059	0.00069	0.00128
0.04	0.2365	0.2355	0.23728	0.001	0.00078	0.00178
0.05	0.35025	0.34878	0.35101	0.00147	0.00076	0.00223
0.06	0.47544	0.47345	0.47605	0.00199	0.00061	0.0026
0.07	0.60679	0.60425	0.60709	0.00254	0.0003	0.00284
0.08	0.73924	0.73617	0.73909	0.00307	-0.00015	0.00292
0.09	0.86817	0.86458	0.86743	0.00359	-0.00074	0.00285
0.1	0.98946	0.98538	0.988	0.00408	-0.00146	0.00262

Table 3.8: Position PI Controller

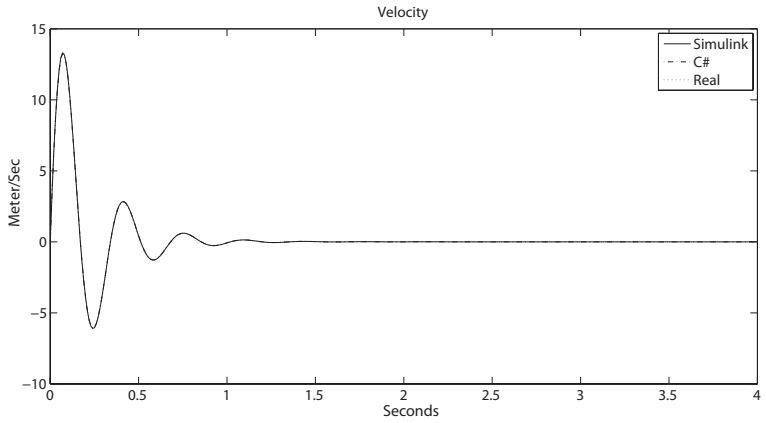


Figure 3.25: Velocity PI Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	3.3324	3.3183	3.3246	0.0141	-0.0078	0.0063
0.02	6.2615	6.235	6.2441	0.0265	-0.0174	0.0091
0.03	8.7216	8.6849	8.693	0.0367	-0.0286	0.0081
0.04	10.669	10.624	10.628	0.045	-0.041	0.004
0.05	12.081	12.031	12.028	0.05	-0.053	-0.003
0.06	12.957	12.903	12.89	0.054	-0.067	-0.013
0.07	13.312	13.257	13.233	0.055	-0.079	-0.024
0.08	13.179	13.126	13.09	0.053	-0.089	-0.036
0.09	12.607	12.556	12.509	0.051	-0.098	-0.047
0.1	11.651	11.605	11.547	0.046	-0.104	-0.058

Table 3.9: Velocity PI Controller

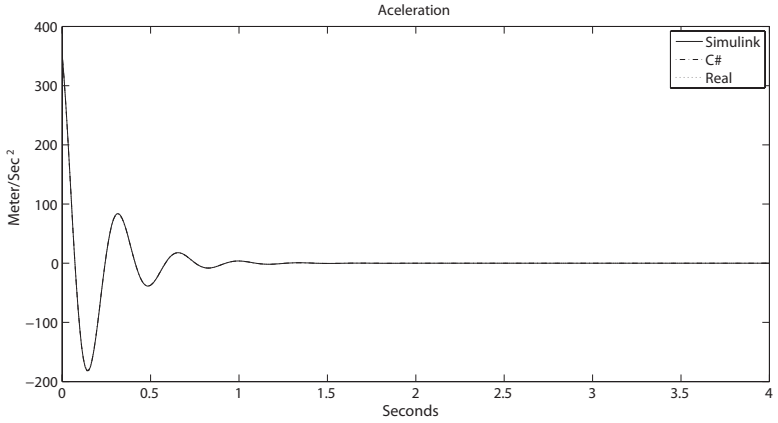


Figure 3.26: Acceleration PI Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	351.5	350	350	1.5	-1.5	0
0.01	314.99	313.65	313.47	1.34	-1.52	-0.18
0.02	270.82	269.69	269.33	1.13	-1.49	-0.36
0.03	221.2	220.29	219.75	0.91	-1.45	-0.54
0.04	168.29	167.61	166.93	0.68	-1.36	-0.68
0.05	114.19	113.75	112.97	0.44	-1.22	-0.78
0.06	60.882	60.668	59.844	0.214	-1.038	-0.824
0.07	10.112	10.115	9.3065	-0.003	-0.8055	-0.8085
0.08	-36.592	-36.39	-37.124	-0.202	-0.532	-0.734
0.09	-77.978	-77.602	-78.205	-0.376	-0.227	-0.603
0.1	-113.09	-112.57	-112.99	-0.52	0.1	-0.42

Table 3.10: Acceleration PI Controller

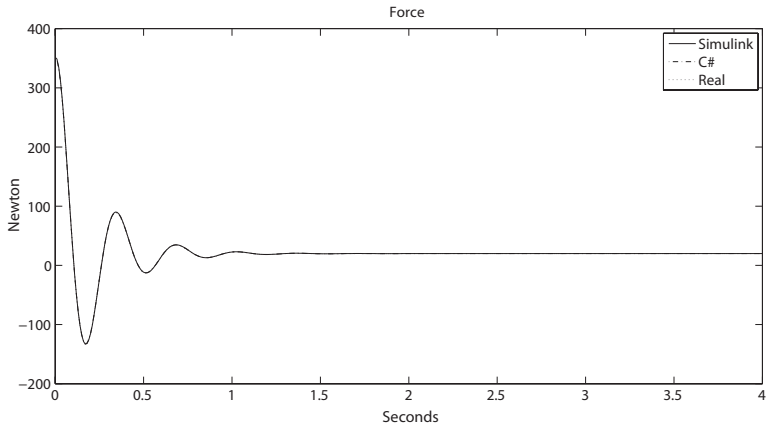


Figure 3.27: Force PI Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	351.5	350	350	1.5	-1.5	0
0.01	348.64	347.17	347.06	1.47	-1.58	-0.11
0.02	334.73	333.33	333.07	1.4	-1.66	-0.26
0.03	311.21	309.91	309.48	1.3	-1.73	-0.43
0.04	279.71	278.56	277.95	1.15	-1.76	-0.61
0.05	242.01	241.04	240.27	0.97	-1.74	-0.77
0.06	199.96	199.17	198.27	0.79	-1.69	-0.9
0.07	155.37	154.77	153.78	0.6	-1.59	-0.99
0.08	109.99	109.59	108.56	0.4	-1.43	-1.03
0.09	65.45	65.247	64.231	0.203	-1.219	-1.016
0.1	23.213	23.191	22.246	0.022	-0.967	-0.945

Table 3.11: Force PI Controller

3.4.4 Results of PID Controller

The PID controller presents very notable and important differences, but if we take a good look to the plots, the big difference occurs in the Simulink's model, this is caused by the D component. If we have a discrete model the first value of the differentiation cannot be calculated with just one point, because there is no tendency on the data, therefore in C#'s model the first differentiation is replaced by a 0, but Simulink does something different, it calculates the first differentiation using: $f'(x_0) = \frac{f(x_0)}{h}$. Taking a look on the position plot (Fig. 3.28), after 4 seconds all the values stabilized to the desired x_d.

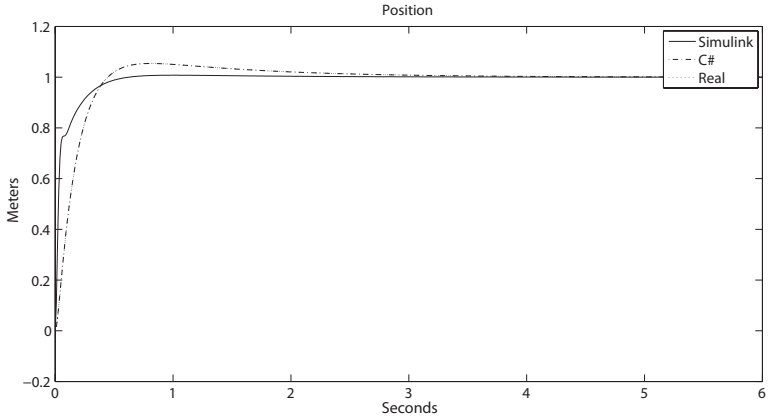


Figure 3.28: Position PID Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	0.11269	0.01484	0.01447	0.09785	-0.09822	-0.00037
0.02	0.34513	0.0519	0.04848	0.29323	-0.29665	-0.00342
0.03	0.54935	0.09812	0.09249	0.45123	-0.45686	-0.00563
0.04	0.67901	0.14684	0.14098	0.53217	-0.53803	-0.00586
0.05	0.74191	0.19627	0.19078	0.54564	-0.55113	-0.00549
0.06	0.7636	0.24513	0.2401	0.51847	-0.5235	-0.00503
0.07	0.76752	0.29254	0.28797	0.47498	-0.47955	-0.00457
0.08	0.76833	0.33801	0.33388	0.43032	-0.43445	-0.00413
0.09	0.77232	0.38134	0.37759	0.39098	-0.39473	-0.00375
0.1	0.78053	0.42246	0.41903	0.35807	-0.3615	-0.00343

Table 3.12: Position PID Controller

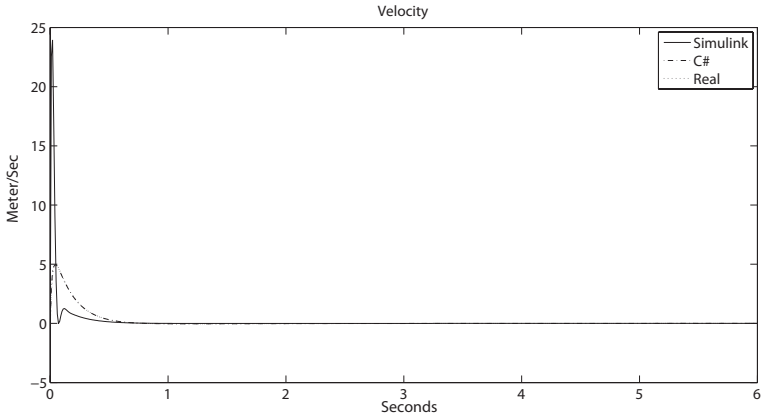


Figure 3.29: Velocity PID Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	0	0	0	0	0	0
0.01	22.538	2.968	2.6282	19.57	-19.9098	-0.3398
0.02	23.951	4.4432	4.0197	19.5078	-19.9313	-0.4235
0.03	16.894	4.8017	4.6933	12.0923	-12.2007	-0.1084
0.04	9.0373	4.9418	4.9531	4.0955	-4.0842	0.0113
0.05	3.5438	4.9442	4.9775	-1.4004	1.4337	0.0333
0.06	0.79389	4.8291	4.871	-4.03521	4.07711	0.0419
0.07	-0.00887	4.6511	4.6945	-4.65997	4.70337	0.0434
0.08	0.16918	4.4437	4.4834	-4.27452	4.31422	0.0397
0.09	0.62936	4.2233	4.2581	-3.59394	3.62874	0.0348
0.1	1.0133	4	4.0303	-2.9867	3.017	0.0303

Table 3.13: Velocity PID Controller

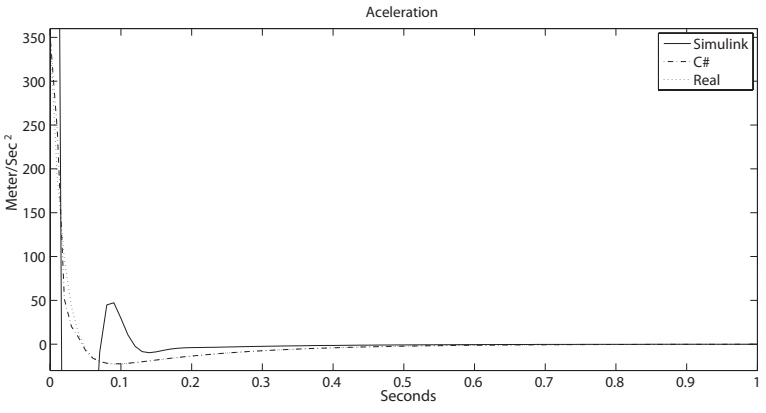


Figure 3.30: Acceleration PID Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	3684.8	350	350	3334.8	-3334.8	0
0.01	822.74	243.61	189.94	579.13	-632.8	-53.67
0.02	-540.21	51.424	96.773	-591.634	636.983	45.349
0.03	-871.18	20.286	42.866	-891.466	914.046	22.58
0.04	-700.09	7.718	11.987	-707.808	712.077	4.269
0.05	-398.61	-7.2221	-5.4034	-391.388	393.207	1.8187
0.06	-151.36	-15.807	-14.909	-135.553	136.451	0.898
0.07	-9.1921	-19.788	-19.822	10.5959	-10.6299	-0.034
0.08	44.802	-21.697	-22.073	66.499	-66.875	-0.376
0.09	47.233	-22.38	-22.797	69.613	-70.03	-0.417
0.1	29.563	-22.276	-22.658	51.839	-52.221	-0.382

Table 3.14: Acceleration PID Controller

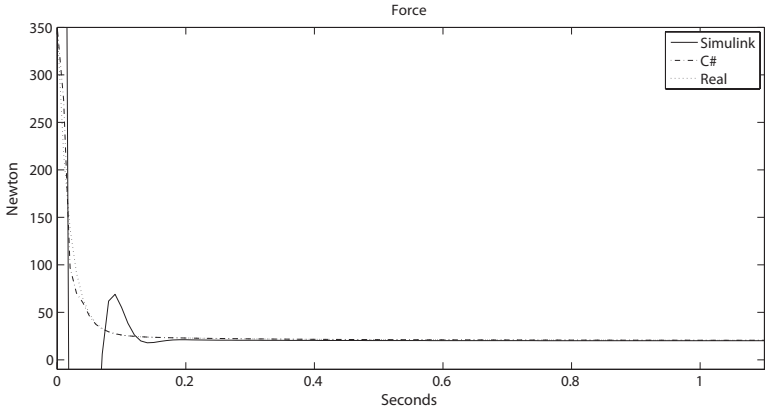


Figure 3.31: Force PID Controller

Time	Simulink	C#	Real	Sim-C#	Real-Sim	Real-C#
0	3684.8	350	350	3334.8	-3334.8	0
0.01	1050.4	273.58	216.51	776.82	-833.89	-57.07
0.02	-293.8	96.894	137.94	-390.694	431.74	41.046
0.03	-691.25	70.266	91.648	-761.516	782.898	21.382
0.04	-596.13	60.072	64.337	-656.202	660.467	4.265
0.05	-348.34	46.146	48.188	-394.486	396.528	2.042
0.06	-128.15	37.386	38.603	-165.536	166.753	1.217
0.07	6.0697	32.573	32.883	-26.5033	26.8133	0.31
0.08	61.861	29.5	29.438	32.361	-32.423	-0.062
0.09	68.973	27.479	27.335	41.494	-41.638	-0.144
0.1	55.307	26.173	26.026	29.134	-29.281	-0.147

Table 3.15: Force PID Controller

3.4.5 Conclusions of the Mass-Spring-Damper's equations

The solution of the Mass-Spring-Damper's equations using C# produced very satisfactory approximation to the differential equation's model, this similitude between the models can be appreciate in both figures and tables of 3.4.1, therefore applying the same methods to the ROV's equations, should provide us a very good approximation to the real behaviour of the ROV.

3.5 Thrusters

The KAXAN ROV is provided with 4 thrusters, two placed in the back to provide back and forth movement, one on the side to provide orientation and the last one on vertical position to control the depth of the ROV, the location of these thrusters can be seen in figure 2.2.

The thrusters selected to move the ROV correspond with two different models, both thrusters on the back are *520 DC Brushless thrusters* and the remaining two are *540 DC Brushless thrusters*, the four thrusters are from the *Tecnadyne* company.

To include the dynamic of the thrusters in the program they must be modelled, taking a look at the datasheets (Appendix B) their behavior is not linear in both forward and backward mode. A linearization or other methods should be applied to obtain a mathematical model of the thruster.

3.5.1 Thruster 520

Let's take a look at the datasheet of the thruster's model 520 (B.1), first notice that the control operation voltage is ±5v, now the graphic *Thrust, lbs* vs *Control Voltage* shows both forward and backward behaviour, the forward mode presents a fairly incremental tendency during the range 0.75v to 4.75v, the table 3.16 was filled with the measurements taken of this plot, these values were analysed in Matlab to obtain an interpolation polynomial.

V	Lbf	N
0.75	0	0
1	0.149664	0.665739
1.5	0.348198	1.548862
2	0.723885	3.220002
2.25	1.533293	6.820426
2.5	2.669517	11.87461
2.75	3.955406	17.59453
3	5.189371	23.08347
3.25	6.585217	29.29251
3.5	8.124618	36.14011
3.75	10.07636	44.82188
4	12.08002	53.73463
4.25	14.22419	63.27236
4.5	16.63103	73.97852
4.75	19.60904	87.22537
4.86	21.07208	93.7333

Table 3.16: Graphic's values from the 520 forward mode

Using the function $polyfit(X, Y, N)$ where X is the vector of the $x-values$ in this case the voltage control, Y is the vector of the $y-values$ in this case the force, a polynomial interpolation of N degree can be

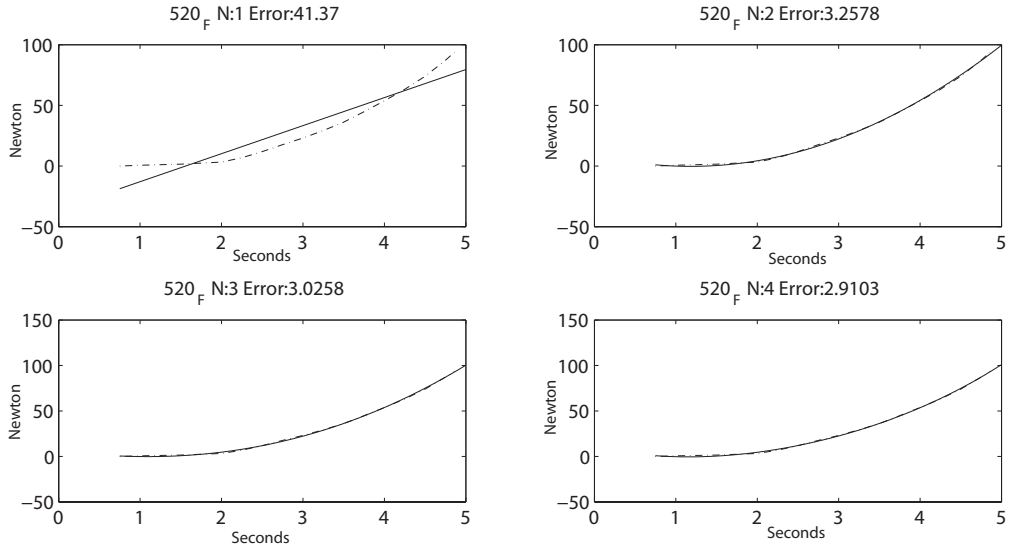


Figure 3.32: 1 to 4 degrees polynomial (Forward mode 520)

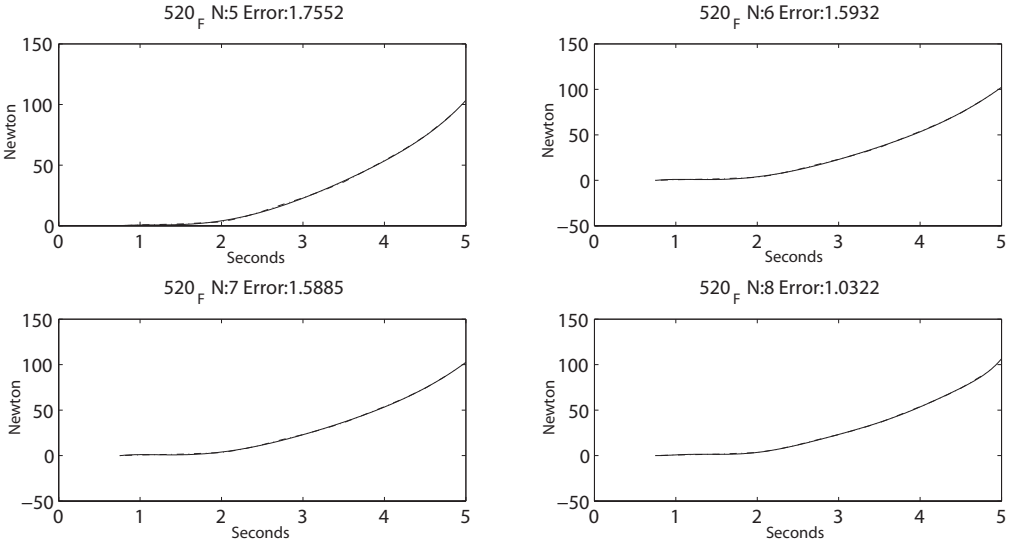


Figure 3.33: 5 to 8 degrees polynomial (Forward mode 520)

calculated. In figures 3.32 and 3.33 both the original polygon and the interpolated polynomial are plotted, each plot displays an *error* which is given by the formula: $error = \sqrt{\sum_{i=0}^{k} [f(a_i) - p_N(a_i)]^2}$. Polynomials of higher degrees were not shown because Matlab showed a warning of *badly conditioned polynomial*, when proving these degrees, instead of getting a more accurate polynomial, a worse fit was obtained.

The polynomial of degree 5 was selected because the improvement between a degree 4 polynomial and a degree 5 is approximately 1.2 units (Table 3.17), which is a very good improvement, but the higher levels do not present a greater improvement and do require more computing effort. The equation of the selected interpolated polynomial is:

$$p_5(x) = 0.3529x^5 - 4.8631x^4 + 25.1635x^3 - 53.5595x^2 + 50.0067x - 16.4616 \tag{3.13}$$

N	Error
1	41.37
2	3.2578
3	3.0258
4	2.9103
5	1.7552
6	1.5932
7	1.5885
8	1.0322

Table 3.17: Errors in $n - degree$ polynomials

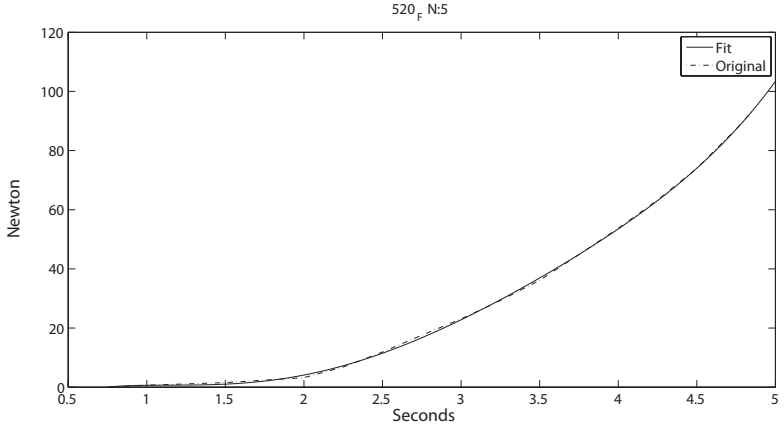


Figure 3.34: Modelled function for the 520 forward mode

Taking a look to the backward mode of the 520 thruster, it can be appreciated that the function is not continuously growing, instead there are certain parts where the function is horizontal. Even though the same procedure of the forward model was used, the results can be seen on table 3.18 and figures 3.35 and 3.36.

From the figures 3.35 and 3.36 can be seen that the error is never smaller than 3 points, and if the degree is big then an exponential grow at the end of the function exists. To model this behaviour a different approach was used, there are some points in table 3.18 that have the same growing tendency, hence instead of having just one big polynomial to represent the thruster, several one degree polynomials can be used to match the behaviour.

V	Lbf	N
-0.75	0	0
-1.00	0.171045	0.76084
-1.50	0.41234	1.83418
-1.77	0.500916	2.22819
-2.00	0.751374	3.34228
-2.14	1.056811	4.70093
-2.39	1.056811	4.70093
-2.50	1.548564	6.88836
-2.75	2.547343	11.33115
-2.87	3.008552	13.38271
-3.12	3.008552	13.38271
-3.25	3.57361	15.89621
-3.50	4.554062	20.25748
-3.75	5.58033	24.82255
-4.00	6.600489	29.36044
-4.25	7.568723	33.66736
-4.36	8.008552	35.62382
-4.50	9.031765	40.17530
-4.62	10	44.48222

Table 3.18: Graphic's values from the 520 backwards mode

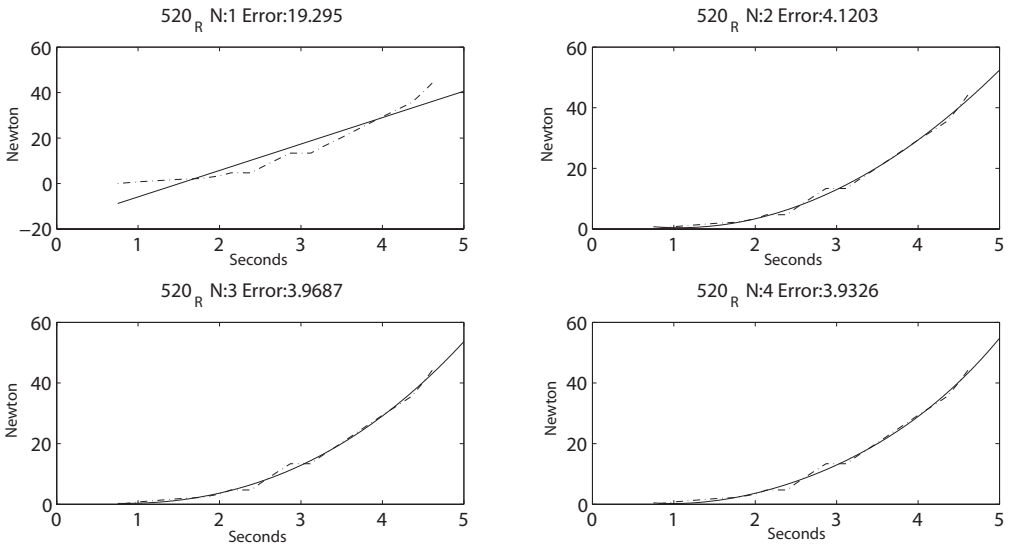


Figure 3.35: 1 to 4 degrees polynomial (Backward mode 520)

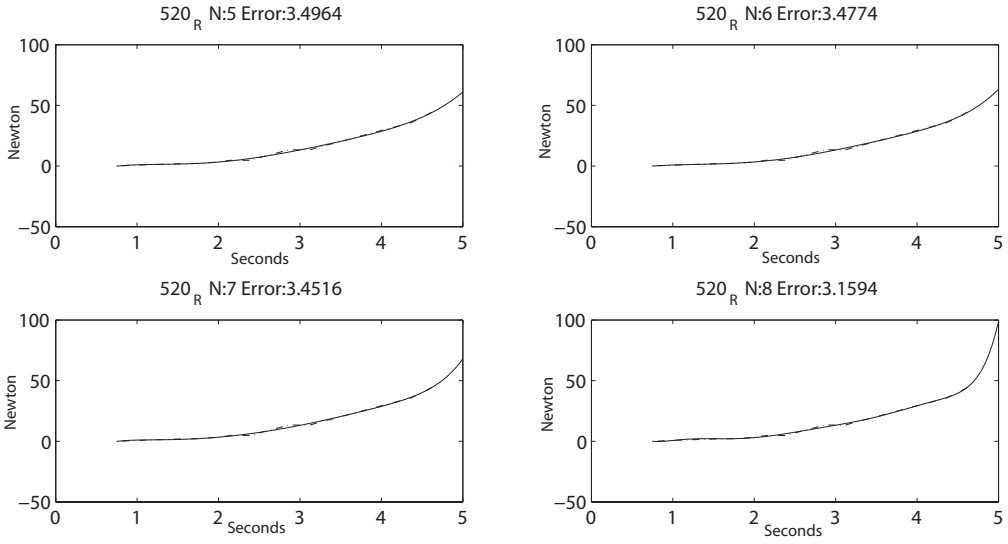


Figure 3.36: 5 to 8 degrees polynomial (Backward mode 520)

Figure 3.37 represents the behaviour of the modelled function. Next is the Matlab's code to represent this backwards mode:

```
v_control = -v_control;

if (v_control<1.77)
        sal=2.17783*(v_control-.75);
elseif(v_control<2)
        sal=4.91053*(v_control-1.77)+2.22819;
elseif(v_control<2.14)
        sal=9.99579*(v_control-2)+3.34228;
elseif(v_control<=2.39)
        sal=4.70093;
elseif(v_control<2.87)
        sal=18.05209*(v_control-2.39)+4.70093;
elseif(v_control<=3.12)
        sal=13.38271;
elseif(v_control<4.36)
        sal=17.89215*(v_control-3.12)+13.38271;
else
        sal=34.29212*(v_control-4.36)+35.62382;
end
sal=-sal;
```

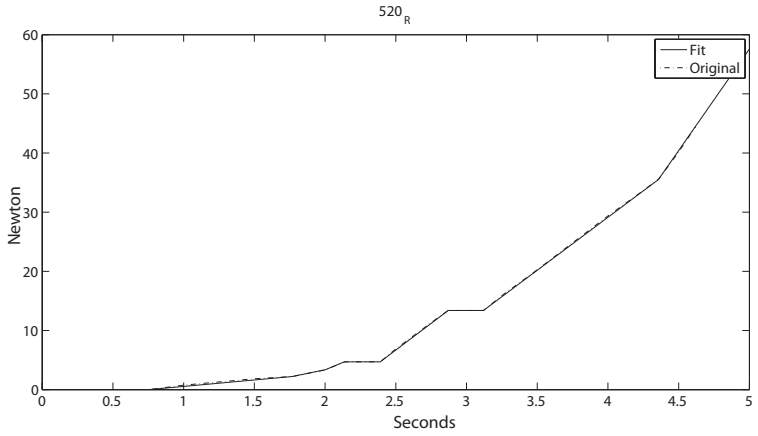


Figure 3.37: Modelled function for the 520 backward mode

3.5.2 Thruster 540

Now that the thruster 520 has been modelled, we must continue with thruster 540. Analysing the datasheet (Appendix B.2) it can be noticed that both forward and backward mode of the 540 model have the same inconstant growing behaviour as the 520 backward mode. Then the same approach was used in both of these modes.

V	Lbf	N
0.75	0	0
1.00	0.211591	0.94120
1.50	0.733722	3.26376
2.00	1.532379	6.81636
2.25	2.548812	11.33768
2.50	3.581588	15.93170
2.75	4.554731	20.26045
3.00	5.512413	24.52044
3.14	6.070766	27.00411
3.25	7.082781	31.50578
3.37	8.112024	36.08408
3.50	8.636805	38.41843
3.62	9.146568	40.68596
3.75	10	44.48222
3.89	11.30047	50.26699
4.00	12.66366	56.33078
4.12	14.16821	63.02336
4.25	14.67091	65.25945
4.38	15	66.72333
4.50	16.65562	74.08791
4.75	19.70536	87.65383
4.87	21.19268	94.26977

Table 3.19: Graphic's values from the 540 forward mode

Here is the Matlab's code for the forward mode:

```
if (v_control<1.5)
```

```
        sal= 4.35168 *(v_control-.75);
elseif(v_control<2)
        sal=7.10521*(v_control-1.5)+3.26376;
elseif(v_control<3.14)
        sal=17.75749*(v_control-2)+6.81636;
elseif(v_control<3.37)
        sal=38.63767*(v_control-3.14)+27.00411;
elseif(v_control<3.62)
        sal=18.29152*(v_control-3.37)+36.08408;
elseif(v_control<3.75)
        sal=29.99730*(v_control-3.62)+40.68596;
elseif(v_control<3.89)
        sal=41.16006*(v_control-3.75)+44.48222;
elseif(v_control<4.12)
        sal=54.51030*(v_control-3.89)+50.26699;
elseif(v_control<4.25)
        sal=17.82614*(v_control-4.12)+63.02336;
elseif(v_control<4.38)
        sal=11.42990*(v_control-4.25)+65.25945;
else
        sal=56.22027*(v_control-4.38)+66.72333;
end
```

Here is the Matlab's code for the backward mode:

```
v_control=-v_control;
if (v_control<1.74)
        sal= 4.69764 *(v_control-.75);
elseif(v_control<2)
        sal=8.58034*(v_control-1.74)+4.66995;
elseif(v_control<2.13)
        sal=17.29162*(v_control-2)+6.86561;
elseif(v_control<2.63)
        sal=9.23392*(v_control-2.13)+9.16918;
elseif(v_control<2.87)
        sal=18.84497*(v_control-2.63)+13.76064;
elseif(v_control<3.88)
        sal=35.60973*(v_control-2.87)+18.33641;
elseif(v_control<4.13)
        sal=50.75553*(v_control-3.88)+54.23024;
elseif(v_control<4.38)
        sal=21.64278*(v_control-4.13)+66.72333;
elseif(v_control<4.64)
        sal=52.36117*(v_control-4.38)+72.13497;
else
        sal=13.96002*(v_control-4.64)+85.69352;
end
sal=-sal;
```

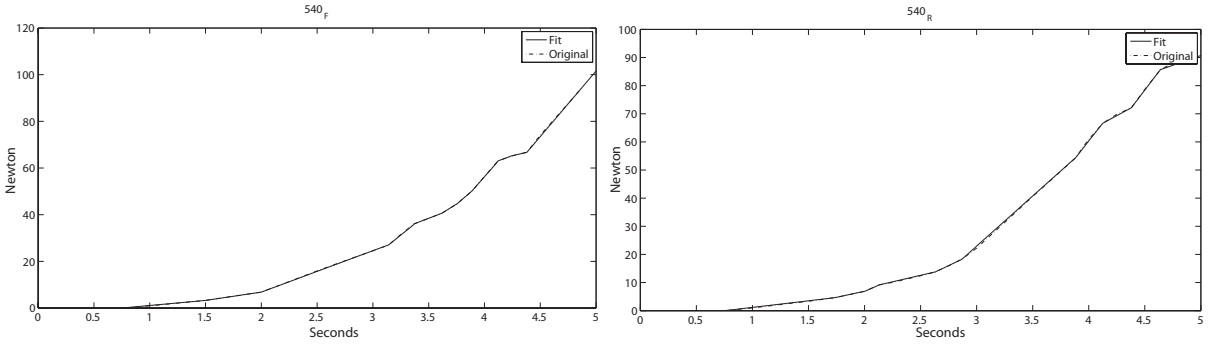
As one can appreciate in figure 3.38, a really good fit between the model and the real behaviour exists.

Whit this last set of equations, both thrusters, 520 and 540, have been modelled for their use in the

simulator. As it can be seen, all of the equations have a very good fit with respect to the real thrusters' behaviours.

V	Lbf	N
0.75	0	0
1.00	0.223644	0.99482
1.50	0.75386	3.35334
1.74	1.049846	4.66995
2.00	1.543449	6.86561
2.13	2.061315	9.16918
2.25	2.258491	10.04627
2.50	2.791795	12.41853
2.63	3.093516	13.76064
2.75	3.585796	15.95042
2.87	4.122188	18.33641
3.00	5	22.24111
3.25	7.052933	31.37301
3.50	9.127922	40.60303
3.75	11.14071	49.55637
3.88	12.19144	54.23024
4.00	13.69872	60.93495
4.13	15	66.72333
4.25	15.69166	69.80000
4.38	16.21659	72.13497
4.50	17.68372	78.66113
4.64	19.26467	85.69352
4.75	19.78077	87.98925
4.87	20	88.96444

Table 3.20: Graphic's values from the 540 backward mode



(a) 540 forward mode

(b) 540 backward mode

Figure 3.38: Modelled functions for the 540

3.6 Training platform (The simulator)

The final product of this project is the simulator or the training platform. In this section the specifics about the program are discussed, for example; how the program looks, what special functions were needed to make it work, how the ROV's equation system was solved.

3.6.1 GUI - Graphical user interface

The GUI of the training platform (Fig. 3.39) was designed to provide the operator with all the required information for his training. In it one can read the position of the ROV on the simulator, one can stablish the parameters for an ocean current, the gains on the controllers, the position of the camera, the voltage control of the thrusters, and more.

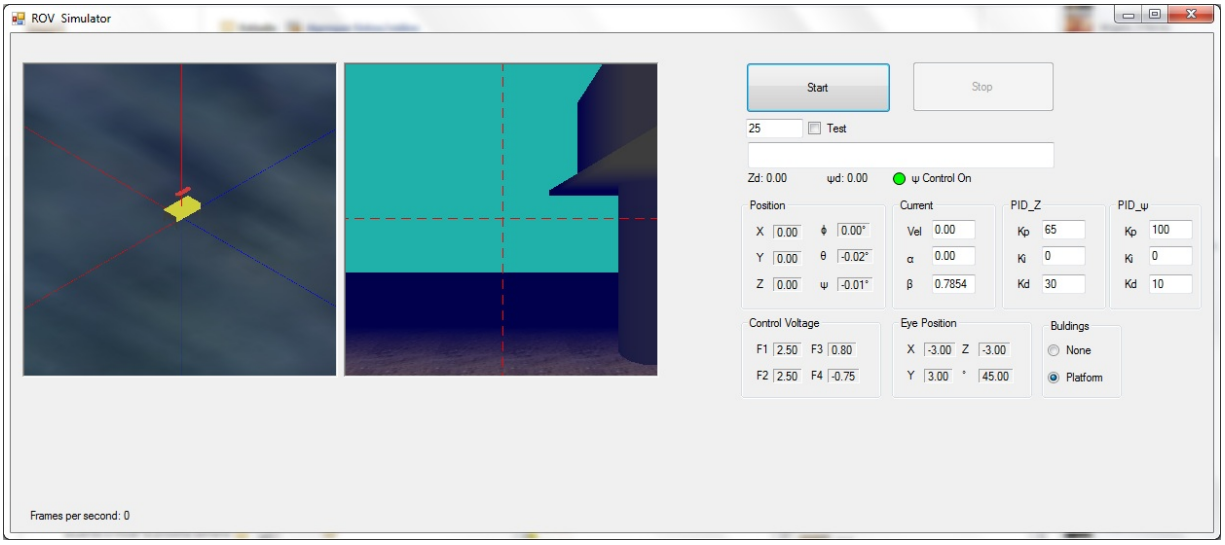
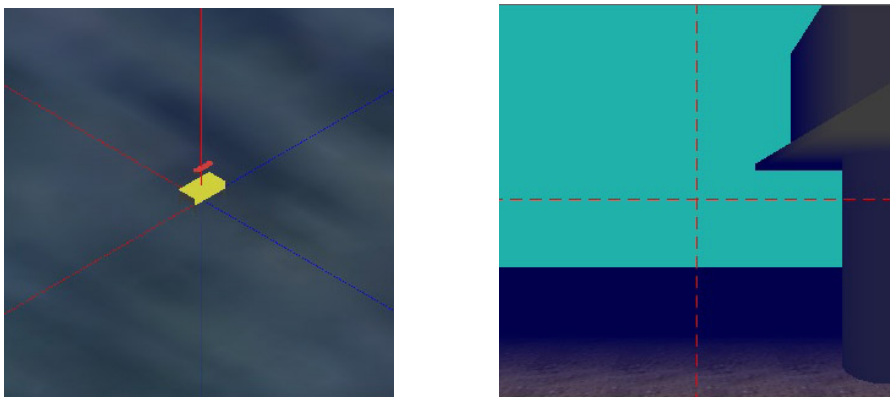


Figure 3.39: Training platform's GUI

In the figure 3.40 both graphical visualizations can be seen, the orthogonal view (Fig. 3.40a) gives us the chance to move the camera around the object to select the desired view, in this option the perspective does not influence in the visualization, every object has the same size no matter were they are placed, this kind of view is good for technical works, is mainly used in CAD software. The second screen (Fig. 3.40b) is the perspective view, here the objects' size change according to the distance between the object and the camera. This is a more realistic perspective and in fact it represents how the operator would see the ocean from the ROV's camera point of view. In the perspective view the movement of the camera is not configured, it moves when the ROV is moving.



(a) Orthogonal view (b) Perspective view

Figure 3.40: Model area

In this trainings platform a single sea current can be configured, the figure 3.41 shows the interface to set

the angles α and β of the current, and its velocity. The velocity is given in $\frac{m}{s}$ and the angles must be written in radians.

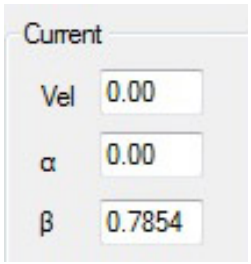


Figure 3.41: Sea current area

In the Depth PID area (Fig. 3.42) the gains for the controller can be set. The default values are: $K_p = 65\, K_i = 0\ and\ K_d = 30$.

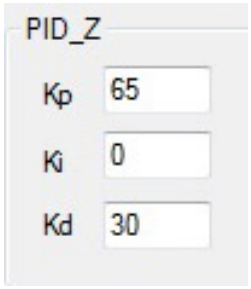
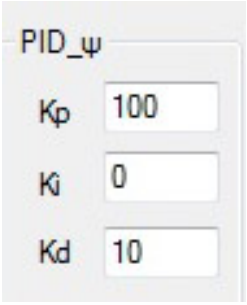


Figure 3.42: Depth PID area

The ψ PID controller also includes an area to write the controller gains (Fig. 3.43a) and a *LED* (Fig. 3.43b) that shows the status of the controller if the PID is on, its color is green, if the PID is off, its color is red. The default gains are: $K_p = 250\, K_i = 300\ and\ K_d = 50$.



(a) Gains of PID controller

(b) On/Off LED

Figure 3.43: ψ PID area

In *Buildings* area one can select the environment in which the ROV is moving (Fig. 3.44), the default option is not to see anything, and the second is a representation of an oil platform (Fig. 3.3).

In the status area of the GUI, as it can be seen on figure 3.45 the user can read the current position and rotations of the ROV as well as the desired Z (Z_d) and the desired ψ (ψ_d). The rotations are given in degrees and the position in meters.

The voltage control for the thrusters is also updated every 16ms, and the units shown are volts (v).

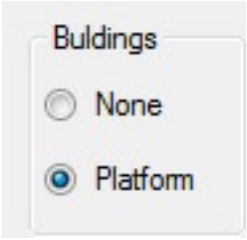


Figure 3.44: Visual environment selection

The eye position section shows the coordinates in which the camera is positioned for the orthogonal view (Fig. 3.40a). The camera is always $\sqrt{18} \approx 4.2426$ units separated in the XY coordinate plane from the center of mass of the ROV, and the angle is the rotation used to calculate this x *and* y coordinates.

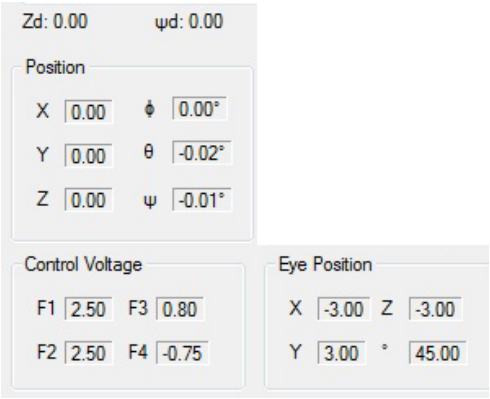


Figure 3.45: Status area

In the last area (Fig. 3.46) of the GUI, the training platform can be initialized or stopped. Checking the *checkbox Test* the user can access to the test mode operation of the simulator, and using the *textbox* at its left, the user can set how much time (in seconds) he wants the test to run.

Figure 3.46: Start/Stop area

3.6.2 Matrices' functions class

The KAXAN's mathematical model is formed by matrices' multiplications and additions, normally there are no native functions to do these matrix operations, hence a new class was created for this project (and others) to cover this necessity. The new class contains functions to do:

- Addition and subtraction of matrices
- Multiplication of matrices

- Multiplication of a matrix times a constant
- Inverse of a matrix, using *Gauss-Jordan Elimination*
- Transpose of a matrix
- Integration of matrices
- Integration of variables
- Derivation of variables
- Printing function to show the matrix as a text
- Printing to file function to save the matrix in a text file
- Reading function to load a matrix from a text file

Every matrix that is used as an input parameter for these functions, must be a 2 dimensions array of double type, they can be declared as follows:

```
double[,] m1;
m1= new double[2,2] {{1, 2},{3,4}};
```

these instructions produce a matrix: $m_1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, also the output of these functions is a 2 dimensions array.

The code of these functions can be found on appendix D.

The addition/subtraction function works when its input parameters are of the same dimension, in case they are not a matrix 1×1 with value 0 is returned. The function has an input parameter called *sum*, if true then the returned value is an addition, if false it is a subtraction of the $m_1 - m_2$ begin m_1 the first input parameter and m_2 the second.

The multiplication function must be fed with 2 matrices of dimensions $m \times n$ and $n \times l$, in case this condition is satisfied the resulting matrix of dimension $m \times l$ product of m_1 and m_2, if the condition is not satisfied then the result is a matrix of dimension $m \times l$ full of zeros but with a diagonal of ones.

The matrix times constant function returns a $k \times m_1$ matrix. The first value must be the array discussed before and the second a double value k.

The inverse function uses the *Gauss-Jordan elimination* to obtain the inverse matrix. In case the m_1 matrix is not square or it is a singular matrix, the function's output is the same m_1 matrix.

The transpose function transforms m_1 of dimension $m \times n$ in a matrix m_2 of dimension $n \times m$ where $m_2 = m_1^T$.

The integration and derivation of variables' function was already defined in section 3.4, the same code was used here to solve the integrations and derivations in this system.

The integration of matrix is a modified version of the variable integration function. As in the original function it's required to save the matrix into an *Arraylist*. When a matrix is being integrated, the first element of the matrix, is integrated with all the first elements of the other matrices.

The matrix printing function returns a string containing the given matrix. For example with the matrix $m_1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the output of the function is: *1 \t 2 \n 3 \t 4*, which is show in a C#'s textbox as $\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$. The second input parameter *format* specifies the number's format desired.

The matrix printing to file function does something very similar to the past function with the difference that the matrix is saved in a text file. The function does not return any value, and in the second input parameter the name of the file must be specified. The matrix is saved in the file with a Matlab compatible format.

The reading matrix from file function is able to read a matrix from a text file with a format similar to Matlab, where the matrix is enclosed by square brackets *[,]*, the columns are divided by a coma *(,)* and the rows divided by a semicolon *(;)*. In case the file specified in *name* is not found a matrix 1×1 with a value of zero is returned.

3.6.3 Equation's solution

The ROV's dynamics follow the equation 2.10:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) = \tau$$

where $\dot{\nu}$ is the acceleration of the system and ν is the velocity of the system. Clearing $\dot{\nu}$ the equation 3.1:

$$\dot{\nu} = M^{-1}[\tau - (C(\nu)\nu + D(\nu)\nu + g(\eta))]$$

To obtain the value of ν it's necessary to integrate $\dot{\nu}$:

$$\nu = \int \dot{\nu}\, dt \tag{3.14}$$

2.7 Integrating ν has no physical interpretation because ν is the vector of velocity referred to the *body-fixed* coordinate system. Therefore ν must be transformed to refer the velocity vector with the *earth-fixed* frame, using equation 3.1 the transformation is achieved:

$$\dot{\eta} = J(\eta_2)\, \nu$$

Now, $\dot{\eta}$ is the velocity vector referred to the *earth-fixed* frame, after its integration the vector η is obtained, which is the position vector of the ROV in *earth* coordinates:

$$\eta = \int \dot{\eta}\, dt \tag{3.15}$$

In section 3.4 was proved that, with the use of numerical methods it's possible to obtain a really good result in a *Newtonian* system. This same approach was used to solve the ROV's equation system.

Because of the use of a sea current in this project, instead of using the equation 2.10 the equation 2.18 must be used:

$$\begin{gathered} M\dot{\nu} + C_{RB}(\nu)\nu + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r + g(\eta) = \tau \\ \dot{\nu} = M^{-1}[\tau - (C_{RB}(\nu)\nu + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r + g(\eta))] \end{gathered} \tag{3.16}$$

where $\nu_r = \nu - \nu_c$ (see equation 2.19).

The logical course of action has been set, now it's time implement this solution on C#. The first step is to declare the matrices. the easiest matrix to declare is matrix M, because its value doesn't change during run time execution.

```
matrix_m = new double[6, 6] {{m-Xup, 0, 0, 0, 0, 0},
                             {0, m-Yvp, 0, 0, 0, 0},
                             {0, 0, m-Zwp, 0, 0, 0},
                             {0, 0, 0, Ixx-Kpp, 0, 0},
                             {0, 0, 0, 0, Iyy-Mqp, 0},
                             {0, 0, 0, 0, 0, Izz-Nrp}};
```

Where m is the mass, $I_x x$, $I_y y$, $I_z z$ are the inertias, and the other values are the hydrodynamic constants for the matrix of added mass.

Other important and easy matrix to define is matrix B used in equation 2.17 to transform the force vector of the thrusters to the τ vector.

```
matrix_f_2_tau = new double[6, 4] {{1,1,0,0},
                                   {0,0,1,0},
                                   {0,0,0,1},
                                   {0,0,-0.07,0.02},
                                   {-0.1,-0.1,0,0.022},
                                   {0.175,-0.2150,0.1350,0}};
```

The rest of the matrices are not constant matrices since they are in function of the velocity vector ν or the position vector η. The coding for these matrices can be found in the appendix E.

The solution of equation 3.16 is done with the following function written on appendix E.8.

Similar to the Mass-Spring-Damper system, this solution must be written inside a recursive logarithm to calculate the values, the condition to leave the recursive loop is being in the loop more than 10 times, or until the converged value has a difference no bigger than ± 0.001 of the new value.

Let's not forget that to obtain the vector $\dot{\eta}$ it's necessary to apply the transformation $J(\eta)$, therefore both the integration 3.14 and the transformation 2.9 must be inside the recursive loop.

The output signal of both of the PID controllers must be also inside of the recursive loop, this is important since the error depends on the position that is being updated in every loop.

3.6.4 PID Controllers' code

In chapters 4 and 5 the theoretical design of the controller is discussed, that means the calculation for the gains of the controller and the responses of the system. This section is how the controllers were implemented inside the coding.

First, 6 new *Arraylists* were created, two of them to store the error of each controller, other two to store the integrations of theses arrays and the last two to store the differentiations of the error arrays.

The gains of the PID controllers cannot be changed until the system has stopped, when the system it's on, it saves the values of the controllers area (figures 3.42 and 3.43) into the constants K_p, K_i *and* K_d of each controller.

Then we run the equation 3.10, which in the case of the depth controller is:

$$\varepsilon(t) = Z_d - Z, \qquad Z = \text{ Third element of } \eta$$

or in the case of the orientation controller:

$$\varepsilon(t) = \psi_d - \psi, \qquad \psi = \text{ Sixth element of } \eta$$

The orientation controller, can be turned on and off by pressing the joystick's *circle* button, when it is on, the thruster number 3 (Fig. 2.2) cannot be controlled with the joystick, because the control voltage of this thruster is given by the orientation controller. When it is off then the thruster is controlled by the joystick.

By default the value of the desired ψ is 0, and the orientation controller is on, when the user wants to change this value it's necessary to turn off the controller, move the ROV to the desired angle and then turn on again the controller.

3.6.5 Normal mode and Test mode

There are two operation modes for the training platform: Normal mode and Test mode.

In normal mode (*Test* box, unchecked), after the button *Start* is pressed the simulation will stop when the user presses the button *Stop*. If the joystick is not connected to the computer or if during the simulation in normal mode the *Test* box is checked, then the value of the force voltage control for the first, the second and the third thruster is 2.5v, but of course the voltage of the third thruster is subject to whether or not the orientation controller is enabled or disabled.

The text mode (*Test* box, checked), calculates the system with a sample frequency of 100Hz, during the specified time on the *textbox* located at the left of the *Test* box. The test is initialized with the *Start* button, and if its required it can be stopped with the *Stop* button. During test mode the desired depth is always 5, the desired orientation is 0° and the voltage on thruster 1 and 2 is 2.5v.

At the end of both modes, the vectors η, ν, $\dot{\nu}$, *and* τ are saved into text files as well as a record of the time and the values of the desired depth and orientation (Z_d *and* ψ_d). This files are used by Matlab to do the comparison between Matlab's results and C#'s results.

3.6.6 Matlab's results vs C#'s

To check the results of C# some Matlab Simulink models were created.

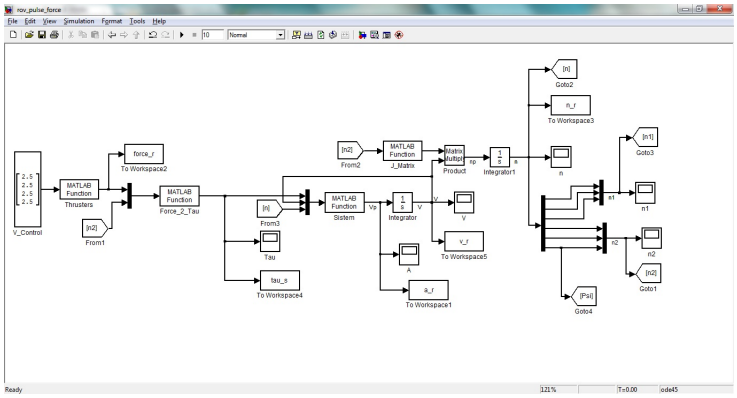


Figure 3.47: Simulink model for a constant control voltage

The figure 3.47 represents the model used to simulate the behaviour of the ROV, with a constant voltage control. The Matlab models are configured as:

- Matlab configured in *Variable-Step* mode
- Max step size, min step size and initial step size in automatic mode
- The Dormand-Prince algorithm is used as integrator, (Matlab's default)

Constant force

The figures 3.48, 3.49 and 3.50 represents the responses of both systems (Matlab and C#) at a constant force. As it can be seen both responses are very similar between each other, which is a good validation for the C#'s results.

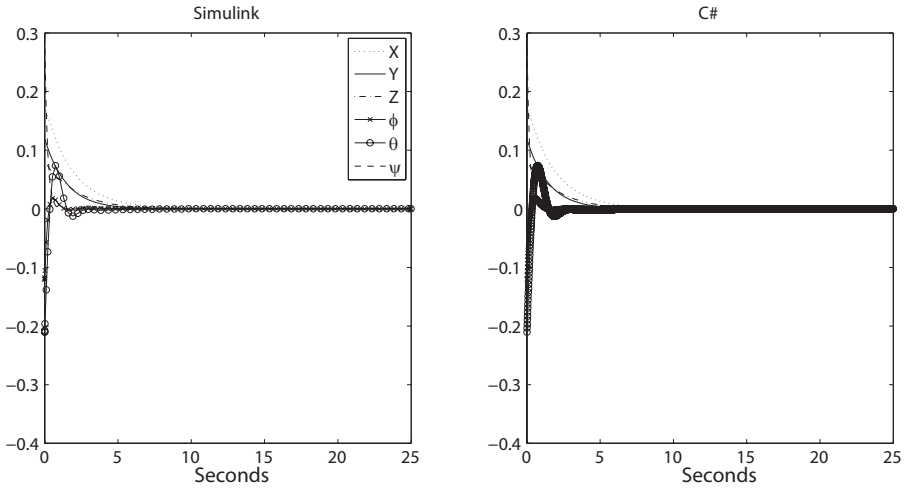


Figure 3.48: Acceleration at constant force

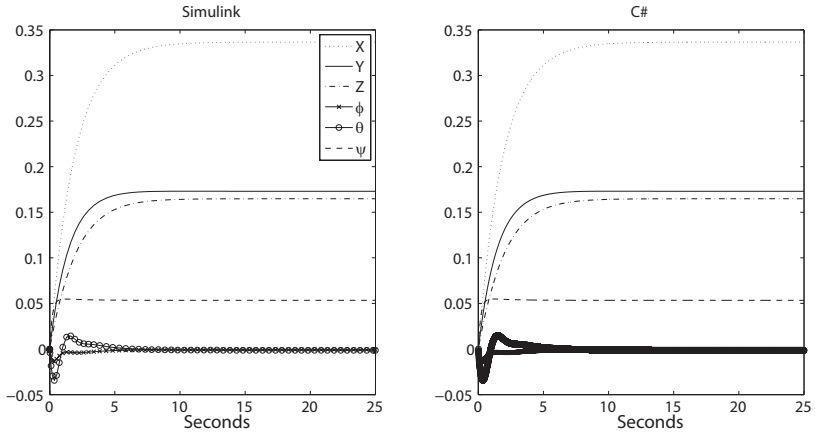


Figure 3.49: Velocity at constant force

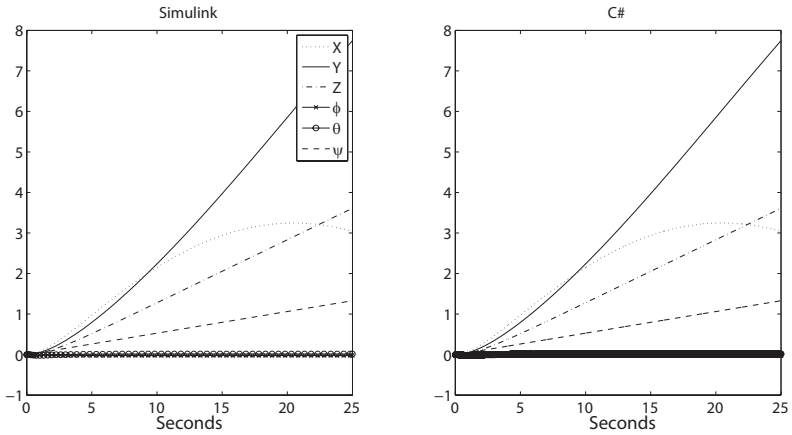


Figure 3.50: Position at constant force

Depth controller

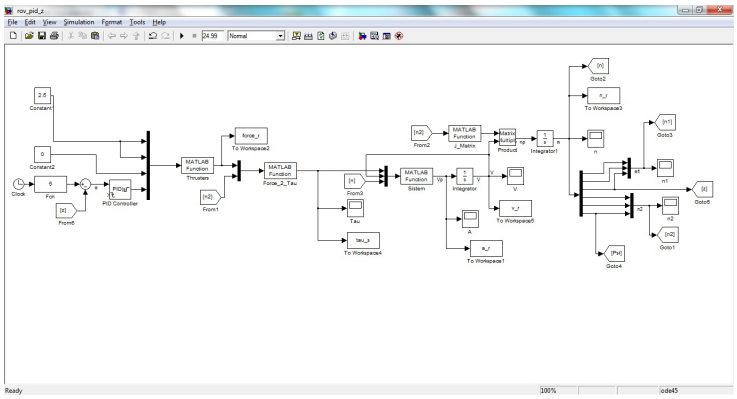


Figure 3.51: Simulink model for the Depth controller

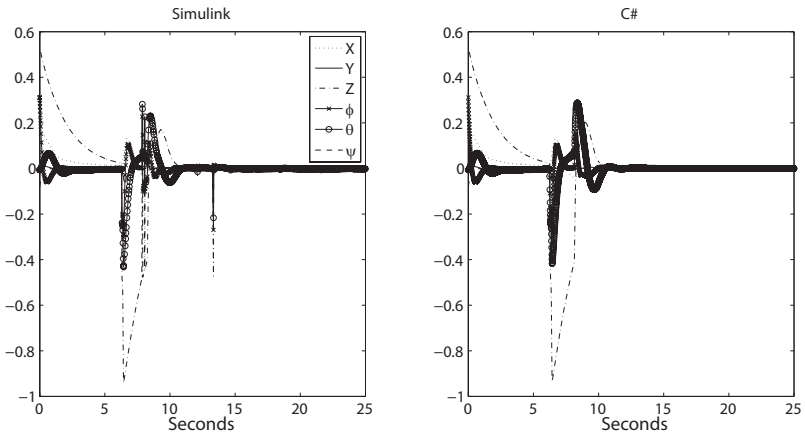


Figure 3.52: Acceleration using the Depth controller

The initial conditions for this test were:

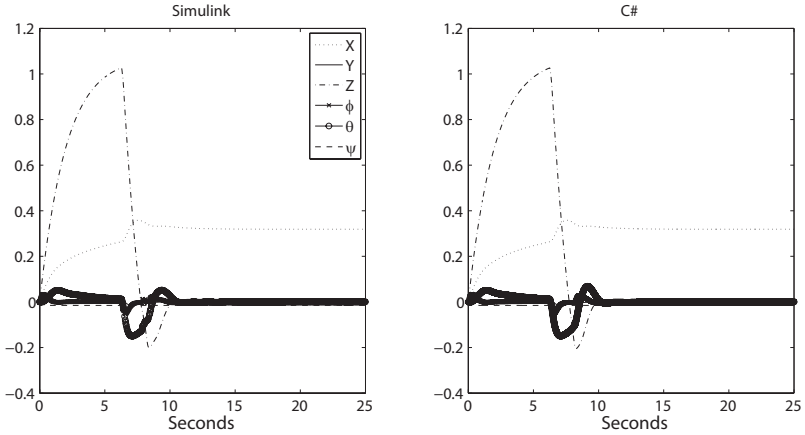


Figure 3.53: Velocity using the Depth controller

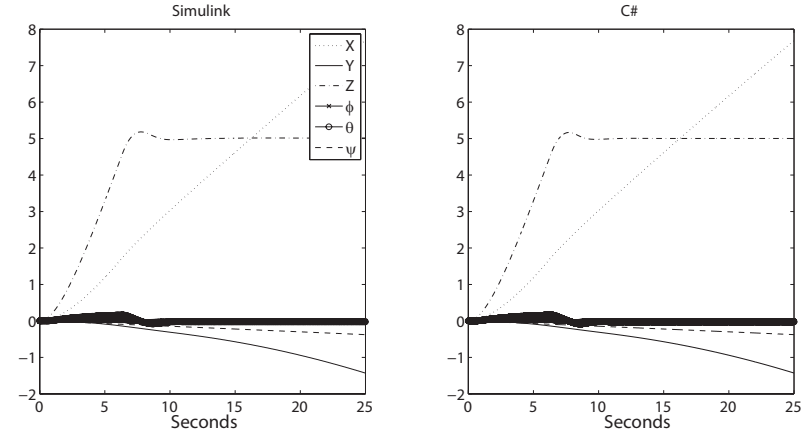


Figure 3.54: Position using the Depth controller

- Desired depth, $Z_d = 5$
- Thruster 1 and 2 controlled with a voltage of 2.5v
- Thruster 3 controlled with a voltage of 0v
- The ψ controller was disabled

As it can be seen on the figures 3.52, 3.53 and 3.54 both Simulink's and C#'s responses are once more very similar to each other.

Orientation controller

The initial conditions for this test were:

- Desired depth, $Z_d = 5$
- Desired orientation, $\psi_d = 10°$
- Thruster 1 and 2 controlled with a voltage of 2.5v

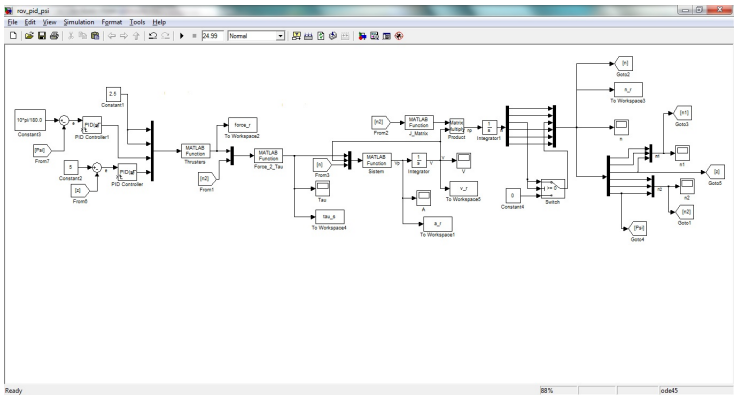


Figure 3.55: Simulink model for the Orientation controller

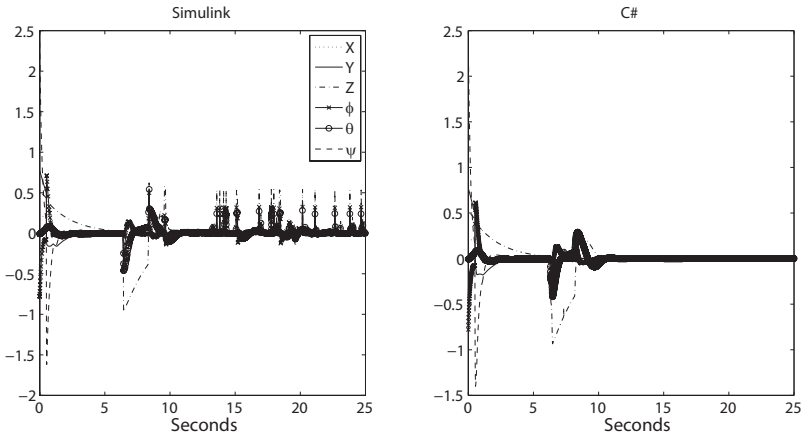


Figure 3.56: Acceleration using the Orientation controller

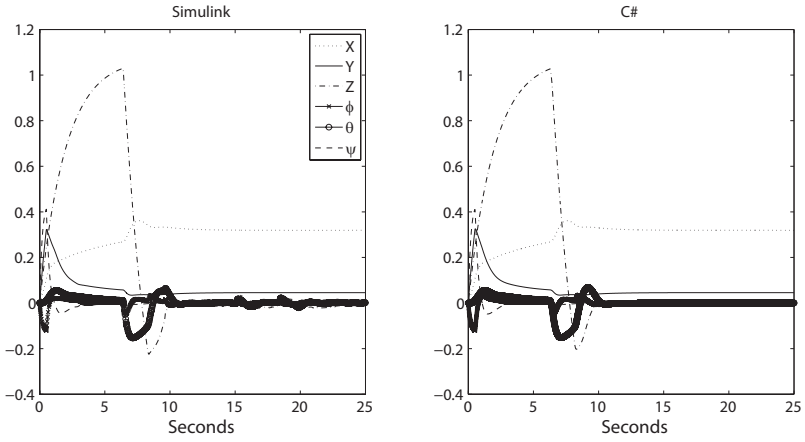


Figure 3.57: Velocity using the Orientation controller

In general the figure 3.58 presents a very similar response between both systems, but the acceleration figure (Fig. 3.56) shows some peaks, in this case C# works with a smaller step size as Matlab, therefore in some points Matlab calculates that the acceleration goes up instead of maintaining the tendency, but

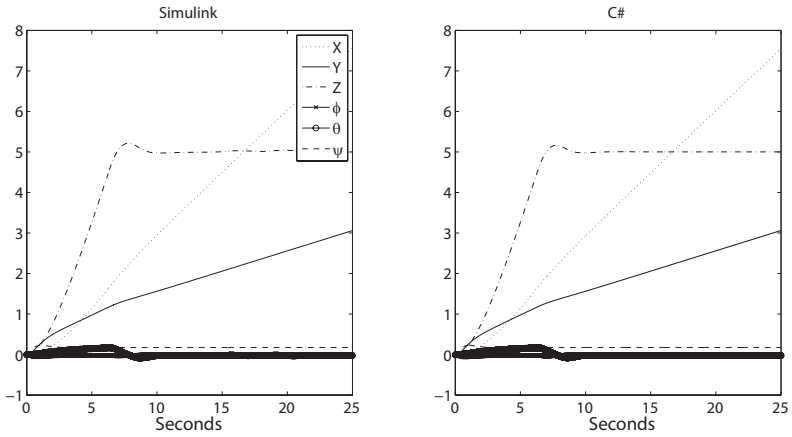


Figure 3.58: Position using the Orientation controller

after a peak is produced immediately this value is brought back by the same controller, which is why the peaks doesn't last long.

Normal mode operation

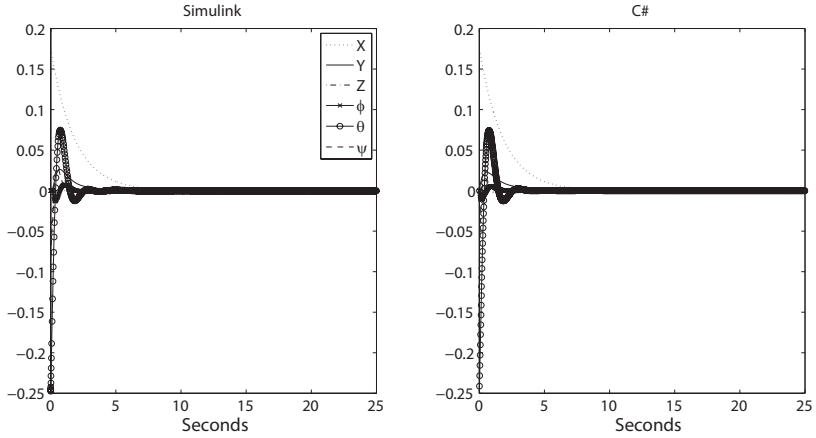


Figure 3.59: Acceleration on Normal mode

The initial conditions for this test were:

- Desired depth, $Z_d = 0$
- Desired orientation, $\psi_d = 0°$
- Thruster 1 and 2 controlled with a voltage of 2.5v

The important difference between the normal mode and the test is that the normal mode works with a step size of 0.016 seconds and the test mode with a step size of 0.01. This last test was made to prove the accuracy of the normal mode against the Matlab model.

As it can be seen on the figures 3.61, 3.60 and 3.59 the behaviour between the two systems is very similar, with the exception of little undulations produced on Matlab's model.

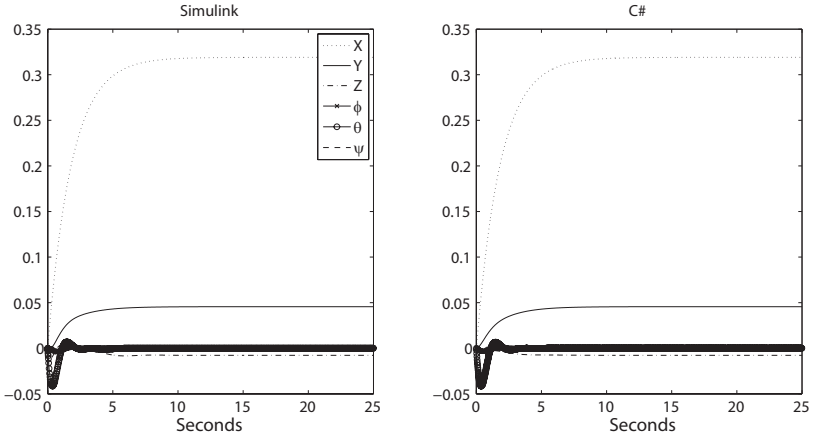


Figure 3.60: Velocity on Normal mode

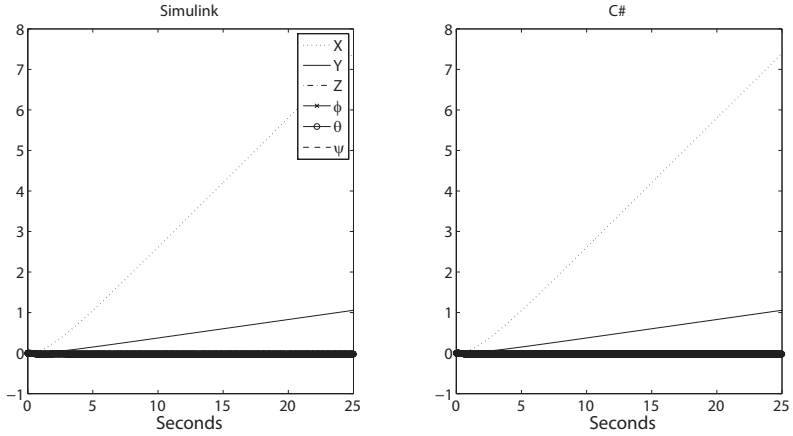


Figure 3.61: Position on Normal mode

4 Depth PID controller

In the master thesis *Control de un robot submarino (Control of an underwater robot)*[12] there is a proposal for the PID controller's gains, as part of this project a new set of gains must be calculated to improve the functioning of the PID controller, it exists also the possible scenario where the same past PID's gains are found.

The gains of the PID controller were calculated using an heuristic method, the main reason is the huge non-linearities of the system, where the linearization methods discard several factors which were of great importance to the system.

The initial considerations were the following:

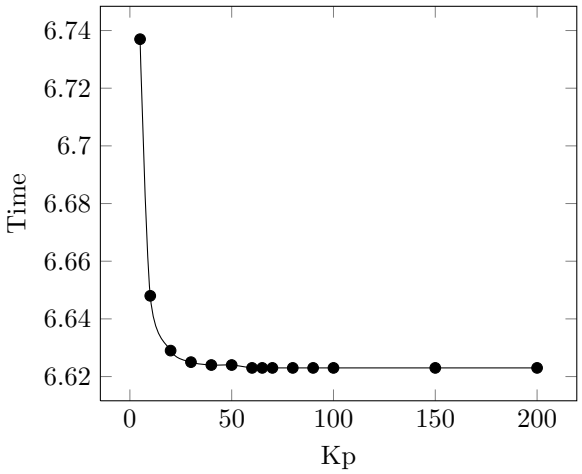
- For tuning purposes, the desired depth was 5m
- The output signal of the controller was limited to ±5v, because that is the range of the thruster's voltage control
- The training platform was used to calculate the response of the PID controller
- The sampling time was from 0 to 25 seconds in almost all the cases
- The sampling frequency was 1kHz (0.001 seconds)
- Here the settling time is defined as the moment when the response is contained inside an error band of ±5% of the desired value.
- A small overshot and a fast stabilization of the system are the desired properties of the PID controller

4.1 Selecting K_p

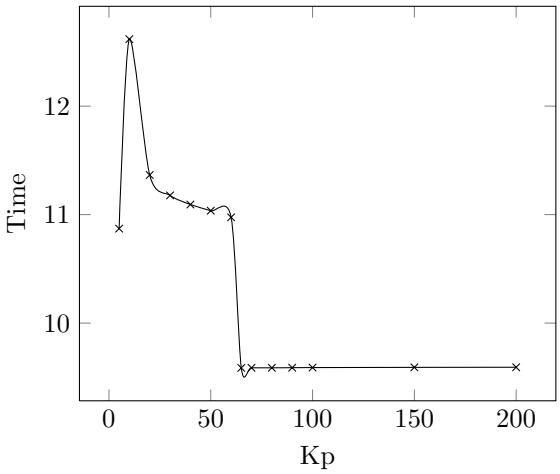
The first step with the heuristic method is to calculate the K_p gain, the table 4.1 shows the obtained values for rise time, settling time and overshot.

Kp	Rise Time	Settlling Time	Overshot
5	6.74	10.87	14.52
10	6.65	12.62	13.36
20	6.63	11.37	13.24
30	6.63	11.18	13.26
40	6.62	11.09	13.28
50	6.62	11.04	13.3
60	6.62	10.98	13.3
65	6.62	9.59	13.31
70	6.62	9.59	13.31
80	6.62	9.59	13.32
90	6.62	9.59	13.33
100	6.62	9.59	13.33
150	6.62	9.59	13.35
200	6.62	9.59	13.35

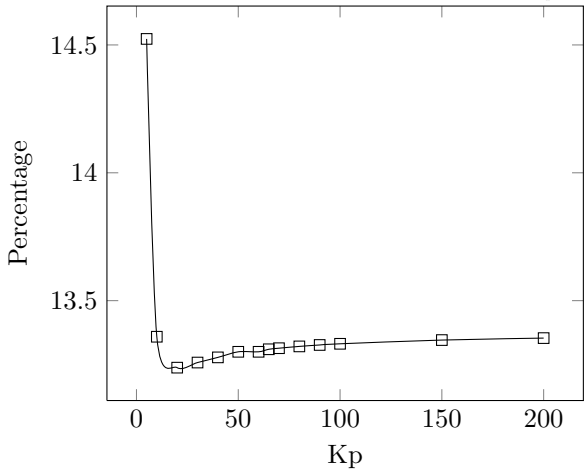
Table 4.1: Responses for the P controller



(a) K_p Rise time



(b) K_p Settling time



(c) K_p Overshot (%)

Figure 4.1: K_p Rise time, Settling time & Overshot

As one can see in the figure 4.1c, with the value of $K_p = 20$ the smallest overshot is reached, after this value the overshot increments, but as it can be seen this increment is not very pronounced and it shows an almost horizontal line.

In the figure 4.1b it can be appreciated that a significant improvement exists between $K_p = 60$ and $K_p = 65$. Between $K_p = 65$ and $K_p = 200$ this value doesn't change a lot, this is mainly due to the sampling frequency.

The value of $K_p = 65$ was chosen because it has a low overshot value and the smallest settling time of the following values.

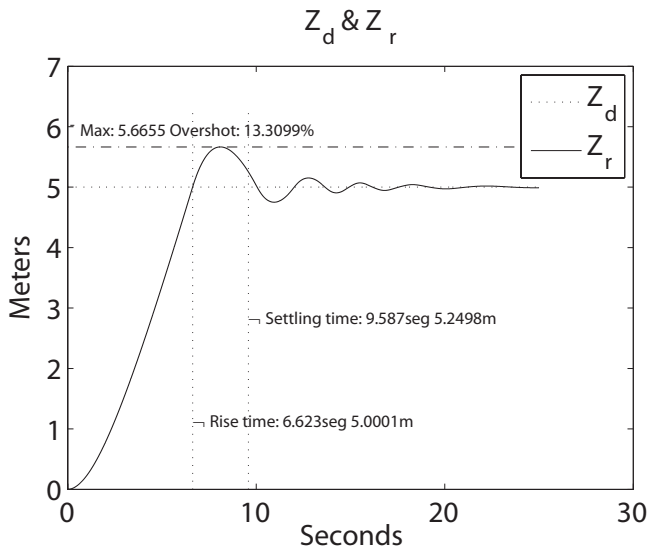


Figure 4.2: $K_p = 65$

4.2 Selecting K_d

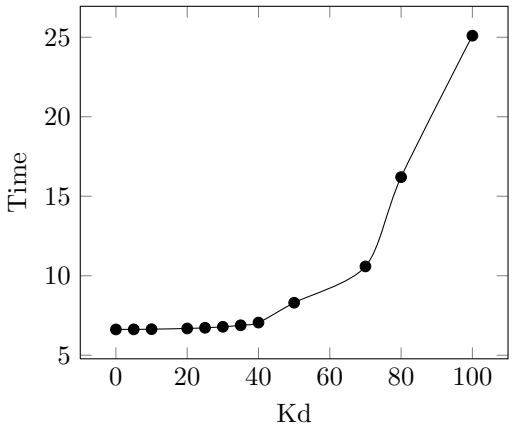
Kd	Rise Time	Settling Time	Overshot
0	6.62	9.59	13.31
5	6.63	9.35	11.74
10	6.64	9.09	10.17
20	6.68	8.5	7.07
25	6.73	8.07	5.54
30	6.79	6.41	4.01
35	6.89	6.43	2.5
40	7.05	6.47	1
50	8.3	6.6	0.18
70	10.59	7.01	$1.78 \cdot 10^{-2}$
80	16.2	7.25	$1.32 \cdot 10^{-5}$
100	25.1	7.78	$-2.33 \cdot 10^{-5}$

Table 4.2: Responses for the PD controller

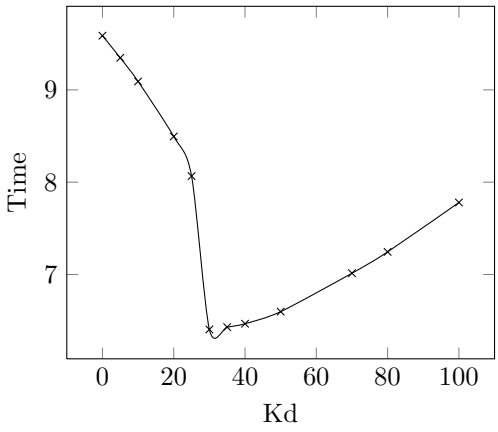
As it can be seen on figure 4.3c, while the value of K_d increments the overshot decreases until it becomes almost 0, this is exactly the desired operation of the system.

But as K_d increments also the rise time increments (Fig. 4.3a), having $K_d = 0$ and $K_d = 40$ an almost horizontal behaviour.

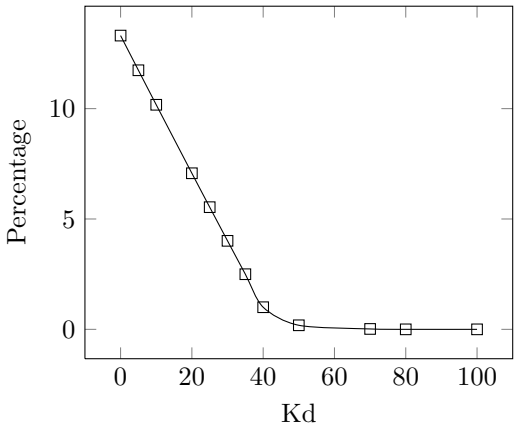
The settling time was the definitive factor in choosing the value of K_d, here there is a significant change between the values of $K_d = 25$ and $K_d = 30$ (Fig. 4.3b). With $K_d = 30$ there is an improvement of more than one and a half seconds in settling time, besides the rising time is $\pm 0.1 sec$ different in comparison with its immediate neighbour values, and its overshot is just of 4%.



(a) K_d Rise time



(b) K_d Settling time



(c) K_d Overshot (%)

Figure 4.3: K_d Rise time, Settling time & Overshot

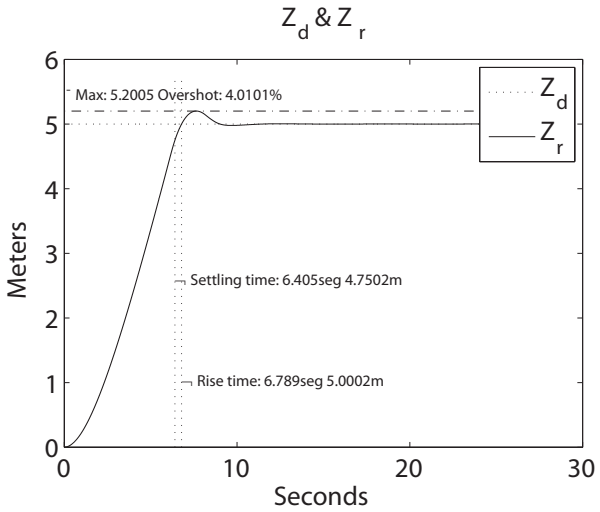
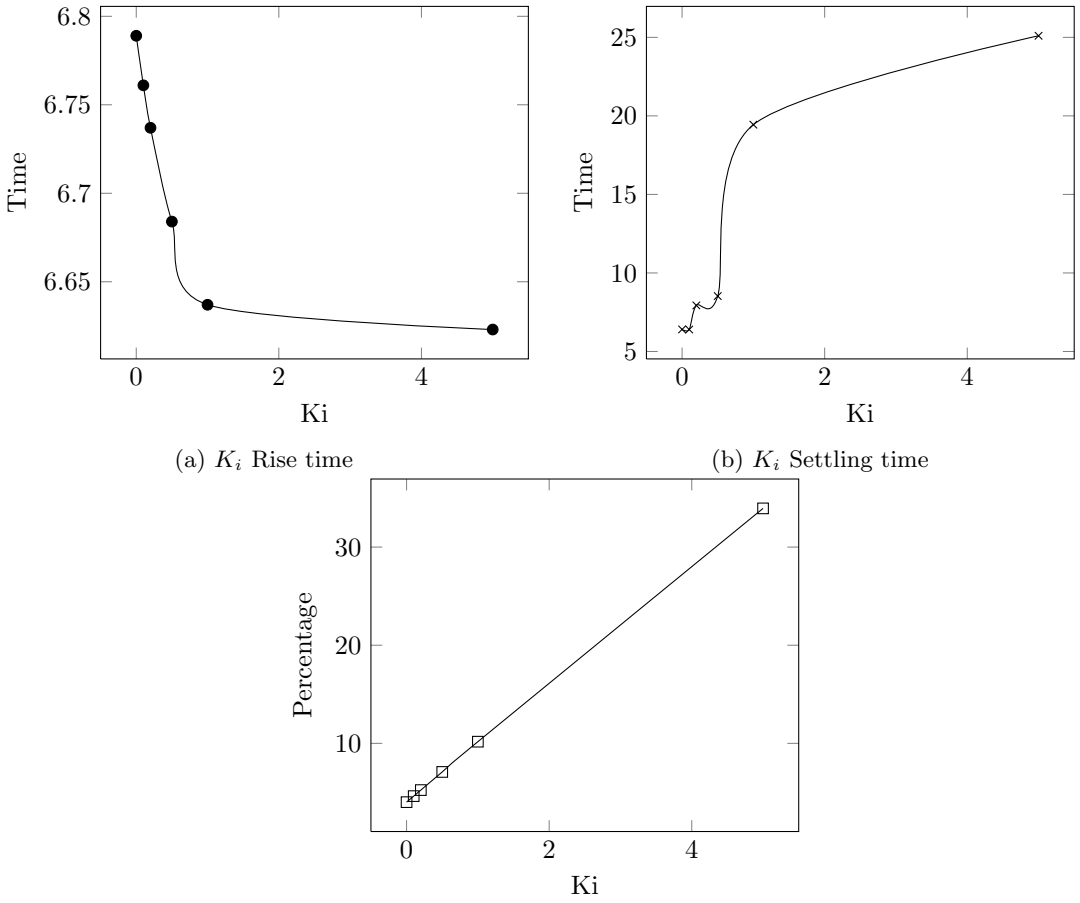


Figure 4.4: $K_p = 65$, $K_d = 30$

4.3 Selecting K_i

Ki	Rise Time	Settling Time	Overshot
0	6.79	6.41	4.01
0.1	6.76	6.4	4.62
0.2	6.74	7.94	5.24
0.5	6.68	8.53	7.09
1	6.64	19.44	10.17
5	6.62	25.1	33.94

Table 4.3: Responses for the PID controller



(a) K_i Rise time

(b) K_i Settling time

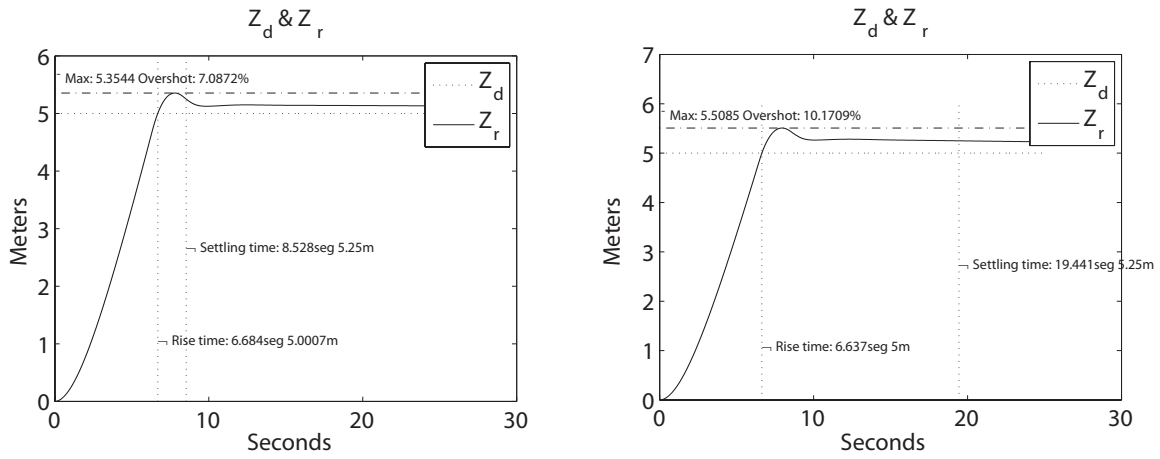
(c) K_i Overshot (%)

Figure 4.5: K_i Rise time, Settling time & Overshot

To select the correct value of K_i it's important to look at figures 4.6a and 4.6b and the table 4.3. For big values of K_i it corresponds a big value of the overshot, a big settling time and there is not a significant gain on the rise time.

For very small values of the gain ($K_i < 1$) it exists also a great increment on the overshot, settling time and the error in steady state, which is not present when $K_i = 0$.

Finally instead of using a PID controller a PD controller was chosen, the shape of the response of this controller presents a small overshot in comparison with other options, a small settling time and by special



(a) $K_p = 65,\ K_i = 0.5\ K_d = 30$ (b) $K_p = 65,\ K_i = 1\ K_d = 30$

Figure 4.6: $K_i = 0.5$ & $K_i = 1$

reasons of the system it does not presents a steady error state. The figure 4.4 shows the behaviour of the selected controller.

4.4 New controller vs. Old controller

Controller	Zd	Rise Time	Settling time	Overshot
Old	1	2.32	5.6	29.34
Kp=100	3	4.63	7.27	16.57
Ki=0	5	6.63	9.27	11.27
Kd=10	10	11.38	13.47	6
	20	20.74	19.8	3.02
New	1	2.6	2.4	3.74
Kp=65	3	4.81	5.71	5.16
Ki=0	5	6.79	6.41	4.01
Kd=30	10	11.53	10.9	2.27
	20	20.89	19.8	1.15

Table 4.4: Old controller vs. New controller

The experiment was performed with different values for the desired depth ($Z_d = [1, 3, 5, 10, 20]$), as it can be seen in the figure 4.7 the rise time is lightly bigger (max 0.3 seconds bigger) in the new controller, but as figures 4.8 and 4.9 show, both overshot and settling time are considerable smaller in the new controller rather than in the old one.

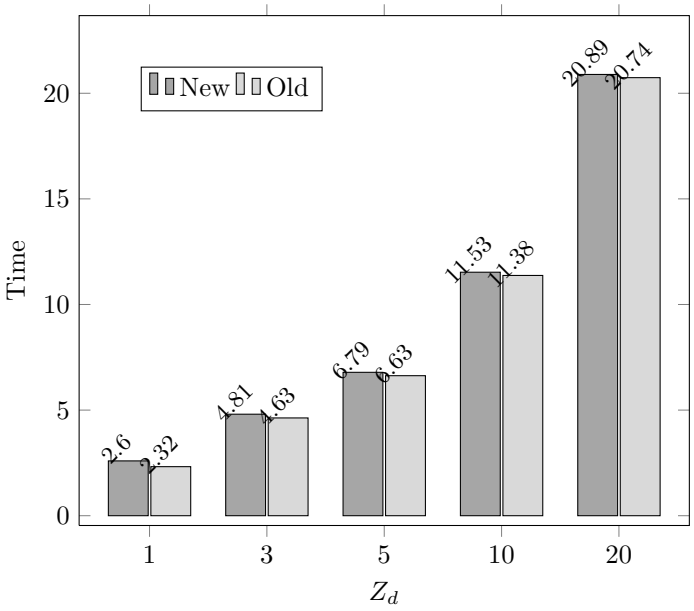


Figure 4.7: New vs. Old, Rise time

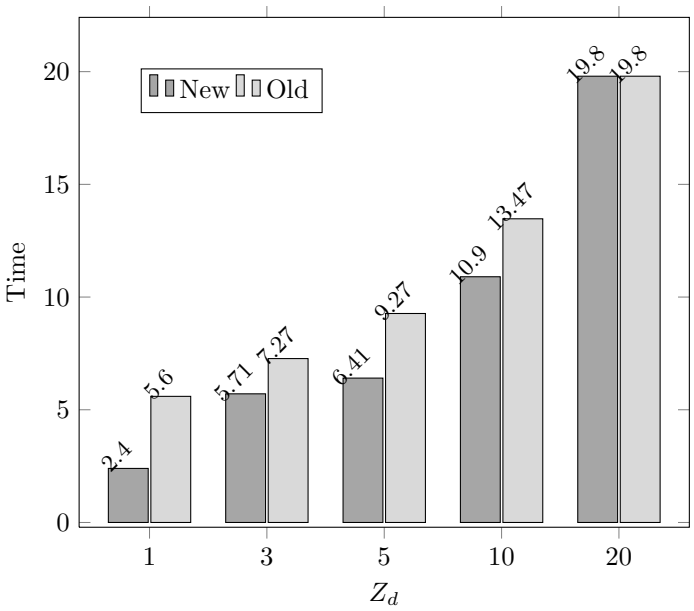


Figure 4.8: New vs. Old, Settling time

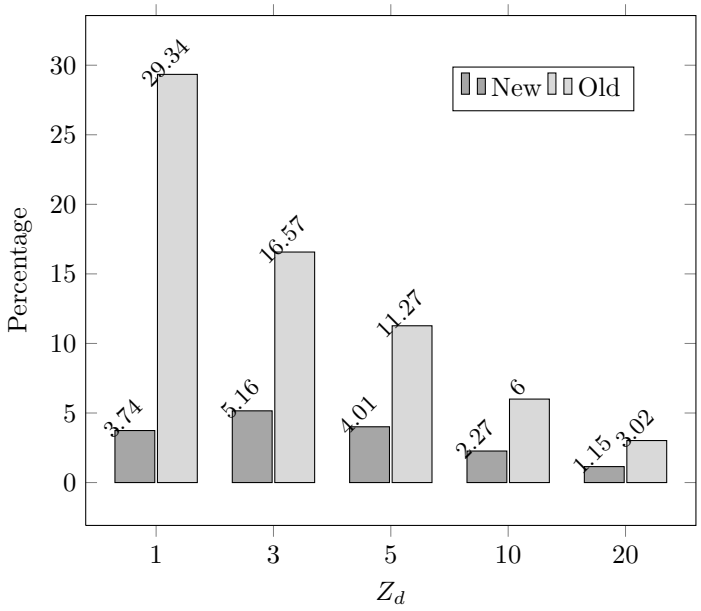


Figure 4.9: New vs. Old, Overshot

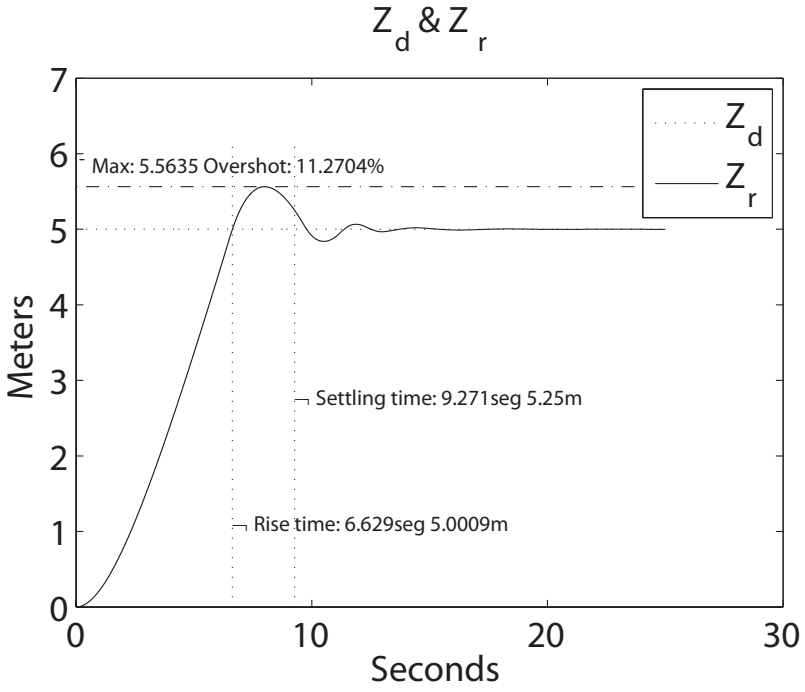


Figure 4.10: Old controller, $K_p = 100\, K_i = 0\, K_d = 10$

5 Orientation PID controller

The gains of this orientation PID controller were also calculated using an heuristic method, because of the non-linearities of the system.

The main difference between this PID controller and the one on chapter 4 is that, in this case the main purpose of the controller is not to reach the desired position but to keep it. As it was explained on section 3.6.4 to set a new desired orientation the controller must be disabled, then the ROV must be turned to the desired orientation and then the controller must be enabled again, therefore at the beginning of the controller operation, the desired position is already reached so the idea is to continue in that orientation.

The initial considerations were the following:

- For tuning purposes, the desired angle was 5°
- The initial ψ angle is 5°
- The output signal of the controller was limited to ±5v, because that is the range of the thruster's control voltage
- The depth controller was disabled during the experimentations
- Thrusters 1 and 2 work with a control voltage of 2.5v
- The thrusters push the ROV to a negative turn of ψ, then an *Undershot* parameter was also taken in consideration.
- The training platform was used to calculate the response of the PID controller
- The sampling time was from 0 to 25 seconds
- The sampling frequency was 1kHz (0.001 seconds)
- Here the settling time is defined as the moment when the response is contained inside an error band of ±5% of the desired value.
- The mean between the last 100 values of the response and the desired value was taken as *Stationary Error*
- The *No Error* parameter was taken when the response during the rest of the sample is contained inside an error band of ±0.005° of the desired value.
- The fastest stabilization (*No Error* parameter) is desired

5.1 Selection K_p - Orientation controller

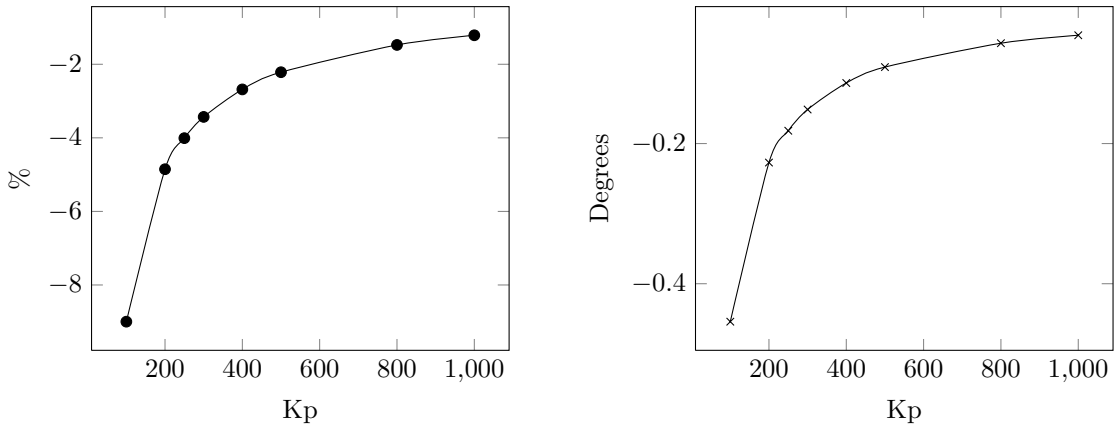
First the gain K_p was calculated, on table 5.1 are the results obtained by the system.

The values of *settling time, overshot and no-error* were not included, because settling time and overshot were 0 for all the cases, and the *no-error* parameter was never reached, that means that the stationary error was never less than 0.005, as it can be seen on figure 5.1b.

In general the value of K_i is in charge of reducing the stationary error, then incrementing the value K_p wasn't going to help to reduce this value.

Kp	Undershot	Stationary Error
100	−8.9978	−0.4542
200	−4.8532	−0.2271
250	−4.0088	−0.1817
300	−3.4311	−0.1514
400	−2.6836	−0.1135
500	−2.216	−0.0908
800	−1.4745	−0.0568
1,000	−1.212	−0.0454

Table 5.1: Responses for the P controller



(a) K_p Undershot

(b) K_p Stationary error

Figure 5.1: K_p Undershot & Stationary error

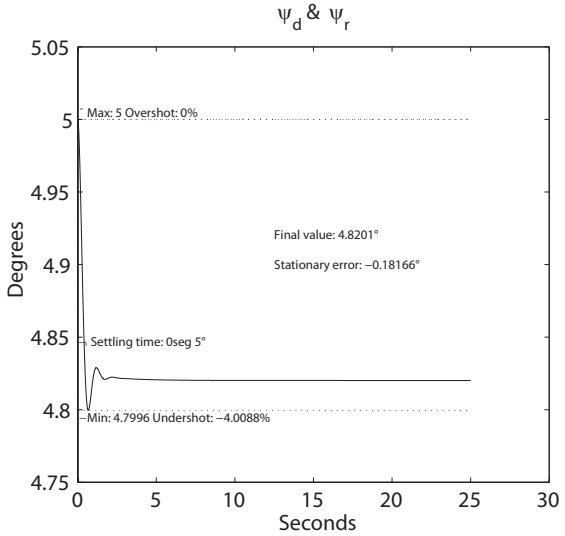


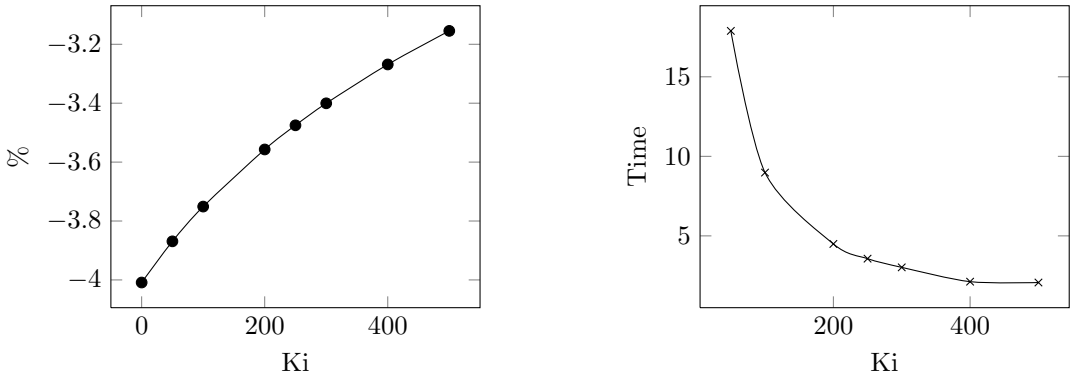
Figure 5.2: $K_p = 250$

The selected K_p value was $K_p = 250$ because with values higher than 200, the *settling time* is 0, and there is an improvement of .8% in *undershot* between $K_p = 200$ and $K_p = 250$, if we increment the value of K_p this improvement is not as big as with these 2 values.

5.2 Selection K_i - Orientation controller

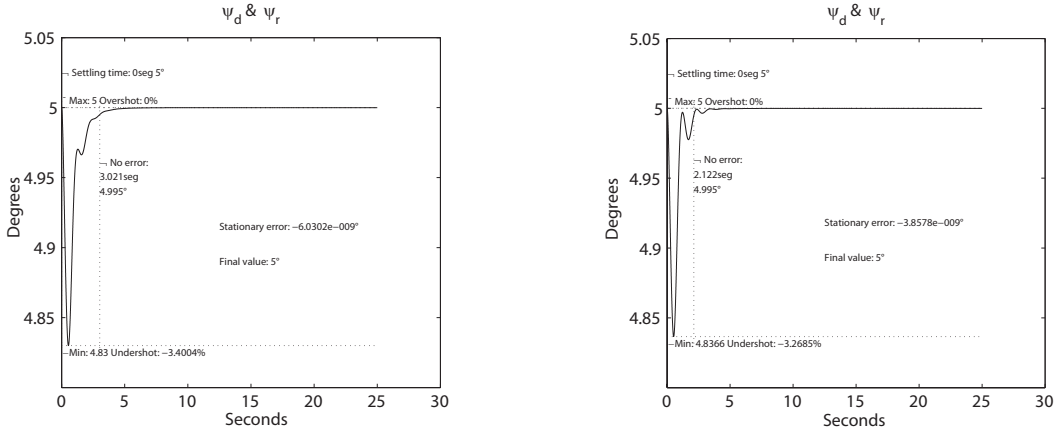
Ki	Undershot	Overshot	Stationary Error	No error (Sec)	Last
0	-4.0088	0	-0.1817		4.8201
50	-3.869	0	$-1.18 \cdot 10^{-3}$	17.886	4.9988
100	-3.751	0	$-6.21 \cdot 10^{-6}$	8.979	5
200	-3.5571	0	$-1.38 \cdot 10^{-8}$	4.493	5
250	-3.4751	0	$-8.39 \cdot 10^{-9}$	3.568	5
300	-3.4004	0	$-6.03 \cdot 10^{-9}$	3.021	5
400	-3.2685	0	$-3.86 \cdot 10^{-9}$	2.122	5
500	-3.1545	0.38	$-2.84 \cdot 10^{-9}$	2.068	5

Table 5.2: Responses for the PI controller



(a) K_i Undershot (b) K_i No-Error

Figure 5.3: K_i Undershot & No-Error



(a) $K_i = 300$ (b) $K_i = 400$

Figure 5.4: K_i responses

With the introduction of K_i to the system, the steady state error has practically erased (Table 5.2) as expected, the *undershot* was also reduced and as K_i was incremented. The selected value of K_i is $K_i = 300$, because there is an improvement of 0.5 seconds on the *No-error* parameter with respect to $K_i = 200$, and even though there is also and improvement with $K_i = 400$ the signal response to the system is more unstable with this value (Fig. 5.4b).

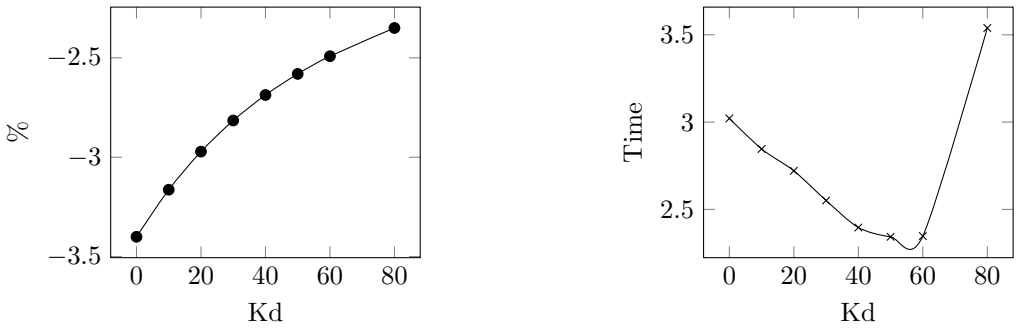
5.3 Selection K_d - Orientation controller

Kd	Undershot	Overshot	Stationary Error	No error (Sec)	Last
0	-3.4004	0	$-6.03 \cdot 10^{-9}$	3.021	5
10	-3.1636	0	$-5.93 \cdot 10^{-9}$	2.846	5
20	-2.9718	0	$-5.83 \cdot 10^{-9}$	2.721	5
30	-2.8153	0	$-5.74 \cdot 10^{-9}$	2.551	5
40	-2.687	0	$-5.65 \cdot 10^{-9}$	2.396	5
50	-2.5809	0.0139	$-5.56 \cdot 10^{-9}$	2.343	5
60	-2.4918	0.0483	$-5.48 \cdot 10^{-9}$	2.347	5
80	-2.3498	0.1154	$-5.31 \cdot 10^{-9}$	3.539	5

Table 5.3: Responses for the PID controller

Finally after adding the K_d value to the controller the orientation controller is finished. The selected value of K_d is $K_d = 50$ because with this value the smallest *No-error* parameter is reached, and although there is a little overshot (0.01391%) it does not improve with any superior value of K_d. With this PID controller, it takes the system only 2.343 seconds to establish.

There cannot be a comparison with the old controller because they were designed for different purposes, in this controller the main task is to maintain the desired value, instead of reaching it.



(a) K_d Undershot

(b) K_d No-Error

Figure 5.5: K_d Undershot & No-Error

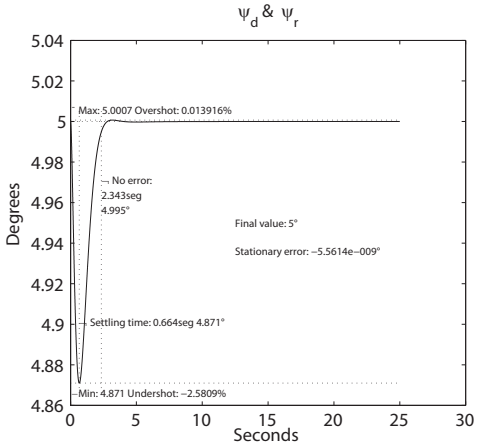


Figure 5.6: $K_p = 250$, $K_i = 300$, $K_d = 50$

6 Conclusions & Results

6.1 Final results

The specific objectives of the project were accomplished, the final product is the first version of a training platform capable of simulating in real time the dynamics of the KAXAN ROV from CIDESI.

In test mode a step size of 0.01 seconds proved to give similar results to the ROV's model in Matlab, in fact the results obtained by the C#'s model present a better tendency in the data rather than the Matlab's model (Fig. 3.56). Normal mode generated very similar results to Matlab's model as well.

The depth controller showed a really good improvement against the old depth controller, both settling time and overshot were significantly smaller for different depths than the old controller.

The orientation controller was designed to prevent the ROV of moving away from a given position, the idea of this controller is not to reach the desired position, but to keep it. With a constant force produced by a voltage control of 2.5v on the thrusters 1 and 2, the response time of the controller to keep the desired position after the force's impulse is about 2.3 seconds.

A lot of the graphical work of the system is done by the graphic card, this keeps the CPU's resources free even though the refresh rate of the graphics is about 60hz.

The simulation of the marine current was in fact a success, if no external forces are applied to the ROV and the current is working then the ROV is driven away in the direction of the current. It is a good exercise for the operator to try to maintain an static position when the marine current is active.

6.2 Conclusions

Using numerical methods on C# is possible to simulate the dynamic behaviour of the KAXAN ROV in real time operation.

For the PID tuning, the linearization methods are not a good option because the system is highly non-linear. The best option in these cases is using the heuristic method.

Given the operational design of this training platform it's better to design an orientation PID controller focus on maintaining the desired position, rather than reaching the desired variable.

Working directly with the graphic card reduces the CPU's resources usage and allows the implementation of better graphics without losing speed in the simulation.

6.3 Future work

It's important to remember that the objective of this project is to provide a solid base for the creation of a complete training platform, therefore here are some key points to improve in further work:

- Adding collision's algorithms to the system, to prevent the ROV to pass through the buildings and structures
- Adding the possibility to change the current during the simulation execution, and even the possibility to add more than one current

- Improving the graphics is possible if a texture loader capability is added to the *Object loader class*
- The water surface can have a better look, if *Pixel Shaders* are programmed
- Adding the capacity to use different interfaces to control the ROV
- Other students at CIDESI are developing intelligent controllers for the ROV, these controllers can be added easily to the current simulator, this can improve greatly the work and behaviour of the controllers
- Adding the possibility to save and load different configurations for different tasks
- In future versions of the simulator is recommended to use an specialized computer for virtual environments or a server for group access

A Flow Charts

A.1 Mass-Spring-Damper on C#

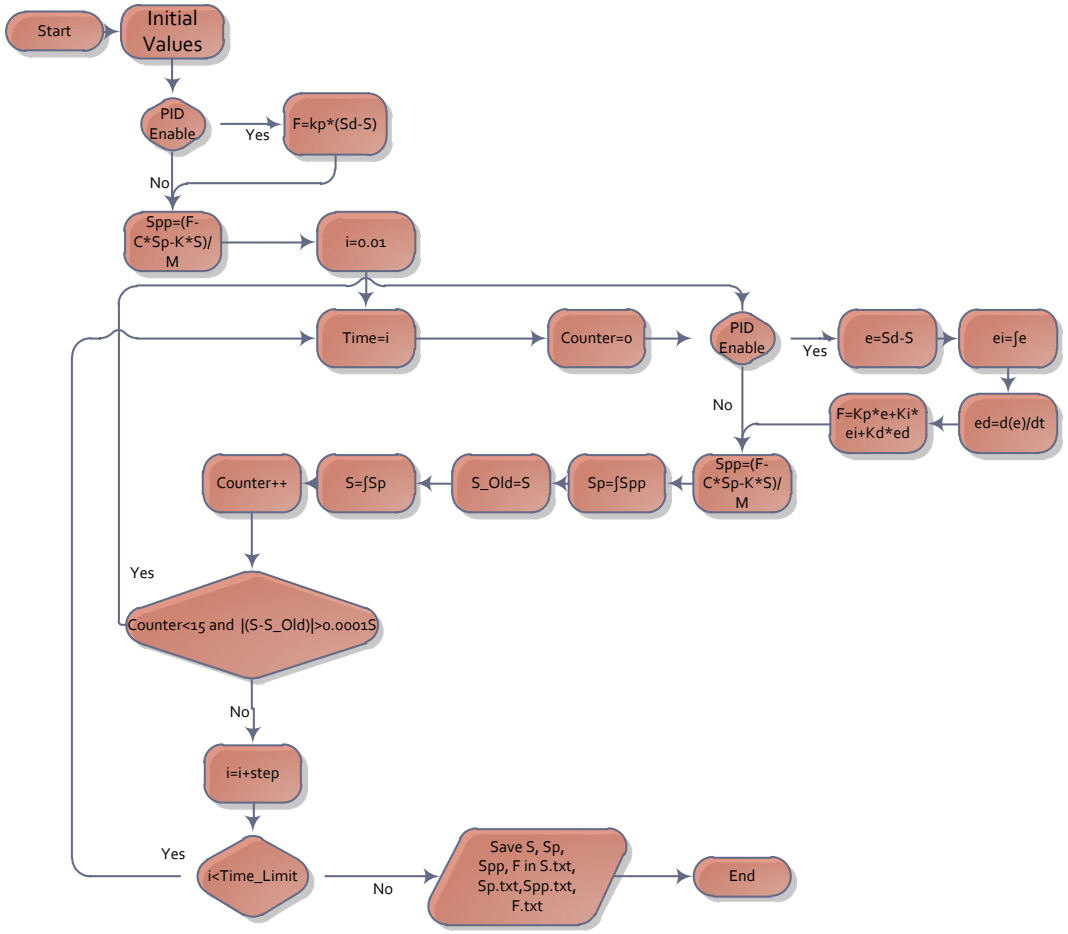


Figure A.1: Flow chart of Mass Spring Damper program on C#

A.2 Simulator on C#

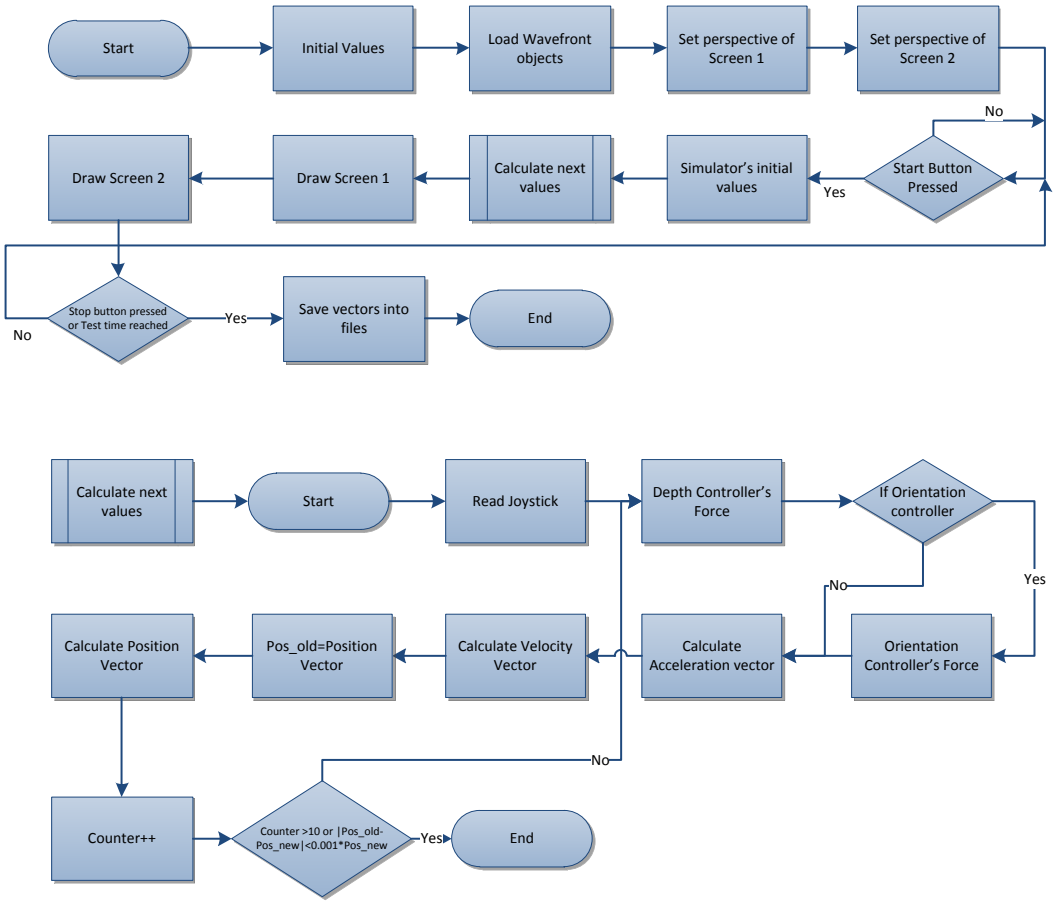
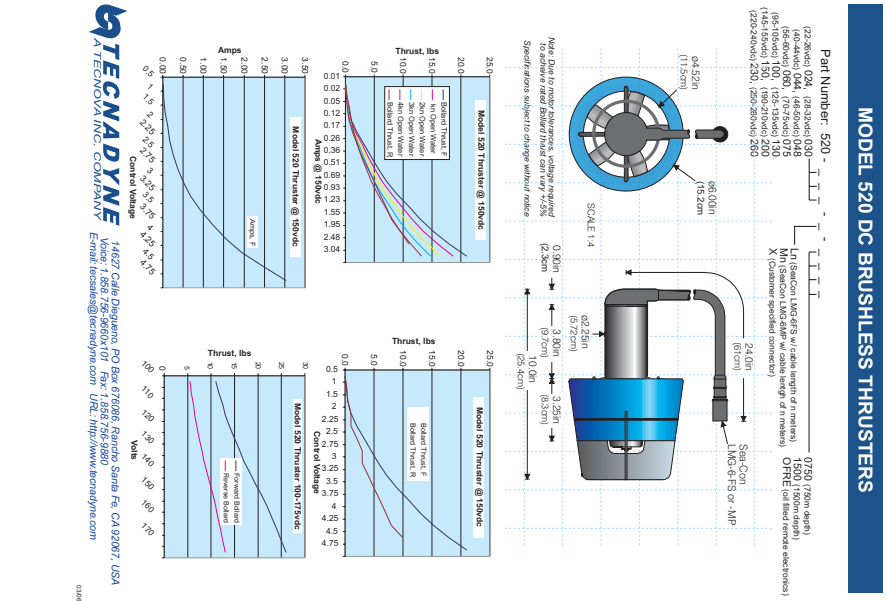


Figure A.2: Flow chart of the Trainings Platform

B Thrusters datasheets

B.1 Model 520



B.2 Model 540

MODEL 540
DC BRUSHLESS THRUSTERS

The Model 540 is the tunnel thruster version of the Model 520 & 560 ducted thrusters, of which Tecnadyne has delivered over 1,400 units to customers worldwide. The Model 540 fits within a tunnel duct of 15.2cm inside diameter. Since it produces the same Bollard thrust in both forward and reverse directions, the Model 540 is ideally suited as a lateral or vertical thruster on mid-size AUV's and on hydrodynamic ROV's.

The precision stainless steel propeller of the Model 540 is magnetically coupled using a design perfected by Tecnadyne. With this design, a magnet array in the hub of the propeller is driven by a matching magnet array attached to the drive motor. By eliminating the rotating drive shaft and shaft seals that always seem to leak over time, the Model 540 achieves extremely high reliability. Additionally, the magnetic coupling will ratchet if overloaded, preventing damage caused by objects jammed in the propeller. And since the water lubricated propeller bearings are external to the pressure housing, they can be easily replaced in several minutes.

Employing a high RPM, low inertia DC brushless motor, coupled to a 6/1 ratio planetary gearset, the Model 540 delivers maximum reliability, high efficiency and high power in an extremely compact , lightweight and easy to maintain package.

For depths to 1,000 meters and optionally to 2,500 meters, the power and control electronics are housed within the hard anodized aluminum motor casing, greatly simplifying the installation and electrical interface. For full ocean depth rating, the electronics are installed in a remote, one atmosphere housing (either the customers housing or one supplied by Tecnadyne) and the thruster is oil filled for pressure tolerance.

The Model 540 is available for operation at voltages from 48vdc to 330vdc (150vdc standard) supplied by a well filtered battery bank, rectified and filtered AC or a DC power supply. In addition to the main power, the thruster requires an isolated +-5v analog speed and direction control signal and 12vdc instrumentation power. Alternately, a full servo RS232 or RS485 input controller is available which, due to its size, must be installed in a remote, one atmosphere housing. Please refer to the Tecnadyne website for detailed installation and interface instructions.

The standard depth rating of the Model 540 is 850 meters. 2,500 meters and full ocean depth with remote one atmosphere electronics is an available option. Remote electronics options include the extremely compact Tecnadyne controller module or larger, full servo brushless or sensorless units. For applications requiring extremely low noise, Tecnadyne offers an optional remote linear drive. Customer specified subsea connectors and cables, stainless steel or titanium housings and custom mountings are also available.

MODEL 540 SPECIFICATIONS

Bollard Output	***Input***	***Weight***
21lbf (9.5kg) forward	*150vdc, 3A power*	*4.2lb (1.9kg) in air*
21lbf (9.5kg) reverse	*(450 watts at alternate voltages)*	*3.0lb (1.4kg) in water*
	12vdc, 200mA isolated instrumentation power	
	+/-5v analog speed command	

Depth Rating
2,800ft (850m) standard,
8,000ft (2,500m) and full ocean depth (oil filled) optional

(1,000m & greater depths subject to US Govt. export approval)

TECNADYNE
A TECNOVA INC. COMPANY

MODEL 540 DC BRUSHLESS THRUSTERS

TECNADYNE
A TECNOVA INC. COMPANY

14627 Calle Plegueno, PO Box 670036, Rancho Santa Fe, CA 92067, USA
Voice: 1.858.756-9760x101 Fax: 1.858.756-9980
E-mail: tecsales@tecnadyne.com URL: http://www.tecnadyne.com

03/05

C Joystick programming

C.1 Joystick structure code

```
struct joy_config
{
        public int xp, xn, yp, yn, zp, zn, x2p, x2n, y2p, y2n;
        public int axis_x, axis_y, axis_z, axis_x2, axis_y2;
}
joy_config joy_param = new joy_config();
//Values read form the Joystick for different positions.
joy_param.xn = -32767;
joy_param.xp = 32767;
joy_param.x2n = -32767;
joy_param.x2p = 32767;
joy_param.yn = 32767;
joy_param.yp = -32767;
joy_param.y2n = 32767;
joy_param.y2p = -32767;
joy_param.zn = 32639;
joy_param.zp = -32639;
joy_param.axis_x = 0;
joy_param.axis_y = 1;
joy_param.axis_z = 2;
joy_param.axis_x2 = 4;
joy_param.axis_y2 = 3;
```

C.2 Joystick reading code

```
Sdl.SDL_JoystickUpdate();
int joy_y = Sdl.SDL_JoystickGetAxis(Joystick, joy_param.axis_y);
joy_y++;
//The control voltage of thrusters 1 and 2
//is managed by Joystick's y-axis
//Maximum and minimum of +/-5 volts
if (joy_y > 0)
{
        f1 = (double)joy_y / (double)joy_param.yp;
        f1 = 5 * f1;
        f2 = f1;
}
else
{
        f1 = -5 * (double)joy_y / (double)joy_param.yn;
        f2 = f1;
}
//The control voltage of thruster 3
//is managed by Joystick's x-axis
//Maximum and minimum of +/-5 volts
int joy_x = Sdl.SDL_JoystickGetAxis(Joystick, joy_param.axis_x);
```

```
joy_x++;
if (joy_x > 0)
        f3 = 5 * (double)joy_x / (double)joy_param.xp;
else
f3 = -5 * (double)joy_x / (double)joy_param.xn;

//The desired depth
//is managed by Joystick's z-axis
//the increment is constant of 0.6 of the sample time
int joy_z = Sdl.SDL_JoystickGetAxis(Joystick, joy_param.axis_z);
joy_z++;
if (joy_z > 0)
        zd +=  d_time*(.6 * (double)joy_z / (double)joy_param.zp);
else
        zd +=  d_time*(-.6 * (double)joy_z / (double)joy_param.zn);
if (zd < 0) zd = 0;

//Eye
//The rotation of the camera's position
//is given by the movement of x2-axis
//The increment is a constant of 90*sample time
int joy_x2 = Sdl.SDL_JoystickGetAxis(Joystick, joy_param.axis_x2);
joy_x2++;
if (joy_x2 > 0)
{
        eye_angle+= (float)(90*d_time * joy_x2) / joy_param.x2p;
}
else
{
        eye_angle-= (float)(90*d_time * joy_x2) / joy_param.x2n;
}
//Relative position of camera on x coordinate
eye_x = -eye_r * (float)(Math.Cos((double)eye_angle*Math.PI/180));
//Relative position of camera on y coordinate
eye_y = eye_r * (float)(Math.Sin((double)eye_angle * Math.PI / 180));

//The Z position of the camera's position
//is given by the movement of y2-axis
//The increment is a constant of 3*sample time
int joy_y2 = Sdl.SDL_JoystickGetAxis(Joystick, joy_param.axis_y2);
joy_y2++;
if (joy_y2 > 0)
        eye_z += (float)((3*d_time * joy_y2) / joy_param.y2p);
else
        eye_z -= (float)((3*d_time * joy_y2) / joy_param.y2n);

//X buton on Playstation Joystick
//Returns the camera to its initial relative position
if (Sdl.SDL_JoystickGetButton(Joystick, 0) == 1)
{
        eye_x = -3;
        eye_y = 3;
        eye_z = -3;
```

```
        eye_angle = 45;
}

//O buton on Playstation Joystick
//Turns on and off the psi controller
//only if the button wasn't press before
if (Sdl.SDL_JoystickGetButton(Joystick, 1) == 1 && !o_button_press)
{
        psi_control = !psi_control;
        o_button_press = true;
}
else if (Sdl.SDL_JoystickGetButton(Joystick, 1) == 0 && o_button_press)
        o_button_press = false;
```

D Matrix Operations Class

D.1 Addition-Subtraction

```
public static double[,] sum_m(double[,] m1, double[,] m2, bool sum)
{
        double[,] m3 = new double[m1.GetLength(0), m1.GetLength(1)];
        //Checks m1 & m2 are of the same size
        if ((m1.GetLength(0) == m2.GetLength(0)) && (m1.GetLength(1) == m2.GetLength(1)))
        {
        for (int i = 0; i < m1.GetLength(0); i++)
        {
             for (int j = 0; j < m1.GetLength(1); j++)
             {
                                //Is sum is true, then the function realized an addition
                                //if not then a subtraction m1-m2
                if (sum)
                {
                    m3[i, j] = m1[i, j] + m2[i, j];
                }
                else
                                {
                                        m3[i, j] = m1[i, j] - m2[i, j];
                                }
                        }
                }
        }
        //If m1 and m2 are not of the same size the function return a 0
        else
        {
                m3 = new double[1, 1] { { 0 } };
        }
        return m3;
}
```

D.2 Multiplication of matrices

```
public static double [,] mul_m(double [,] m1, double [,] m2)
        {
        double [,] m3 =new double[m1.GetLength(0),m2.GetLength(1)];
        //For m1 of dimension mxn it can only multiplied witha a matrix
        // m2 of dimension nxl
        //the result is m3 of dimension mxl
        if (m1.GetLength(1) == m2.GetLength(0))
            {
                for (int i = 0; i < m1.GetLength(0); i++)
                {
                    for (int l = 0; l < m2.GetLength(1); l++)
                    {
                        double s = 0;
                        for (int j = 0; j < m1.GetLength(1); j++)
                        {
                            s += m1[i, j] * m2[j, l];
                        }

                        m3[i, l] = s;
                    }

                }
            }
            //if m2 is not nxl, the result is matrix
            //m3 of mxl full of zeros with a diagonal equal to 1
            else
            {
                                for (int i = 0; i < m1.GetLength(0); i++)
            {
                    for (int j = 0; j < m3.GetLength(1); j++)
                    {
                        if (i == j)
                        {
                            m3[i, j] = 1;
                        }
                        else
                        {
                            m3[i, j] = 0;
                        }
                    }
                }
            }
        return m3;
}
```

D.3 Matrix $\times k$

```
public static double[,] mul_k( double[,] m1,double k )
{
```

```
        for (int i = 0; i < m1.GetLength(0); i++)
        {
                for (int j = 0; j < m1.GetLength(1); j++)
                {
                 m1[i, j] *= k;
                }
        }
        return m1;
}
```

D.4 Inverse using *Gauss-Jordan Elimination*

```
public static double[,] inv_m(double[,] m1)
{
        double[,] m4 = new double[m1.GetLength(0), m1.GetLength(0) * 2];
        bool singular = false;
        //Check if the matrix is square
        if (m1.GetLength(0) == m1.GetLength(1))
        {
                //Add the Gauss matrix to m4
                for (int i = 0; i < m4.GetLength(0); i++)
                {
                        for (int j = 0; j < m4.GetLength(1); j++)
                        {
                                if (j < m4.GetLength(0))
                                {
                                        m4[i, j] = m1[i, j];
                                }
                                else if (i == (j - m4.GetLength(0)))
                                {
                                        m4[i, j] = 1;
                                }
                                else
                                {
                                        m4[i, j] = 0;
                                }
                        }
                }
                for (int l = 0; l < m4.GetLength(0); l++)
                {
                        //Check if the element [l,l] is not 0
                        if (m4[l, l] == 0)
                        {
                        singular = true;
                        //If there is a row below of row l, that does not contains
                        //a 0 in position [l,l] these two rows
                        //are shifted
                        for (int i = l + 1; i < m4.GetLength(0); i++)
                        {
                                if (m4[i, l] != 0)
                                {
```

```
                                        double[] shift = new double[m4.GetLength(1)];
                                        for (int j = l; j < m4.GetLength(1); j++)
                                        {
                                                shift[j] = m4[l, j];
                                                m4[l, j] = m4[i, j];
                                                m4[i, j] = shift[j];
                                        }
                                                l--;
                                                singular = false;
                                        }
                                }
                        }
                        else
                        {
                        //The Gauss-Jordan elimination begins
                        for (int i = 0; i < m4.GetLength(0); i++)
                        {
                                double factor;
                                if (i == l)
                                {
                                        factor = m4[l, l];
                                }
                                else
                                {
                                        factor = m4[i, l] / m4[l, l];
                                }
                                for (int j = 0; j < m4.GetLength(1); j++)
                                {
                                if (i == l)
                                {
                                        m4[i, j] = m4[i, j] / factor;
                                }
                                else
                                {
                                        m4[i, j] = m4[i, j] - (factor * m4[l, j]);
                                }
                                }
                        }
                        }
                }
                }
        else
        {
                singular = true;
        }
        //If it is a singular matrix the return matrix is m1
        if (singular)
        {
                return m1;
        }
        else
        {
                //The second half of matrix m4 is stored in matrix m5
```

```
                //m5 is now the inverse matrix
                double[,] m5 = new double[m4.GetLength(0), m4.GetLength(0)];
                for (int i = 0; i < m4.GetLength(0); i++)
                {
                for (int j = m4.GetLength(0); j < m4.GetLength(1); j++)
                {
                        m5[i, j - m4.GetLength(0)] = m4[i, j];
                }
                }
        return m5;
        }
}
```

D.5 Transpose

```
public static double[,] trans(double[,] m1)
{
        int m = m1.GetLength(0);
        int n = m1.GetLength(1);
        double[,] m2= new double[n, m];
        for (int i = 0; i < m; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        m2[j, i] = m1[i, j];
                }
        }
        return m2;
}
```

D.6 Integration

```
public static double[,] int_m(ref ArrayList func, ref ArrayList func1, double time)
{
        double[,] integrate;
        double[,] f0, f1;
        int count = func.Count;
        if (count != 1)
        {
                f0 = (double[,])func[count - 2];
                f1 = (double[,])func[count - 1];
                //Trapezoidal Method (f1+f0)*dt/2
                integrate = mul_k(sum_m(f1, f0,true), .5*time);
                //Add the value of the integral to the old values,
                //this is equal to the integration from 0 to the current value
                integrate = sum_m(integrate, (double[,])func1[count - 2],true);
        }
        else
        {
                //If there is only one value on the array,
```

```
                //the result of the integration is f(0)*dt/2
                return mul_k((double[,])func[0],time / 2);
        }
        return integrate;
}
```

D.7 Matrix printing

```
public static string print_m(double[,] m1,string format)
{
        string str = "";
        for (int i = 0; i < m1.GetLength(0); i++)
        {
                for (int j = 0; j < m1.GetLength(1); j++)
                {
                        str += m1[i,j].ToString(format);
                        // \t adds a Tab space to the string
                        str += "\t";
                }
                // \n is the command to add a new line to the string
                str += "\n";
        }
        return str;
}
```

D.8 Matrix printing to a file

```
public static void print_2_file(double[,] m1, string name)
{
        string str = "[";
        for (int i = 0; i < m1.GetLength(0); i++)
        {
                for (int j = 0; j < m1.GetLength(1); j++)
                {
                        str += m1[i, j].ToString();
                        if (j < m1.GetLength(1) - 1)
                        {
                                str += ",";
                        }
                }
                if (i < m1.GetLength(0) - 1)
                {
                        str += ";";
                }
        }
        str += "]";
        System.IO.StreamWriter objWriter = new System.IO.StreamWriter(name);
        objWriter.WriteLine(str);
        objWriter.Close();
}
```

D.9 Reading matrix from file

```
public static double[,] read_f_file(string name)
{
        double[,] m1;
        if (System.IO.File.Exists(name) == true)
        {
                System.IO.StreamReader objReader = new System.IO.StreamReader(name);
                string str = objReader.ReadLine();
                str=str.Trim('[',']');
                string[] row = str.Split(';');
                string[] element = new string[row.Length];
                element = row[0].Split(',');
                m1 = new double[row.Length, element.Length];
                for (int i = 0; i < row.Length; i++)
                {
                        element = row[i].Split(',');
                        for (int j = 0; j < element.Length; j++)
                        {
                                m1[i, j] = double.Parse(element[j]);
                        }
                }
        }
        else
                m1 = new double [1,1] {{1}};
        return m1;
}
```

E Dynamic model's matrices

E.1 Matrix C_{RB}

Coriolis and centripetal matrix from inertia matrix

```
public static double[,] fill_matrix_c(ref double[,] vel, ref double m,_
                                      ref double Ixx, ref double Iyy, ref double Izz)
{
        //double[,] vel = matrix_op.sum_m(vel_rov, vel_c, false);
        double uu = vel[0, 0];
        double vv = vel[1, 0];
        double ww = vel[2, 0];
        double pp = vel[3, 0];
        double qq = vel[4, 0];
        double rr = vel[5, 0];
        double [,] m_c = new double[6,6] {{0, 0, 0, 0, m*ww, -m*vv},
                                      {0, 0, 0, -m*ww, 0, m*uu},
                                      {0, 0, 0, m*vv, -m*uu, 0},
                                      {0, m*ww, -m*vv, 0, -Izz*rr, -Iyy*qq},
                                      {-m*ww, 0, m*uu, -Izz*rr, 0, -Ixx*pp},
                                      {m*vv, -m*uu, 0, -Iyy*qq, -Ixx*pp, 0}};
        return m_c;
}
```

E.2 Matrix C_A

Coriolis and centripetal matrix from added mass matrix

```
public static double[,] fill_matrix_ca(ref double[,] vel_rov,ref double[,] vel_c,_
                        ref double Xup,ref double Yvp, ref double Zwp,_
                        ref double Kpp,ref double Mqp,ref double Nrp)
{        double[,] vel = matrix_op.sum_m(vel_rov, vel_c, false);
        double uu = vel[0, 0];
        double vv = vel[1, 0];
        double ww = vel[2, 0];
        double pp = vel[3, 0];
        double qq = vel[4, 0];
        double rr = vel[5, 0];
    double [,] m_ca = new double[6,6] {{0, 0, 0, 0, -Zwp*ww, Yvp*vv},
                                       {0, 0, 0, Zwp*ww, 0, -Xup*uu},
                                       {0, 0, 0, -Yvp*vv, Xup*uu, 0},
                                       {0, -Zwp*ww, Yvp*vv, 0, -Nrp*rr, Mqp*qq},
                                       {Zwp*ww, 0, -Xup*uu, Nrp*rr, 0, -Kpp*pp},
                                       {-Yvp*vv, Xup*uu, 0, -Mqp*qq, Kpp*pp, 0}};
        return m_ca;
}
```

E.3 Matrix D

Hydrodynamic damping

```
public static double[,] fill_matrix_d(ref double[,] vel_rov,ref double[,] vel_c,_
                                      ref double Xu, ref double Xuu, ref double Yv,_
                                      ref double Yvv, ref double Zw, ref double Zww,
                                      ref double Kp, ref double Kpp2, ref double Mq,
                                      ref double Mqq, ref double Nr, ref double Nrr)
{
        double[,] vel = matrix_op.sum_m(vel_rov, vel_c, false);
        double uu = vel[0, 0];
        double vv = vel[1, 0];
        double ww = vel[2, 0];
        double pp = vel[3, 0];
        double qq = vel[4, 0];
        double rr = vel[5, 0];
    double[,] m_d = new double[6, 6] {{-(Xu+Xuu*Math.Abs(uu)), 0, 0, 0, 0, 0},
                                      {0, -(Yv+Yvv*Math.Abs(vv)), 0, 0, 0, 0},
                                      {0, 0, -(Zw+Zww*Math.Abs(ww)), 0, 0, 0},
                                      {0, 0, 0, -(Kp+Kpp2*Math.Abs(pp)), 0, 0},
                                      {0, 0, 0, 0, -(Mq+Mqq*Math.Abs(qq)), 0},
                                      {0, 0, 0, 0, 0, -(Nr+Nrr*Math.Abs(rr))}};
        return m_d;
}
```

E.4 Matrix g

Restoring forces & Moments

```
public static double[,] fill_matrix_g(ref double[,] pos, ref double zB,_
                                                  ref double W, ref double B)
{
        double phi = pos[3,0];
        double theta = pos[4,0];
        double psi = pos[5,0];
        //This function can be used only when W=B
    double[,] m_g = new double[6, 1] {{ 0 },{ 0 },{ 0 },
                                      { -zB * W * Math.Cos(theta) * Math.Sin(phi) },
                                      { -zB * W * Math.Sin(theta) },{ 0 } };
        return m_g;
}
```

E.5 Matrix J_1

Rotation matrix for position coordinates (*body-fixed* frame to *earth-fixed* frame)

```
public static double[,] fill_matrix_j1(ref double[,] pos)
{
```

```
        double phi = pos[3, 0];
        double theta = pos[4, 0];
        double psi = pos[5, 0];
        double[,] m_j1 = new double[3, 3]_
    {{ Math.Cos(psi)*Math.Cos(theta),_
     (-Math.Sin(psi)*Math.Cos(phi))+(Math.Cos(psi)*Math.Sin(theta)*Math.Sin(phi)),_
      (Math.Sin(psi)*Math.Sin(phi))+(Math.Cos(psi)*Math.Sin(theta)*Math.Cos(phi)) },_
    {  Math.Sin(psi)*Math.Cos(theta),_
      (Math.Cos(psi)*Math.Cos(phi))+(Math.Sin(psi)*Math.Sin(theta)*Math.Sin(phi)),_
     (-Math.Cos(psi)*Math.Sin(phi))+(Math.Sin(psi)*Math.Sin(theta)*Math.Cos(phi)) },_
    {-Math.Sin(theta),_
      Math.Cos(theta)*Math.Sin(phi),_
      Math.Cos(theta)*Math.Cos(phi)}};
         return m_j1;
}
```

E.6 Matrix J_2

Rotation matrix for angles (*body-fixed* frame to *earth-fixed* frame)

```
public static double[,] fill_matrix_j2(ref double[,] pos)
{
        double phi = pos[3, 0];
        double theta = pos[4, 0];
        double psi = pos[5, 0];
        double[,] m_j2 = new double[3, 3]_
    { { 1,_
      Math.Sin(phi)*Math.Tan(theta),_
      Math.Cos(phi)*Math.Tan(theta) },_
    { 0,_
      Math.Cos(phi),_
     -Math.Sin(phi) },_
    { 0,_
      Math.Sin(phi)/Math.Cos(theta),_
      Math.Cos(phi)/Math.Cos(theta)} };
        return m_j2;
}
```

E.7 Matrix J

Rotation matrix (*body-fixed* frame to *earth-fixed* frame)

```
public static double[,] fill_matrix_j(ref double[,] pos)
{
        double[,] m_j = new double[6, 6];
        double[,] m_j1, m_j2;
        m_j1 = fill_matrix_j1(ref pos);
        m_j2 = fill_matrix_j2(ref pos);
        for (int i = 0; i < 6; i++)
```

```
        {
                for (int j = 0; j < 6; j++)
                {
                        if (i < 3 && j < 3)
                                m_j[i, j] = m_j1[i,j];
                        else if (i >= 3 && j >= 3)
                                m_j[i, j] = m_j2[i-3, j-3];
                        else
                                m_j[i, j] = 0;
                }
        }
        return m_j;
}
```

E.8 Vector $\dot{\nu}$

Function to calculate the value of equation 3.16

```
private double[,] acc_func(ref double[,] vel_val, ref double[,] pos_val,_
                           ref double[,] tau_val)
{
        double[,] acc_val, help_matrix;
        matrix_c = Model_Matrix.fill_matrix_c(ref vel_val, ref m, ref Ixx,_
                                              ref Iyy, ref Izz);
        matrix_ca = Model_Matrix.fill_matrix_ca(ref vel_val, ref v_current,_
                                              ref Xup, ref Yvp, ref Zwp,_
                                              ref Kpp, ref Mqp, ref Nrp);
        matrix_d = Model_Matrix.fill_matrix_d(ref vel_val, ref v_current, ref Xu,_
                                              ref Xuu, ref Yv, ref Yvv, ref Zw,_
                                              ref Zww, ref Kp, ref Kpp2,ref Mq,_
                                              ref Mqq, ref Nr, ref Nrr);
        matrix_g = Model_Matrix.fill_matrix_g(ref pos_val, ref zB, ref W, ref B);
        //help=t-C_rb(v)*v
        help_matrix = matrix_op.sum_m(tau_val, matrix_op.mul_m(matrix_c,vel_val),_
                                  false);
        //help=help-C_a(vr)*vr
        help_matrix = matrix_op.sum_m(help_matrix, matrix_op.mul_m(matrix_ca,_
                          matrix_op.sum_m(vel_val, v_current, false)), false);
        //help=help-D(vr)*vr
        help_matrix = matrix_op.sum_m(help_matrix, matrix_op.mul_m(matrix_d,_
                          matrix_op.sum_m(vel_val, v_current, false)), false);
        //help=help-g(n)
        help_matrix = matrix_op.sum_m(help_matrix, matrix_g, false);
        //acc_val=vp=(M^-1)*(help)
        acc_val = matrix_op.mul_m(matrix_op.inv_m(matrix_m), help_matrix);
        return acc_val;
}
```

References

[1] Gang Chen, YiLong Jia, and Shang Xiang. Simulation training platform for large tactical communication equipment based on vr. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*, volume 2, pages 121 –125, jan. 2009. doi: 10.1109/ICCET.2009.172.

[2] Robert D. Christ and Robert L. Wernli. *The ROV manual: A User Guide for Observation-Class Remotely Operated Vehicles*. Elsevier, 1 edition, 2001.

[3] OpenTK community. The open toolkit, c# library, October 2006. URL `http://www.opentk.com`.

[4] Thor I. Fossen. *Guidance and control of ocean vehicles*. John Wiley & Sons, Ltd, 1994.

[5] Thor I. Fossen. *Marince Control Systems, Guidance, Navigation, and Control of Ships, Rigs and Underwater Vehicles*. Marine Cibernetics, 1 edition, 2002.

[6] GameDev.net. Nehe productions, 2012. URL `http://www.opentk.com`.

[7] L.G. Garcia-Valdovinos and T. Salgado-Jimenez. On the dynamic positioning control of underwater vehicles subject to ocean currents. In *Electrical Engineering Computing Science and Automatic Control (CCE), 2011 8th International Conference on*, pages 1 –6, oct. 2011. doi: 10.1109/ICEEE.2011.6106590.

[8] ECMA Intenational. C# language specification, June 2006. URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[9] Microsoft. Microsoft flight simulator x, 2006. URL `http://www.microsoft.com/games/flightsimulatorx/default.htm`.

[10] Motioninjoy. URL `http://www.motioninjoy.com`.

[11] Antonio Nieves Hurtado and Federico C Dominguez Sanchez. *Metodos Numericos Aplicados a la Ingenieria*. Grupo Editorial Patria, 3 edition, 2007.

[12] Ing. Guillermo Delgado Ramirez. Control de un robot submarino (rov). Master's thesis, CIDESI.

[13] Martin Reddy. C# language specification, 1994. URL `http://www.martinreddy.net/gfx/3d/OBJ.spec`.

[14] T. Salgado-Jimenez, J.L. Gonzalez-Lopez, J.C. Pedraza-Ortega, L.G. García-Valdovinos, L.F. Martínez-Soto, and P.A. Resendiz-Gonzalez. Design of rovs for the mexican power and oil industries. In *Applied Robotics for the Power Industry (CARPI), 2010 1st International Conference on*, pages 1 –8, oct. 2010. doi: 10.1109/CARPI.2010.5624437.

[15] Yang Yang, Chen Guo, Jian bo Sun, and De wen Yan. A novel simulation system for marine main diesel propulsion remote control. In *Virtual Reality and Visualization (ICVRV), 2011 International Conference on*, pages 57 –62, nov. 2011. doi: 10.1109/ICVRV.2011.9.

Glossary

A

API Application Programming Interface, a program that works like an interface to share data or functions with other programs.

C

CIDESI The Center for Engineering and Industrial Development.

Class In object-oriented programming, a class is a construct that is used to create instances of itself, these instances will share the same behaviour as their class parent.

D

DOF Degree Of Freedom.

F

Form Is an easy way to create a GUI window. A form contains components and controls, which are a high-level representation of standard GUI widgets.

G

GUI Graphical user interface.

L

Library Is a collection of implementations of behaviour, written in terms of a language, that has a well-defined interface by which the behaviour is invoked.

O

OpenGL Open Graphics Library, maintained by OpenGL Architectural Review Board or ARB.

OpenTK Open Toolkit, is a C# library that allows .Net programs to access OpenGL, OpenAL and OpenCL.

P

PID Proportional Integral Derivative Controller.

Plug-in In computing is a set of software components that adds specific abilities to a larger software application.

POV Point of view switch also known as hat switch.

R

RGB Red Green Blue, is a color mode in which red, green, and blue light are added together to reproduce a broad array of colors.

RGBA Red Green Blue Alpha, similar to RGB but here the Alpha property is used in as the opacity channel.

ROV Remotely Operate Underwater Vehicle.

S

SNAME Society of Naval Architects and Marine Engineers, was organized in 1893, to advance the art, science, and practice of naval architecture, shipbuilding and marine engineering.

T

Thread In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system, Multithreading applications processes two or more threads in the same process and sharing resources.