A GRAPHICAL SIMULATOR FOR THE DYNAMICS OF MARINE AUTONOMOUS SURFACE ROBOTS

Relatori: Cecilia Laschi,
Gabriele Ferri,
Alessandro Manzi

Candidato: Jörgen van der Wardt

Anno accademico 2009/2010
To my family.

Acknowledgements go to professor Cecilia Laschi and the whole group of the Research Centre on Sea Technologies and Marine Robotics of Livorno, in particular to Gabriele and Alessandro that helped me during this period of thesis.
Contents

1 MOTIVATIONS, GOALS AND STATE OF THE ART 9
  1.1 Motivations and goals .................................. 9
  1.2 Robotic Simulators .................................. 11
    1.2.1 Open Dynamics Engine (ODE) .................. 13
    1.2.2 Player project (Player/Stage/Gazebo) .......... 16
      1.2.2.1 Player .................................. 16
      1.2.2.2 Stage .................................. 18
      1.2.2.3 Gazebo .................................. 23
    1.2.3 Simbad ......................................... 26
    1.2.4 Webots ......................................... 29
    1.2.5 Microsoft Robotics Developer Studio ........... 31
  1.3 Marine robotics ...................................... 34
    1.3.1 Unmanned Surface Vehicles & Autonomous Surface Vehicles 34
    1.3.2 Winds, waves and sea currents .................. 40

2 MATERIALS AND METHODS 46
  2.1 Robot structure .................................... 46
  2.2 MATLAB, Simulink and the Real Time Workshop .... 52
    2.2.1 MATLAB .................................... 53
    2.2.2 Simulink .................................... 54
    2.2.3 Real Time Workshop and Target Language Compiler 57
  2.3 Qt graphical libraries ............................... 60
    2.3.1 Graphics system ................................ 60
    2.3.2 Coordinate system .............................. 63
ABSTRACT

In the laboratory for marine robotics of the Scuola Superiore Sant’Anna of Pisa located in Livorno (Leghorn), at the time of writing this thesis, a group of researchers and doctoral students are building a marine robot (i.e. a catamaran), that will navigate autonomously in open waters moving in the most economic way in terms of energetic resources spent, by examining sea currents and winds. In the same time the robot will analyse the pollutants in the water and refer the results to a workstation on land via wireless network.

The project of my thesis consisted in creating a graphical simulator that represents virtually such robot in the water and simulates the movements of it depending on the environmental variables represented in this virtual world, that should resemble as much as possible those in the real world, like water currents, winds, obstacles, other ships etc.. In addition the simulator can obtain data directly from the robot and communicate with it via wire in order to test the hardware devices as well.

To realise the graphical interface I decided to use the C++ language together with some libraries for graphical issues called Qt.

The reasons of the choice of this set of libraries are fundamentally three. Firstly they are open source, so anyone can use them freely and see and/or modify the source code; secondly they are written in C++ that is a widely used type of computer language since it is very portable on many different kind of platforms and one of the fastest computer language existing. Lastly it has a great variety of classes that cover all the most important areas of computer graphics and a well documented API.

The physics of the environmental variables is described through a model made with Simulink®, an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems that builds upon MATLAB®.
MATLAB® is a high-level technical computing language and interactive environment for algorithm development, data visualisation, data analysis, and numeric computation.

A tool that is part of MATLAB® is called Real Time Workshop®. With this tool one can convert a Simulink® model into C++ code and this is what I did to let interact the graphical interface with the simulated environmental physics.

For the programming part I have used the open source Integrated Development Environment (IDE) Eclipse. In this case I have made use the extension for C/C++ Developers with the plug-in for the Qt toolbox.

All the programming phases and tests were made on a Linux operating system, more precisely on Ubuntu 10.04 LTS – the Lucid Lynx – edition.

The project, the group of research of marine robotics is participating to, is an international project co-funded by the European Commission, called HydroNet, whose coordinator is the Scuola Superiore Sant'Anna. The HydroNet project is aimed at designing, developing and testing a new technological platform for improving the monitoring of water bodies based on a network of sensors and autonomous, floating and sensorised robots, embedded in an Ambient Intelligence infrastructure [25].

The structure of this document is the following:

- In the first chapter are described the motivations and goals for and of this thesis. Furthermore it is analysed the state of the art of robotics simulators and of the marine robotics field.

- The second chapter describes the materials and methods used. The layout of this chapter is the following:
  
  - firstly the general structure of the marine robot is depicted;
  - secondly MATLAB framework with its tools are investigated;
  - thirdly there is an overview of the Qt graphical libraries;

- The third chapter shows the implementation of the thesis project, more in details:
initially there is an overview of the simulator depicting its characteristics and its various components

then the implementation of the Simulink model is described;

finally there is a look on the source code implementation of the whole project.

- The fourth chapter contains the validation of the program i.e. various test of the developed application were performed and described.

- In the last chapter are discussed the overall results of the application and some aspects of the future developments of the project.
1 MOTIVATIONS, GOALS AND STATE OF THE ART

1.1 Motivations and goals

The technical aspects of the motivations that led to the decision to accomplish a graphical simulator for a marine robot are the following. Basically the lack of such simulators available on the market, even if several robot simulators exist, as we will see in the next section, only few of them are capable to represent a marine environment and a self made marine robot characteristics. Secondly for a “home made” simulator like the one I had to create there is always the possibility of further expansions and customisations that would have been impossible if the software was made by a third party. Thirdly such simulator represents the marine environment and the marine robot’s properties in a much more realistic way as if they were simulated on a general purpose simulator, just because it considers all the various typical specificity of the robot and the real environment where the robot will navigate, that could not be possible to recreate on commercial simulators.

More personal motivations that let me be involved in this project were essentially two. First of all the interest in the field of robotics especially the part of this research area that concern the software design and development for the control of the robot or the interaction between the robot and the surrounding environment via the various robot’s sensors data to be elaborated by the software itself. Secondly my personal predilection to graphics and therefore for each kind of graphical user interface such as for example simulators like the one I had to realise for this project.

The goal of this project was to design and implement a graphical simulator to monitor
and test the dynamical behaviour of a marine robot. The simulator had to be able to offer the user various options and customisations like the features of recreating the marine environment, i.e. the sea currents, winds and sea waves both in an automatic way, resembling the natural parameters of these weather effects, and in a manual way, imposed by the user in order to test the robot in different types of weather even some extreme or that rarely could happen in the real environment.

The simulator had also to offer the user a simple interface to test the navigation and control of the marine robot, for example the possibility of setting some waypoints that would be communicated to the navigation algorithm in order to let the robot move where the user liked to.

Furthermore the graphical application had to show the user all the parameters of the robot and its devices like its speed, heading, the propellers speed of rotation, the rudder's angle etc.

Beside these feature the user and the navigation algorithm had to be able to see and know the state of the robot's sensors like the laser and the sonar, in order to test the obstacle avoidance algorithm knowing the distances of external objects measured by the aforementioned sensors. Such objects could have put on the map by the user in each moment of the simulation.
1.2 Robotic Simulators

In this chapter we will describe what means simulation, what are simulators, with particular emphasis on robotic simulators, and we will consider the state of the art of this field.

Simulation is the process of designing a model of a real or imagined system and conducting experiments with that model. The purpose of simulation experiments is to understand the behaviour of the system or evaluate strategies for the operation of the system. Assumptions are made about this system and mathematical algorithms and relationships are derived to describe these assumptions - this constitutes a “model” that can reveal how the system works [1]. If the system is simple, the model may be represented and solved analytically. However, problems of interest in the real world are usually so complex that a simple mathematical model can not be constructed to represent them. In this case, the behaviour of the system must be estimated with a simulation. Exact representation is seldom possible in a model, this fact forces the model developers to approximations to a degree of fidelity that is acceptable for the purposes of the study.

Simulation allows the analysis of a system’s capabilities, capacities, and behaviours without requiring the construction of or experimentation with the real system. Since it may be extremely expensive to experiment with a real system, a simulation of it can be highly valuable. There are also systems which are too dangerous to carry out for the purpose of analysis, but which can be usefully and safely analysed through simulation. Simulation is used in nearly every engineering, scientific, and technological discipline. Today, the simulation techniques are employed in the design of new systems, the analysis of existing systems, training for all types of activities, and as a form of interactive entertainment. Simulation data lends itself very well to graphic displays. Graphical user interfaces provide easy model construction, operation, data analysis, and data presentation. These tools place a new and more attractive face on simulations. Like all computer applications, modelling and simulation is expanding as a result of improvements in computer hardware and software technologies, this fact leads to more complex, more detailed and more realistic simulations and therefore even more complex
visualisations of them, like for example three-dimensional representation of the model and real-time animations.

Considering our kind of simulation, we will now focus the topic of conversation on the robotic simulators. Let us introduce the fundamentals components of any robotic system and see how and which parts of them can be virtually simulated. Any robot usually has a set of sensors used to perceive the surrounding environment, a set of effectors to manipulate the environment, and a control system that allows the robot to act in an intentional and useful way (Figure 1.1).

![Figure 1.1: Simple representation of a robot’s components.](image)

A sensor can be defined as a device sensible to a particular physical quantity and capable to transform it in a measurable and transferable signal.

The most easy type of sensors are the switches; they provide only one datum, contact or non-contact and are used for example as limit switches for manipulators’ joints or impact sensors for mobile robots.

Other notable sensors are potentiometers and Hall effect measuring sensors. With the former one it can be calculated the pressure of the object on the sensor. Hall sensors are used for proximity switching, positioning, speed detection, and current sensing applications.

For long distances between the sensor and an external object the previously mentioned sensors are practically useless, in fact there are other kinds of devices like the ultrasonic sensors and the optic ones.

Examples of other frequently used sensors are temperature sensors, Global Positioning System (GPS), sonar system, laser sensors etc. Effectors are simply those parts of the robot that are moved by the actuators (i.e. engines), after the control system has
performed its calculations depending on the inputs from the various sensors and the algorithm used.

All the sensors described above are often simulated in a virtual environment since they could be too expensive to use in real world or it could be too risky to use them without a correct trial period. Thus simulation in robotics permits experimentation that would otherwise be expensive and/or time-consuming, it permits the developers to try ideas in dynamic, synthetic environments while collecting stimulus response data to determine the quality of the control system.

Several open source toolkits among proprietary ones are available for building and simulating robotic control systems, we are now going to mention some of them.

**1.2.1 Open Dynamics Engine (ODE)**

Russell Smith’s Open Dynamics Engine (ODE) is an open source physics engine with which one can simulate articulated rigid body dynamics. It is platform independent since it uses C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is utilised for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools [27]. In this way, the developer can simulate the physics of real-world objects independently of a graphics library. The ODE can be used to model all sort of objects in synthetic environments. At the core of ODE stands the Physics SDK; such development kit uses the mathematical formulae that represent the physical world to simulate the behaviours of the objects involved in the simulation. The usual routine is the following: the description of the physical world is given to the SDK besides the current state of the system, the forces that control the system and the time step interval is taken as input from the SDK. It computes the following state after the time interval, so that it can be drawn onto the screen.

The following is a source example showing a simple world with Mars’ gravity and a sphere that currently has some upward velocity. Given that the world has gravity, the upward velocity does not last long; eventually, the sphere reaches the apex and
begins its descent phase. After the initialisation is complete (that is, objects created
in the world and their attributes set), one can simulate the physics of the world with a
call to dWorldStep. To understand what is happening to the variables of this synthetic
environment, one can for example make regular calls to dBodyGetPosition passing as
argument the sphere’s identifier to get its current position [26].

#include <iostream>
#include <ode/ode.h>

#define time_step (float)0.1

int main()
{
    dWorldID myWorld_id;
dBodyID mySphere_id;
dMass sphereMass;
    const dReal *pos;
    float time = 0.0;

    /* Create a new world */
    myWorld_id = dWorldCreate();

    /* Create a sphere in the world */
    mySphere_id = dBodyCreate( myWorld_id );

    /* Set the world’s global gravity vector (Mars) -- x,y,z */
    dWorldSetGravity( myWorld_id, 0, 0, -3.77 );

    /* Set the Sphere’s position in the world -- x,y,z */
    dBodySetPosition( mySphere_id, 0, 0, 100 );

    /* Set the Sphere’s mass (density, radius) */
    dMassSetSphere( &sphereMass, 1, 2 );
dBodySetMass( mySphere_id, &sphereMass );

/* Give the sphere a small amount of upward (z) velocity */
dBodySetLinearVel( mySphere_id, 0.0, 0.0, 5.0 );

/* Run the simulation */
while (time < 5.0)
{
    /* Simulate the world for the defined time-step */
dWorldStep( myWorld_id, time_step );

    /* Get the current position of the sphere */
    pos = dBodyGetPosition( mySphere_id );

    /* Next time step */
    time += time_step;
}

/* Destroy the objects */
dBodyDestroy( mySphere_id );
dWorldDestroy( myWorld_id );

return 0;
}

Various video-games and some simulators use these libraries to represent the physics of the world they are simulating. For our interest robot simulators like Player Gazebo, anyKode Marilou and Cyberbots Webots are currently using the ODE physical engine.
1.2.2 Player project (Player/Stage/Gazebo)

1.2.2.1 Player

The Player Project creates Free Software that enables research in robot and sensor systems.

Player is a socket-based device server that allows control of a wide variety of robotic sensors and actuators. Player executes on a machine that is physically connected to a collection of such devices and offers a TCP socket interface to clients that wish to control them [2].

The Player robot server is probably the most widely used robot control interface in the world. Its simulation back-ends, Stage and Gazebo, are also very widely used [28].

Player provides a network interface to a variety of robot and sensor hardware. Player’s client/server model allows robot control programs to be written in any programming language and to run on any computer with a network connection to the robot. Player supports multiple concurrent client connections to devices, creating new possibilities for distributed and collaborative sensing and control. Running on the robot, Player provides a simple interface to the robot’s sensors and actuators over the IP network; the client program talks to Player over a TCP socket, reading data from sensors, writing commands to actuators, and configuring devices on the fly.

Because Player’s external interface is simply a TCP socket, client programs can be written in any programming language that provides socket support, and almost every language does. In fact currently it has client-side utilities available in C++, Tcl, Java, and Python. Further, Player makes no assumptions about how one might want to structure the robot control programs. The simulator allows multiple devices to present the same interface thus the same control code could work on more kinds of robot. This feature is very useful when combined with the Stage simulator; control programs written for Stage’s simulated robots will often work unchanged on real hardware.

Player does not implement any device locking, so when multiple clients are connected to a Player server, they can simultaneously issue commands to the same device. In general, there is no queueing of commands, and each new command will overwrite the old one. Not implementing locking was chosen in order to provide maximal power
and flexibility to the client programs. If multiple clients are concurrently controlling a single device, then those clients are probably cooperative, in which case they should implement their own arbitration mechanism at a higher level than Player. By the other hand if the clients are not cooperative, then the subject of research is presumably the interaction of competitive agents, in which case device locking would be a hindrance.

Player takes some influences from classic operating system design in the way that device interfaces are separated from device drivers. For example, in an operating system there is a joystick interface that defines the API for interacting with joysticks, and there are joystick drivers that allow the programmer to control various joysticks through that same API. Similarly, in Player, a device interface is a specification of data, command, and configuration formats, and a device driver is a module that controls a device and provides a standard interface to it. Probably the most used of such interfaces is the \textit{position2d} interface that covers ground-based mobile robots, allowing them to accept commands to make them move (either velocity or position targets) and to report their state (current velocity and position).

Player also provides transport mechanisms that allow data to be exchanged among drivers and control programs that are executing on different machines. The most common transport in use now is a client/server based on a TCP socket. The Player server is executed with a configuration file that defines which drivers to instantiate and how to bind them to hardware. The drivers run inside the Player server (often in multiple threads), and the user’s control program runs as a client to that server.

As a result of its network-centric architecture, Player permits any client, located anywhere on the network, to access any device. There were made experiments in this direction like works on concurrent control, in which approximately fifty independent agents were simultaneously controlling a single robot’s motors through Player. Similarly, Player allowed a team of robots bound in cooperative localisation to directly access each others’ sensors, thereby facilitating sensor fusion.

An addition to the server architecture is the “passthrough” driver. Executing within the context of a Player server, this driver acts as a client to another Player server. The passthrough driver connects to a remote server and provides a local proxy for a remote device by forwarding commands and data. In this way, remote resources can be
made to appear as local resources. For example let us consider the encapsulation of sophisticated algorithms into Player drivers. When algorithms are made into drivers, they must run within Player on the server side, which is often a robot. If the robot has only modest computational facilities, then it may not be well-suited to run, those time expensive algorithms. In this case, another instance of Player can run off-board, on a more powerful machine, with passthroughs providing data from and control of the remote devices.

1.2.2.2 Stage

Stage simulates a population of mobile robots, sensors and environmental objects. It has two original purposes; firstly to enable rapid development of controllers that will eventually drive real robots; and secondly to enable robot experiments without access to the real hardware and environments. More recently the sensor models have been extended and generalised beyond the limits of any available hardware, adding another purpose: to enable “what if?” experiments with novel devices that do not (yet) exist. The goal of these experiments is using Stage as a tool to determine the possible benefits of developing one type of sensor over another.

Another important feature of Stage is the linear scaling with population, i.e. it means that all sensor models algorithms are independent of population size. The essential design feature that enables scaling is that the ray tracing computation required for collision detection and sensor modelling is approximately $O(1)$ per robot. Ray tracing is by far the most frequent operation, performed possibly hundreds of times per robot per simulation time-step.

Let us now see in details how the physical objects are modelled and how ray tracing is implemented in Stage.

Stage models physical bodies as tree assemblies of “blocks”. Blocks are specified as arbitrary polygons on the $[x, y]$-plane, extruded into the $z$-axis. The block data structure consists of a pointer to the model that owns the block, an array of two-dimensional points $[x_0, y_0, \ldots, x_n, y_n]$, the array length $n$, and a $[z_{min}, z_{max}]$ pair indicating the extent of the block in $z$. This implies that the blocks can be rotated only around the $z$-axis. Because of the missing degrees of freedom, and adopting terminology
from computer graphics and video games, Stage is referred as a 2.5D (two-and-a-half dimensional) simulator [3]. Block trees can be constructed piecemeal by user code, specified in the “world-file” with arrays of points, or loaded from bitmaps in most common file formats (JPEG, PNG, BMP, etc.). Figure 1.2 shows two models supplied with Stage, both approximating the popular Pioneer 3DX robot.

![Figure 1.2: Pioneer 3DX robot (left) from MobileRobots Inc., and two virtual Stage robots models (centre and right).](image)

Collisions between blocks and range sensor data are computed using ray tracing. The population of 2.5D blocks is rendered into a 2-dimensional discrete occupancy grid by projecting their shapes onto the ground plane \((z = 0)\). Grid cell size is configurable, with a default size of \(0.02\) m. Each grid cell contains a list of pointers to the blocks that have been rendered into that cell. When a block moves, it must be deleted from cells that it no longer occupies and rendered into newly-occupied cells. Non-moving blocks such as walls are rendered into the grid only once. Very fast bit-manipulation and integer arithmetic are used to look up grid cell locations from simulation world coordinates specified in meters. Ray tracing involves walking through the nested grid data structure using Cohen’s integer line-drawing algorithm [4]. As the ray visits each non-empty cell, it is inspected the \(z\)-extent of the blocks at that cell, and the properties of the models owning the block, to see if they interact with the ray.

The Stage user interface was built using OpenGL and the Fast Light Toolkit framework (FLTK), chosen for speed, ease of use, and wide availability. The OpenGL interface provides a full 3D view of the world, and alpha blending and antialiasing provide sophisticated effects for visualisation of sensor data, motion history, etc. Moreover
OpenGL takes advantage of graphics processor (GPU) hardware, to ensure the maximal speed in computing and rendering the 3D view of the world.

The following is a sample source code of a simple robot modelled with Stage:

```c++
#include <stage.hh>

int main( int argc, char* argv[] )
{
    StgWorld::Init( &argc, &argv ); // initialise libstage
    StgWorldGui world( 800, 600, "My Stage simulation" );
    world.Load( argv[1] ); // load the world file

    // get a pointer to a mobile robot model
    // char array argv[2] must match a position model named in
    // the world file
    StgModelPosition* robot = (StgModelPosition*)world.GetModel( argv[2] );
    robot->Subscribe();

    world.Start(); // start the simulation clock running

    while( !world.TestQuit() )
    {
        if( world.RealTimeUpdate() )
        {
            // [ read sensors and decide actions here ]
            robot->Do( forward_speed, side_speed, turn_speed );
        }
        delete robot;
        exit( 0 );
    }
}
```

The next piece of code describes the so called configuration “world-file” that is used by Stage to construct the model of the world in which the robot(s) will move and interact with:
Simulation worlds are specified using a simple tree-structured configuration file, or by user programs making individual object creation and configuration function calls. This is an example of a simple configuration file

# size of the world in meters [x y z]
size [16 16 3]

# create and specify an environment of solid walls loaded from an image model

( # size of the model in meters [x y z] - fill the whole world
size3 [16 16 0.5] color "gray30"

# draw a 1m grid over this model
gui_grid 1

# this model can not be moved around with the mouse
gui_movemask 0

# interpret this image file as an occupancy grid and
# construct a set of blocks that fill this grid
bitmap "bitmaps/cave.png"

# model has a solid boundary wall surrounding the blocks
boundary 1
)

# create and specify a mobile robot carrying a laser scanner
position

( name "MySimpleRobot"
In Figure 1.3 there is the result of the processing of the “world-file” listed above.

Figure 1.3: Graphical representation of the virtual world described in the “world file” in Stage.

The goal of the Player/Stage project is to provide Open Source software infrastructure to support experimental research with multi-robot systems (MRS). To this end the project has developed the robot device server Player and the multiple robot simulator Stage. In addition to facilitating ongoing MRS research, Player and Stage offer new opportunities for research in emerging areas, including distributed sensing and control systems [2].
1.2.2.3 Gazebo

Gazebo is a multi-robot simulator for both indoor and outdoor environments. Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in a three-dimensional world. It generates realistic sensor feedback, object collisions and dynamics [28]. Gazebo is normally used in conjunction with the Player device server. When used with Gazebo, Player provides simulated data in the place of real sensor data. In principle, client programs cannot tell the difference between real devices and the Gazebo simulation of those devices. Gazebo can also be controlled through a low-level C API (libgazebo).

![Figure 1.4: A screenshot of the Gazebo robot simulator.](image)

The graphical user interface allows the user to navigate through the world, and receive information about the simulation. Movement through the world is accomplished via a combination of the mouse and keyboard. The mouse is used to free-look when left
mouse button is pressed. Only the pitch and yaw of the camera is controllable.

As for Stage the world file contains a description of the world to be simulated by Gazebo. It describes the layout of robots, sensors, light sources, user interface components, and so on. The world file can also be used to control some aspects of the simulation engine, such as the force of gravity or simulation time step.

Gazebo world files are written in XML. The world consists mainly of model declarations. A model can be a robot (e.g. a Pioneer2AT or SegwayRMP), a sensor (e.g. SICK LMS200), a static feature of the world (e.g. Terrain) or some manipulable object.

Other elements within a world file indicate how to render a scene via the Ogre declaration, attributes of the GUI via the gui declaration, and properties of the physics engine via the ode declaration. These declarations are required by Gazebo.

Each world file is encapsulated by:

```xml
<gazebo:world>
...
</gazebo:world>
```

By convention, Gazebo uses a right-handed coordinate system, with x and y in the plane, and z increasing with altitude. The tag `<xyz;>` is used to indicate an object’s position (x, y and z coordinates); the tag `<rp y;>` is used to indicate an object’s orientation (Euler angles; i.e., roll, pitch and yaw). For example, `<xyz;>1 2 3</xyz;>` indicates a translation of 1 metre along the x-axis, 2 metres along the y-axis and 3 metres along the z-axis; `<rp y;>10 20 30</rp y;>` indicates a rotation of 30 degrees about the z-axis (yaw), followed by a rotation of 20 degrees about the y-axis (pitch) and a rotation of 10 degrees about the x-axis (roll).

A model is a physical entity within the world. It consists of one or more bodies, each of which contains one or more geometries or sensors. For example the following is a description of a box:

```xml
<model:physical name="box_model">
  <xyz>0 1.5 0.5</xyz>
  <rp y>0.0 0.0 0.0</rp y>
  <canonicalBody>box1_body</canonicalBody>
  <static>false</static>
</model:physical>
```
xyz is the position relative to parent,

rpy is the orientation relative to parent,

canonicalBody is the body used to attached child and parent models,

static set to true to make the model immovable,

mesh is the “skin” of the geometry,

size is the size of the geometry,

density is the density of the geometry,

material is the material used to colour and texture the geometry.

The entire physical characteristics of a model is encapsulated within the xml. Joints are attached between two bodies with the same model. Complex models can become quite long and complicated to use, especially when the same model is required numerous times in the same world. To alleviate this issue, model’s can be written in separate files and included within a world.
Sometimes it is useful to include the same model with slightly different parameters. This can be accomplished by creating a new model, with a nested include:

```xml
<model:physical name="box_model1">
    <xyz>1 1 0</xyz>
    <include embedded="true">
        <xi:include href="box.model"/>
    </include>
</model:physical>
```

Eventually physics can be simulated, using the ODE physics simulator, in this way:

```xml
<physics:ode>
    <stepTime>0.03</stepTime>
    <gravity>0 0 -9.8</gravity>
    <cfm>0.008</cfm>
    <erp>0.2</erp>
</physics:ode>
```

**stepTime** is the number of seconds the physics simulation should progress during each update,

**gravity** is the vector that describes the direction and magnitude of gravity,

**cfm** is the global constraint force mixing,

**erp** is the global error reduction parameter.

### 1.2.3 Simbad

There are also robotics simulators written in other languages as for example in Java like Simbad.

Simbad is an open source Java 3D robot simulator for scientific and educational purposes. It is mainly dedicated to researchers and programmers who want a simple basis for studying Situated Artificial Intelligence, Machine Learning, and more generally AI algorithms, in the context of Autonomous Robotics and Autonomous Agents [23].
It includes a rich graphical user interface (GUI) for visualisation not only of the robot’s actions but also from the robot’s perspective.

What makes Simbad interesting is that it’s simple to use and allows the user to create new robot behaviours quickly. With the simulator, the developer can create an environment, and then develop the robot controller using a variety of sensors.

The simulator provides the following functionalities:

- Single or multi-robot simulation.
- Colour Mono-scopic cameras.
- Contact and range sensors.
- Online or batch simulation.
- Python scripting with jython.
- Extensible user-interface.
- Simplified physical engine.

A very simple examples is shown in the following script:

```java
int sonar_id, total_sonars;
// If at least one sensor has a hit
if (sonars.oneHasHit()) {
    // Find out how many sonars are on the robot
    total_sonars = sonars.getNumSensors();
    // Iterate through each sonar
    for (sonar_id = 0; sonar_id < total_sonars; sonar_id++) {
        // Does this one have a hit?
        if (sonars.hasHit(sonar_id)) {
            // Emit the details (angle, range)
            System.out.println("Sonar hit at angle "+sonars.getAngle(0) + " at range "+sonars.getMeasurement(i));
        }
    }
}
```
What really makes Simbad so useful is its console for robot simulation and visualisation. As Figure 1.5 shows, the Simbad console gives the user a real-time view of the world, an inspector panel that provides robot details, and a control panel for managing the simulation.

![Simbad robot simulator's GUI.](image)

Simbad is free to use and modify under the conditions of the GPL (GNU General Public Licence).
1.2.4 Webots

Among proprietary simulators figures Webots of the Cyberbotics Ltd. company. With Webots the user can design complex robotic setups, with one or several, similar or different robots, in a shared environment. The properties of each object, such as shape, colour, texture, mass, friction, etc., are chosen by the user. A large choice of simulated sensors and actuators is available to equip each robot. The robot controllers can be programmed with the built-in IDE or with third party development environments [34].

Webots™ has a number of essential features intended to make this simulation tool both easy to use and powerful:

- Models and simulates a great variety of mobile robot, including wheeled, legged and flying robots.
- Includes a complete library of sensors and actuators.
- Lets the developer program the robots in C, C++ and Java, or from third party software through TCP/IP.
- Transfers controllers to real mobile robots.
- Uses the ODE (Open Dynamics Engine) library for accurate physics simulation.
- Creates AVI or MPEG simulation movies for web and public presentations.
- Includes many examples with controller source code and models of commercially available robots.

A library of sensors is provided so that the developer can plug a sensor in the robot model and tune it individually (range, noise, response, field of view, etc.). This sensor library includes distance sensors (infra-red and ultra-sonic), range finders, light sensors, touch sensors, global positioning sensor (GPS), inclinometers, compass, cameras (1D, 2D, colour, black and white), receivers (radio and infra-red), position sensors for servos, incremental encoders for wheels.

The simulation system used in Webots uses virtual time, making it possible to run simulations much faster than it would take on a real robot. Depending on the complexity
of the setup and the power of the computer, simulations can run up to 300 times faster than the real robot when using the fast simulation mode [24]. The basic simulation time step can be adjusted to suit the developer’s needs and a step-by-step mode is available to study in detail how the robots behave.

![Webots graphical interface](image)

Figure 1.6: Webots graphical interface.

The graphical user interface of Webots allows the user to easily interact with the simulation while it is running. By dragging the mouse, one can change the viewpoint position, orientation and zoom using the mouse wheel. Pressing the shift key while dragging the mouse allows the user to move or rotate objects. This feature facilitates interactive testing.

Here there is an example on how to program a robot using the C language.

```c
#include <robot.h>
#include <differential_wheels.h>
#include <distance_sensor.h>
DeviceTag ir;
void my_robot_reset() {
    ir = robot_get_device("ir");
}
```
void main() {
    robot_live(my_robot_reset);
    for(;;) { /* infinite loop */
        if (distance_sensor_get_value(ir)>100)
            differential_wheels_set_speed(0,0);
        else differential_wheels_set_speed(10,10);
        robot_step(64) /* run for 64 ms */
    }
}

The robot is a differential wheeled robot equipped with an infra-red distance sensor named “ir” looking forward. The robot will stop moving if the distance sensor detects an obstacle and restart moving when the obstacle is no longer detected. A similar Java programming interface is also included. Once tested in simulation the robot controllers can be transferred to real robots.

1.2.5 Microsoft Robotics Developer Studio

Microsoft® Robotics Developer Studio (RDS) is a Windows-based environment to create robotics applications for a variety of hardware platforms.

Microsoft Robotics Developer Studio can support a broad set of robotics platforms by either running directly on the platform (if it has an embedded PC running Windows) or controlling it from a Windows PC through a communication channel such as Wi-Fi or Bluetooth®.

RDS includes various frameworks the most important of which are:

- Concurrency and Coordination Runtime (CCR)
- Decentralized Software Services (DSS)
- Visual Programming Language (VPL)
- Visual Simulation Environment (VSE)

The CCR addresses the need of service-oriented applications to manage asynchronous operations, deal with concurrency, exploit parallel hardware and deal with partial failure.
Decentralized Software Services (DSS) is a lightweight .NET-based runtime environment that sits on top of the CCR. DSS provides a lightweight, state-oriented service model that combines the notion of representational state transfer, with a system-level approach for building high-performance, scalable applications. A primary design goal of DSS is to couple performance with simplicity and robustness. This makes DSS particularly suited for creating applications as compositions of services regardless of whether these services are running within the same node or across the network.

Microsoft Visual Programming Language (VPL) is an application development environment designed on a graphical dataflow-based programming model. Rather than series of imperative commands sequentially executed, a dataflow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. Non-programmers can create robot applications using this visual programming environment that enables to create and debug robotics programs very easily. The user has only to drag and drop blocks that represent services, and connect them. It is also possible to take a collection of connected blocks and reuse them as a single block elsewhere in the program.

The Visual Simulation Environment (VSE) provides a high-fidelity simulation environment powered by NVIDIA™ PhysX™ engine for running game-quality 3D simulations with real-world physics interactions [35].

Figure 1.7: A sample environment provided with VSE.
In Figure 1.7 it is shown a sample 3D environment provided by the VSE; meanwhile in Figure 1.8 there are two screenshots of a robot with differential drive, laser range finder and bumper array, in the first one the general graphical overview is shown, in the second image the physics primitive view, which shows how the table and robot are approximated by solid shapes, is shown.

Figure 1.8: A robot shown in the virtual environment.
1.3 Marine robotics

The field of robotics has applications in various disciplines and one of the most interesting and fast growing field of research is the one of marine robotics. It involves the study and development of a family of robots that can operate in open waters like seas or oceans and as well in rivers or lakes. An important difference between this field of robotics and the one that deal with autonomous ground vehicles is the study of the water physics (e.g. hydrodynamics) and the responses of the developing robots to the external stimuli caused by this particular environment. These studies led to a various types of vehicles both underwater and surface ones.

Marine robotics is useful nowadays for various applications especially for time-consuming or dangerous operations such as surveillance, hydrographic survey, mine-hunting, ocean exploration, marine archaeology or offshore industry, where inspection of submerged structures is crucial to both human safety and the protection of the environment.

We will focus our attention on those marine robots that travel on the water surface, such robots are called Autonomous Surface Vehicles (ASV) if their movements are completely autonomous or Unmanned Surface Vehicles (USV) if they are guided remotely via a human-computer interface. As second instance we will analyse the theory that lies behind the study of the marine environmental characteristics like sea currents, waves formation and winds that interact massively with the ASVs and have an important role in the developing of the controls for such vehicles.

1.3.1 Unmanned Surface Vehicles & Autonomous Surface Vehicles

First Unmanned Surface Vehicles were developed during the second world war, but they only proliferate from the 1990s due to better technologies and to some governments efforts in researching in this field like in the USA.

Considering the following list of categories for such vehicles,

- Small (<1 t)
- Medium (< 100 t)
• Large (< 1000 t)

• Extra large (> 1000 t)

Nowadays all the USV are in the small or medium one. Most USVs are about the size of recreational watercraft, i.e. 2 to 15 m long with displacements of 1.5 to 10 t. Some can operate at more than 35 knots in calm water [15].

As we said most of this kind of vehicles were built in the USA especially by the Navy. The interests in USVs for reconnaissance and surveillance missions emerged in the late 1990s, with the development of the Autonomous Search and Hydrographic Vehicle (ASH), later called the Owl, and the Roboski. The Roboski, initially developed as Shipboard Deployed Surface Target (SDST) as a jet-ski type target for ship self-defence training, now also serves as a reconnaissance vehicle test-bed. By the early 2000s, several concepts for stealthy USV sensor platforms have been proposed and are under consideration by the surface fleet.

In the military field there are unmanned surface vessels such as the testbed of the SSC San Diego, based on the Bombardier SeaDoo Challenger 2000, the Israeli Stingray USV, with a top speed up to 40 knots, and Protector USV, equipped with electro-optic sensors, radar, GPS, inertial navigation system and a stabilised 12.7 mm machine gun, and the QinetiQ Ltd shallow water influence mine-sweeping system (SWIMS).

Among civil applications of this type of robotics, we can mention the family of autonomous vessels developed at MIT for education and civil applications, consisting of the fishing trawler-like vehicle ARTEMIS, the catamarans ACES (Autonomous Coastal Exploration System) and Auto-Cat, and the kayak SCOUT (Surface Craft for Oceanographic and Undersea Testing). These USVs demonstrated the feasibility of automatic heading control and DGPS (Differential Global Positioning System)-based waypoint navigation, as well as the possibility of operating autonomously collecting hydrographic data.

Other interesting applications are those civil vessels built in Europe like:

• the Measuring Dolphin, designed and developed by the University of Rostock (Germany), for high accuracy positioning and track guidance and carrying of measuring devices (e.g. depth and current) in shallow water;
• the autonomous catamaran Delfim, developed by the DSOR lab of Lisbon IST-ISR as a communication relay for a companion AUV (Autonomous Underwater Vehicle) in the European Union funded project ASIMOV (Advanced System Integration for Managing the coordinated operation of robotic Ocean Vehicles), and then exploited as a stand-alone unit for collecting bathymetric maps and marine data;

• the autonomous catamaran Charlie by CNR-ISSIA Genova (Italy), originally designed, developed and accomplished for the collection of sea surface micro-layer, and then upgraded for robotic research on autonomous vessels;

• the autonomous catamaran Springer, developed by the University of Plymouth (UK), for tracing pollutants.

The above-mentioned prototypes reveal different approaches in basic design issues and trends as for example:

• the catamaran hull shaped vehicle, which optimises the easiness of mounting and the loading capacity of different payloads, minimising movement in rough sea, is usually preferred by research developers, while, in the military field, rigid inflatable boats (RIBs) are preferred due to their diffusion as standard vessels for naval operations and their capability in carrying on larger fuel tanks;

• electrical power supply is preferred for environmental sampling applications, where the constraint of not polluting the operating area is mandatory, while, when long missions have to be performed, e.g. in the case of coastal surveillance or MCM operations, gasoline propulsion is more practical;

• the design and development of new vehicles is typical of research institutions, while the needs of low cost development and easy transfer to the end-user motivated, even in military applications, the development of conversion kits to transform existing vessels in remote controlled ones;

• the goal of fully autonomous operations is the pole star of civil and research applications, while military applications see in remote controlled vessels the solution,
through suitable human-computer interactions, to optimise system performances in many different mission conditions [16].

Miniaturised sensors and electronics, as well as communications equipment allow, for the development of small payload packages with significant information, gathering capability for almost any USV platform. This includes digital and video cameras, EO/IR sensors, and radars.

Although this means that most USV can provide some level of "Intelligence, Surveillance, and Reconnaissance" (ISR) capability, larger and more complex towed sensors, required for a robust mine warfare or anti-submarine warfare competence, require significant payload and towing capability, as well as platform stability and endurance.

Generally the larger USV platforms tend to be more stable and offer more mission functionality. An additional consideration for platform stability is the basic design of the USV. Most USVs are adapted from manned surface vessel designs that necessarily have to deal with the fluctuations of the air/water interface to accommodate human occupants. The developers of USVs realised that this limitation need not apply to unmanned systems, and designed the USV to be an air-breathing semi-submersible, where the entire vehicle except for a snorkel is submerged to improve stealth and platform stability.

Let us now investigate some of the main components and technologies used for the guidance and control of such vehicles.

Operational results showed that using basic autopilots to control the vehicle’s heading and speed, in many practical applications a simple P(I)D heading controller is sufficient for guaranteeing satisfactory performance in controlling the horizontal motion of a USV.

PID stands for Proportional, Integral and Derivative control. In such control can figure also only the PD components as well the PI. The underlying idea of this control system is to minimise errors between the actual value that is being controlled (i.e. the measured one) and the one estimated or wanted by the navigation algorithm.

The PID algorithm is composed by three parts:

- **proportional** so called because its effect is proportional to the error;
**integral** because it produces as output a correction that represents the integral of the error in time. The integral controller can be considered as an “looking back”, summing all the errors and then responding;

**derivative** because it generates a correction that is function of the first derivative of the error. Derivative control can be considered as an anticipatory control that measures the existing rate of change of error, anticipates incoming larger error, and applies correction before the larger error is arrived [17].

![PID control system diagram](image)

Figure 1.9: The PID control system.

The adopted formulae are the following:

\[
V = K_p \cdot e_q + K_d \cdot \dot{e}_q + K_i \cdot \int e_q (t) \, dt;
\]

\[
e_q = q_d - q;
\]

\[
\dot{e}_q = \frac{de_q}{dt}.
\]

Where \(K_p\) is the proportional constant, \(K_i\) is the integral constant, \(K_d\) is the derivative constant, \(e\) represents the error i.e. the difference between the desired position \(q_d\) and the actual position \(q\).

In many applications, the vehicle is required to follow desired paths with great accuracy with a speed profile specified by the end-user, relaxing the strong constraints typical of trajectory tracking, defined as requiring the vessel to follow a time-parametrised reference curve, i.e. to be in specific points at specific instants of time. Thus, the so-called
path following problem is faced, i.e. the vehicle has to follow a planar path without temporal constraints.

Path following algorithms have to define, compute and reduce to zero the distance between the vehicle and the path as well as the angle between the vector representing the vessel speed and the tangent to the desired path.

The following data are usually considered while modelling a control system for a marine robot as one can see in Figure 1.10.

![Figure 1.10: Nomenclature of basics data for a marine robot.](image)

Taken as origin of a Cartesian plane an earth-fixed point \(<e>\), the position and orientation of a vessel are \([x \ y \ \psi]\) and considering a local coordinate system of the vessel respect of its origin \(<b>\), where surge and sway velocities ([\(u v\)] absolute, [\(u_r v_r\)] with respect to the water), yaw rate \(r\), are represented. Denoting with \([\dot{x}_c \ \dot{y}_c]^T\) the
sea current, the body-fixed absolute velocity and velocity with respect to the water are related by:

\[
\begin{align*}
    u &= u_r + \dot{x}_C \cdot \cos(\psi) + \dot{y}_C \sin(\psi); \\
    v &= v_r - \dot{x}_C \cdot \sin(\psi) + \dot{y}_C \cdot \cos(\psi).
\end{align*}
\]

and the vehicle kinematics is usually expressed in the earth-fixed coordinate system \( \langle e \rangle \) as

\[
\begin{align*}
    \dot{x} &= u_r \cdot \cos(\psi) - v_r \cdot \sin(\psi) + \dot{x}_C; \\
    \dot{y} &= u_r \cdot \sin(\psi) + v_r \cdot \cos(\psi) + \dot{y}_C; \\
    \dot{\psi} &= r.
\end{align*}
\]

relating vehicle speed in the earth-fixed and body-fixed coordinate systems.

### 1.3.2 Winds, waves and sea currents

In order to build a correct control system and also to develop a good software simulator, the developer needs some basic knowledges about the marine environment; in this section we are going to see how this environment can be represented in a mathematical and physical model and how it interacts with the marine vehicles.

The most important disturbances by the marine environment to the motion of a vessel are the waves, generated by wind and the sea or ocean currents.

Let us start from the creation of the waves; they are generated by wind that when starts blowing, on the water surface appear small wavelets. This increases the drag force which allows short waves to grow. These short waves continue to grow until they finally break and their energy is dissipated. During this phase the sea state is said *developing sea* and waves have high frequency and form a spectrum with peak at a relative high frequency; after the wind has blown for sufficiently long time from the same direction, the waves it creates, reach maximum size, speed and period beyond a certain distance (fetch) from the shore the state is called *fully developed sea.*
Because the waves travel at speeds close to that of the wind, the wind is no longer able to transfer energy to them and the sea state has reached its maximum. In this state the waves have lower frequency and greater length and they create a spectrum with a low peak frequency.

Usually wind-generated waves are represented as a sum of a large number of wave components. Let the wave number of one single wave component be denoted by \( k_i \). Hence,

\[
k_i = \frac{2\pi}{\lambda_i}
\]

where \( \lambda_i \) is the wave length. The wave elevation \( \zeta(x, t) \) of an irregular sea propagating along the positive x-axis can be written as a sum of wave components [18]:

\[
\zeta(x, t) = \sum_{i=1}^{N} A_i \cos(\omega_i t - k_i + \phi_i) + \sum_{i=1}^{N} \frac{1}{2} k_i A_i^2 \cos 2(\omega_i t - k_i + \phi_i) + O(A_i^3)
\]

where \( \phi_i \) is a random phase angle with time in \([0, 2\pi]\).

Linear wave theory represents a 1st-order approximation of the wave elevation \( \zeta(x, t) \). This corresponds to the first term \( A_i \) in the above formula. 1st-order wave disturbances will describe the oscillatory motion of the vehicle while the 2nd-order term represents the wave drift forces.

From wave theory it can be shown that the connection between the wave number \( k_i \) and the circular frequency \( \omega_i = 2\pi/T_i \) is:

\[
\omega_i^2 = k_i g \tanh(k_i d)
\]

(1.1)

\( d \) is used to refer to the water depth. Such equation is often called the dispersion relation; for infinite water depth (i.e. for \( d/\lambda_i > 1/2 \)) the dispersion relation reduces to \( \omega_i^2 = k_i g \) since \( \tanh(k_i d) \to 1 \) as \( d/\lambda_i \to \infty \).

Pierson and Moskowitz in 1963 [21] developed a wave spectral formulation for fully developed seas from analyses of wave spectra in the North Atlantic Ocean. The Pierson Moskowitz (PM) spectrum is:
\[ S(\omega) = A\omega^{-5} \exp\left(-B\omega^{-4}\right) \]

where

\[
A = 8.1 \cdot 10^{-3} g^2; \\
B = 0.74 \cdot \left(\frac{g}{V}\right)^4.
\]

Here \( V \) is the wind speed at a height of 19.4 metres over sea surface and \( g \) is the gravity constant.

The PM-spectrum can be reformulated in terms of significant wave height \( H_s \) (mean of the one-third highest waves) as:

\[
A = 8.1 \cdot 10^{-3} g^2; \\
B = 0.0323 \cdot \left(\frac{g}{H_s}\right)^2 = \frac{3.11}{H_s}.
\]

From observations the wind speed \( V \) and significant wave height \( H_s \) are related through:

\[
H_s = 0.21 \cdot \frac{V}{g}
\]

The modal frequency \( \omega_0 \) for the PM-spectrum is found by requiring that:

\[
\left(\frac{dS(\omega)}{d\omega}\right)_{\omega=\omega_0} = 0
\]

the computation yields:
\[ \omega_0 = \sqrt[4]{\frac{4B}{5}} ; \]
\[ T_0 = 2\pi \sqrt[4]{\frac{5}{4B}} \]

where \( T_0 \) is the modal period. Substituting \( A \) and \( B \) in the above formula yields:

\[ \omega_0 = 0.88 \frac{g}{V} = 0.4 \sqrt{\frac{g}{H_s}} \]

For a ship moving with forward speed \( U \), the wave frequency \( \omega_0 \) will be modified according to [5]:

\[ \omega_e (U, \omega_0, \beta) = \omega_0 - \frac{\omega_0^2}{g} U \cos (\beta) \]

where \( \omega_e \) is the encounter frequency, \( g \) the acceleration of gravity, \( U \) the total speed of the ship and \( \beta \) the angle between the heading and the direction of the wave.

In order to know the length \( \lambda_i \) of a wave given the modal frequency \( \omega_i \), for a given finite depth water, equation 1.1 cannot be resolved analytically. There are several methods to find an empirical formulation of such equation like in [20] or more recently in [19], but the most efficient in terms of minimisation of the error between the estimated result and the actual one (maximum 0.1%) is the equation developed by Hunt in [8]:

\[ G = (\omega_i)^2 \cdot \frac{d}{g} ; \]
\[ F = G + \frac{1}{1 + 0.6522 \cdot G + 0.4622 \cdot G^2 + 0.0864 \cdot G^4 + 0.0675 \cdot G^5} ; \]
\[ \lambda_i = T_i \cdot \sqrt{\frac{g \cdot d}{F}} . \]

Let now introduce the study of the ocean or sea currents. Currents in the upper layers of the sea are mainly generated by the winds system above the water surface. Besides these wind generated currents, other phenomena like the heat exchange at the sea surface
and the salinity changes, develop an additional sea current component, usually referred to as *thermohaline currents*. Furthermore, since the Earth is rotating, the Coriolis force will try to turn the major currents to the right in the northern hemisphere and to the opposite direction in the southern hemisphere. Finally there is also a tidal component that arise from planetary interactions like gravity.

During the 10th ISSC [22] it was proposed that the surface current velocity can be written as a sum of the following velocity components:

\[ V_c = V_t + V_{lw} + V_s + V_m + V_{set-up} + V_d \]

where \( V_t \) is the tidal component, \( V_{lw} \) is the component generated by local wind, \( V_s \) is the component generated by non-linear waves, \( V_m \) is the component from major ocean circulation (e.g. Gulf Stream), \( V_{set-up} \) is the component due to set-up phenomena and storm surges, \( V_d \) are the local density driven current components governed by strong density jumps in the upper ocean.

For computer simulations the above equation can be simplified and the average current velocity can be generated by using a 1st-order Gauss-Markov Process. For instance \( V_c(t) \) can be described by the following differential equation (as in [5]):

\[ \dot{V}_c(t) + \mu_0 V_c(t) = w(t) \]

where \( w(t) \) is a zero mean Gaussian white noise sequence and \( \mu_0 \geq 0 \) is a constant. In many case it is sufficient to choose \( \mu_0 = 0 \) which simply corresponds to a random walk, that is time integration of white noise.

This process must be limited such that \( V_{min} \leq V_c(t) \leq V_{max} \) in order to simulate realistic ocean currents. The following algorithm utilising Euler integration and a simple limiter can be used for this purpose:

1. Initial value: \( V_c(0) = 0.5 \cdot (V_{max} + V_{min}) \);

2. Euler Integration step with sampling time \( h \):

\[ V_c(k + 1) = V_c(k) + h \dot{V}_c(k) \]
3. Limiter: if ($V_{c}(k + 1) > V_{\text{max}}$) or ($V_{c}(k + 1) < V_{\text{min}}$) then

$$V_{c}(k + 1) = V_{c}(k) - h \dot{V}_{c}(k)$$

4. $k = k + 1$, return to step 2.

A similar algorithm can be used to simulate time-varying directions of the sea currents.
2 MATERIALS AND METHODS

In this chapter will be enlightened the tools and the techniques that were useful to accomplish the task.

In the first section is shown the structure of the marine robot, whose behaviour the simulator has to emulate; secondly there is the overview of the tools used to build the physical model, they are MATLAB, Simulink and the Real Time Workshop; in the third section the C++ libraries used for the graphics, called Qt, are outlined.

2.1 Robot structure

Now we can analyse the structure of the marine robot from the point of view of our interests. The HydroNet robot is a carbon-fiber catamaran with length of 1991 mm and width of 1164 mm [25].

The robot, an Autonomous Surface Vehicle (ASV), was built considering the following operative scenario and restrictions. The area to be monitored for coastal waters is fixed to 10 x 3 km starting from the coast. Considered the use of 3 robots in the area, there has been identified for each marine robot an area of roughly 10 km$^2$ to monitor. Furthermore in order to develop a vehicle easily usable by end-users, such as environmental agencies, it has been decided to limit the length and weight for ease of transportability and to be able to deploy the robots without needing a dedicated winch. Finally to navigate in shallow water the ASV has to exhibit a low draft and be enabled with obstacle avoidance capabilities. Such considerations led to the following requirements:

- range of more than 20 km during one mission;
- cruise speed of 2 kns (\( \sim 1 \text{ m/s} \));
• ability of sampling down to a depth of 50 m;
• length < 2m and maximum weight < 80 kg;
• maximum operative conditions of sea force 5 (wind speed 17-21 kns);
• obstacle avoidance capabilities and low draft for navigation in shallow waters;
• need of protection for propellers/rudders against floating objects (such as plastic bags).

Each robot is connected among with buoys to a so called Ambient Intelligence (AmI) infrastructure, that executes adaptive/cooperative sampling algorithms, supervises and manages the system sending commands to the different agents and provides an interface to the users to control the platform.

The robot has a sampling system (a probe lowered by a winch) to sample waters up to 50 meters of depth. To increase the endurance it has been used hybrid locomotion approach: electric propulsion using two propellers and a sail used to spare energy when the robot moves downwind.

The main components constituting the robot are listed below:

• Supervisor: it is the main processor managing all the robot operations. Furthermore, through a radio module, it is able to communicate with the Ambient Intelligence core to receive command and to send information;

• Obstacle Avoidance Module: it is an electronic board receiving the data from the range sensors and running obstacle avoidance algorithms. The range sensors present on the robot to avoid static and dynamic obstacles are:

  – a laser scanner, to avoid obstacles above the water surface;
  – a sonar, to avoid obstacles under the water surface;
- an altimeter is also present to measure the depth of seafloor.

- Localisation System: it is composed by different sensors (a compass, a GPS, a velocity sensor) to estimate robot speed and position;

- Sampling System + Sampling Control: the sampling system is constituted by a probe lowered by a winch. The collected water is then routed to the appropriate sensors to detect the compound of interest;

- Four slots for chemical sensors able to analyse the sampled water to detect concentrations of Cr, Cd, Hg and dispersed oil;

- PMU (Power Management Unit) + Battery Pack: they manage the robot energy;
- Two propellers (one for each hull) and two rudders actuated by a single motor: they represent the robot locomotion system;

- Anemometer, Sail Actuator and Sail Driver Electronics: on the base of the measured wind, the robot can control the boom to use the sail for the navigation and, consequently, to save electric energy.

About the sensors, the laser scanner is a Laser Scanner Hokuyo UTM30LX and detects obstacles above the water level up to a distance of 30 metres (Figure 2.2(left)). Interface to the sensor is through USB 2.0 with an additional synchronous data line to indicate a full sweep. Scanning rate is 25 milliseconds across a 270 degree range.

For the sonar has been chosen a Sonar Tritech Micron DST in order to detect obstacles under the water level up to a distance of 50 metres (Figure 2.2(right)). It is the smallest digital CHIRP sonar in the world. CHIRP technology is a feature normally associated with much larger, more expensive systems, in fact it dramatically improves the range resolution compared with conventional scanning sonars, improves discrimination between closely spaced targets, reduces power consumption, from high speed digital circuitry.

Figure 2.2: The laser scanner sensor (left) and the sonar sensor (right).

Let us now inspect the structure of the control system. As described in [14], the modules communicate with the main controller (TITAN PC104 board) through a CAN
bus. A second TITAN board is present to manage the obstacle avoidance and mapping algorithms and connected through an Ethernet channel to the main controller to send updated maps of the area and commands to avoid detected obstacles.

The software on the main controller, called Supervisor, runs under a Linux Embedded OS. Different processes are present communicating each other using Unix Domain Sockets:

**Navigation**

![Diagram of navigation systems of the robot]

Figure 2.3: Overview of the navigation systems of the robot.

**Communication manager**: written in the flexible LUA dynamic scripting language, manages the data exchange with the Ambient Intelligence (AmI);

**Manager**: manages the mission; reads the data from CAN bus, receives the mission commands (waypoints, sampling parameters..) from the Communication manager coming from AmI and dispatches them to the different processes;
**Environmental sensors & probe**: manages the operations of the chemical sensors and via Bluetooth sends the sampling parameters (e.g. sampling depths, physical sensors to use) to the sampling probe and receives the acquired data once the probe returns onboard inside the deck;

**Navigation**: receives the navigation commands from Manager and executes the navigation algorithms.

The latter is very important for the realisation of an autonomous marine robot; let us see some details on how it works.

The Navigation system is based on a three-layer structure:

- **AmI** sends the navigation parameters to the robot (waypoints, cruise speed, etc...).
  The user can set the mission of the robot (also changing the mission on the fly) using a graphical interface;

- the robot builds its own path from its current position to the next waypoint;

- then, the robot follows the path using a path-following and decoupled sliding mode autopilots (one controller for surge speed and one for steering dynamics) avoiding the detected obstacles. On the base of the measurements coming from the range sensors, the robot updates the map of the environment to improve the performance of the path-planner.

The simulator was created to substitute some functionalities of the main controller (TAITAN) in order to simulate some data (e.g. the position of the robot, the sea currents state, the cruise speed etc...) and to communicate with the other modules and test both the hardware answers to the simulation and the ones of the software modules.
2.2 MATLAB, Simulink and the Real Time Workshop

One of the most up-to-date, less error-prone and less time consuming way of creating a model is to use a mathematical framework like MATLAB and a modelling tool like MATLAB’s Simulink. These tools cover basically all the most commonly used functions of mathematical simulation, from continuous to discrete functions, from aerospace function blocksets to signals processing and communication functions, 3D animations, image processing and many others.

Finally another MATLAB embedded technology was used, the Real Time Workshop, that enables the developer to transform a Simulink model into a very efficient and widely
usable C or C++ source code.

An other particular reason on why these tools were chosen is that a Simulink model can be extended every time the developer needs it and re-translate it via the Real Time Workshop so in few steps the developer can easily update the source code he or she is using for the developing of the project.

2.2.1 MATLAB

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualisation, data analysis, and numeric computation. Using the MATLAB product, one can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran [30].

The MATLAB system consists of these main parts:

**Desktop Tools and Development Environment** This part of MATLAB is the set of tools and facilities that help the user to use and become more productive with MATLAB functions and files. Many of these tools are graphical user interfaces. It includes, the MATLAB desktop and Command Window, an editor and debugger, a code analyser, and browsers for viewing help, the workspace, and folders.

**Mathematical Function Library** This library is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms. The Language The MATLAB language is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both “programming in the small” to rapidly create quick programs you do not intend to reuse. You can also do “programming in the large” to create complex application programs intended for reuse.

**Graphics** MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualisation, image processing,
animation, and presentation graphics. It also includes low-level functions that allow the user to fully customise the appearance of graphics as well as to build complete graphical user interfaces on MATLAB applications.

**External Interfaces** The external interfaces library allows the user to write C/C++ and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB, for calling MATLAB as a computational engine, and for reading and writing MAT-files.

### 2.2.2 Simulink

In addition of this framework there is a modelling tool called Simulink that enables the developer to pose a question about a system, model the system, and see what happens.

With Simulink, one can easily build models from scratch, or modify existing models. Simulink supports linear and nonlinear systems, modelled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate i.e. having different parts that are sampled or updated at different rates [31].

Using Simulink, the developer can look into more realistic nonlinear models that describe real-world phenomena. Simulink let the user use his or her computer like a proper laboratory for modelling and analysing systems that would not be possible or practical otherwise.

Probably the most powerful and useful component of Simulink is the graphical user interface (GUI) for building models as block diagrams. Such GUI allows the developer to draw models as he or she would with pencil and paper. Simulink also includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. Furthermore if these blocks do not meet the developer’s needs, new blocks can also be created. The interactive graphical environment simplifies the modelling process, eliminating the need to formulate differential and difference equations in a language or program. Models are hierarchical so that one can view the system at a high level, then double-click blocks to see increasing levels of model detail. This approach provides insight into how a model is organised and how its parts interact.

A Simulink block (Figure 2.5) consists of a set of inputs, a set of states, and a set
of outputs, where the outputs are a function of the simulation time, the inputs, and the states. The following equations express the mathematical relationships between the inputs, outputs, states, and simulation time

\[ y = f_0(t, x, u) \quad (Outputs); \]
\[ x = f_d(t, x, u) \quad (Derivatives); \]
\[ x_{d_{k+1}} = f_u(t, x_c, x_{d_k}, u) \quad (Update) \]

where \( x = [x_c; x_d] \).

After the developer defines a model, it can be simulated, using a choice of mathematical integration methods, either from the Simulink menus or by entering commands in the MATLAB Command Window. Using scopes and other display blocks, the user can see the simulation results while the simulation runs. The user can then change many parameters and see what happens for “what if” exploration.

The simulation results can be put in the MATLAB workspace for postprocessing and visualisation. Because MATLAB and Simulink are integrated, the developer can simulate, analyse, and revise the models in either environment at any point. In fact Simulink software is tightly integrated with the MATLAB environment. It requires MATLAB to run, depending on it to define and evaluate model and block parameters. Simulink can also utilise many MATLAB features. For example, Simulink can use the MATLAB environment to define model inputs, store model outputs for analysis and visualisation or perform functions within a model, through integrated calls to MATLAB operators and functions.

An important feature of the Simulink tool are the so called S-Functions (system-functions). Such functions provide a powerful mechanism for extending the capabilities
of the Simulink environment. An S-function is a computer language description of a Simulink block written in MATLAB, C, C++, or Fortran. C, C++ and Fortran S-functions are compiled as MEX (MATLAB executable) files using the `mex` utility. As with other MEX files, S-functions are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute [32].

S-functions use a special calling syntax called the S-function API that enables the developer to interact with the Simulink engine. S-functions follow a general form and can compose continuous, discrete, and hybrid systems. The developer can implement an algorithm in an S-function and use the S-Function block to add it to a Simulink model.

Now let us look at the general simulation stages of a Simulink model. First comes the initialisation phase. In this phase, the Simulink engine incorporates library blocks into the model, propagates signal widths, data types, and sample times, evaluates block parameters, determines block execution order, and allocates memory. The engine then enters a simulation loop, where each iteration through the loop is referred to as a simulation step. During each simulation step, the engine executes each block in the model in the order determined during initialisation. For each block, the engine invokes functions that compute the block states, derivatives, and outputs for the current sample time.

More in details, while initialising and simulating the model, the engine always makes the required calls to `mdlInitializeSizes` and `mdlInitializeSampleTime` functions to set up the fundamental attributes of the S-function, including input and output ports, S-function dialogue parameters, work vectors, sample times, etc, then the engine calls additional methods, as needed, to complete the S-function initialisation.

After initialisation, the Simulink engine executes the simulation loop in which are called the `mdlOutputs` and the `mdlDerivatives` function in the case the simulation has continuous states, otherwise `mdlOutputs` and `mdlZeroCrossing` functions are called, in order to let the solvers compute the states for the S-function. If the simulation loop is interrupted, either manually or when an error occurs, the engine jumps directly to the `mdlTerminate` method. If the simulation was manually halted, the engine first completes the current time step before invoking `mdlTerminate`. 
An other important Simulink block used in the development of the project is the Embedded MATLAB Function block. It allows you to add MATLAB functions to Simulink models for deployment to embedded processors. This capability is useful for coding algorithms that are better stated in the textual language of the MATLAB software than in the graphical language of the Simulink product. The Embedded MATLAB Function block is composed by source code that follows the syntactical rules of the MATLAB language.

2.2.3 Real Time Workshop and Target Language Compiler

In order to translate into a programming language such as C or C++, the model constructed with Simulink using the aforementioned features there is a framework embedded in MATLAB that is capable to do this. It is called Real Time Workshop and its technology generates C or C++ source code and executables for algorithms that the developer models graphically in the Simulink environment or programmatically with the Embedded MATLAB language subset [33]. One can generate code for any Simulink blocks and MATLAB functions that are useful for real-time or embedded applications.

This code generator’s core engine is the so called TLC (Target Language Compiler). It enables the user to customise generated code, to produce platform-specific code, or to incorporate user’s algorithmic changes for performance, code size, or compatibility with existing methods that the user prefer to maintain.

The Target Language Compiler provides a set of ready-to-use TLC files for generating ANSI® C or C++ code. The user can view the TLC files and make minor, or extensive, changes to them. This open environment gives the user great flexibility when it comes to customising the generated code.

The Target Language Compiler (TLC) is designed for one purpose, to convert the model description file model.rtw, created by the Real Time Workshop from the graphic model, into target-specific code or text. The Target Language Compiler transforms an intermediate form of a Simulink block diagram, called model.rtw, into C or C++ code. The model.rtw file contains a “compiled” representation of the model describing the execution semantics of the block diagram in a very high-level language.

57
After reading the `model.rtw` file, the Target Language Compiler generates its code based on target files, which specify particular code for each block, and model-wide files, which specify the overall code style.

In Figure 2.6 it is shown how the Target Language Compiler works with its target files and the Real-Time Workshop code generator output to produce C/C++ code.

![Figure 2.6: Target Language Compiler code generation process.](image)

The following list defines some of the structures created in the source C/C++ code:

- **Block states structures**: The continuous states structure (`model_X`) contains the continuous state information for any blocks in the model that have continuous states. Discrete states are stored in a data structure called the DWork vector (`model_DWork`).
• Block parameters structure \((model_P)\): The parameters structure contains all block parameters that can be changed during execution.

• External inputs structure \((model_U)\): The external inputs structure consists of all Import block signals.

• External outputs structure \((model_Y)\): The external outputs structure consists of all Outport blocks.

Besides the data structures, functions are generated by the Real-Time Workshop too. Some of the most important of these functions are listed below:

\textbf{model\_initialize}: Performs all model initialisation and should be called once before the starting of the execution of the model.

\textbf{model\_step}: Contains the output and update code for all blocks in the model.

\textbf{model\_terminate}: This contains all model shutdown code and should be called as part of system shutdown.
2.3 Qt graphical libraries

In this section we will focus our attention on the graphical libraries for the C++ programming language used to implement the source code of the project. The choice was to use the Qt framework as it is widely adopted in the computer graphics software development world, it is very portable since it is based on the C++ language, it has a visual designer toolkit easy to use and a complete set of classes that covers all the most important feature of computer graphics. It runs on all major platforms and has extensive internationalisation support; it has also non-GUI features like SQL database access, XML parsing, thread management, network support, and a unified cross-platform API for file handling.

2.3.1 Graphics system

Qt’s 2D graphics engine is based on the QPainter class. QPainter can draw geometric shapes (points, lines, rectangles, ellipses, arcs, chords, pie segments, polygons and Bézier curves), as well as pixmaps, images, and text. Furthermore, QPainter supports advanced features such as antialiasing (for text and shape edges), alpha blending, gradient filling, and vector paths. QPainter also supports linear transformations, such as translation, rotation, shearing, and scaling [12].

QPainter can be used to draw on a “paint device”, such as a QWidget, a QPixmap or a QImage. By reimplementing QWidget::paintEvent(), one can create custom widgets and control completely their appearance as it was done during the developing of this project.

To start painting to a paint device (typically a widget), one has simply to create a QPainter and pass a pointer to the device. For example:

```cpp
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    ...
}
```
The developer can draw various shapes using QPainter’s draw...() functions. A list of the most important ones follows:

drawPoint(), drawLine(), drawPolyline(), drawPoints(), drawLines(), drawPolygon(), drawRect(), drawEllipse(), drawArc(), drawPie(), drawText(), drawPixmap() and drawPath(); most of them were used in the simulator program.

The QPainter object has several properties, the most important are:

**The pen** is used for drawing lines and shape outlines. It consists of a color, a width, a line style, a cap style, and a join style.

**The brush** is the pattern used for filling geometric shapes. It normally consists of a color and a style, but it can also be a texture (a pixmap that is repeated infinitely) or a gradient.

**The font** is used for drawing text. A font has many attributes, including a family and a point size.

For example, to draw an ellipse like in Figure 2.7 one can write the following piece of code:

```cpp
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

The setRenderHint() call enables antialiasing, telling QPainter to use different color intensities on the edges to reduce the visual distortion that normally occurs when the edges of a shape are converted into pixels. The result is smoother edges on platforms and devices that support this feature.

Here’s the code to draw the cubic Bézier curve shown in Figure 2.8:

```cpp
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
```
Figure 2.7: Example of a drawn ellipse.

```
QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);
painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);
```

The QPainterPath class can specify arbitrary vector shapes by connecting basic graphical elements together: straight lines, ellipses, polygons, arcs, Bézier curves, and other painter paths. Painter paths are a very powerful drawing primitive in the sense that using them, any shape or combination of shapes can be expressed as a painter path. A path specifies an outline, and the area described by the outline can be filled using a brush. In the example in Figure 2.8, it was not set a brush, so only the outline is drawn.

Figure 2.8: A Bézier curve drawn with the QPainterPath class.
2.3.2 Coordinate system

Now we can take a closer look at how the coordinate system can be changed using QPainter’s viewport, window, and world transform.

With QPainter’s default coordinate system, the point (0, 0) is located at the top-left corner of the paint device, x-coordinates increase rightward, and y-coordinates increase downward. Each pixel occupies an area of size $1 \times 1$ in the default coordinate system.

The viewport and the window are tightly bound. The viewport is an arbitrary rectangle specified in physical coordinates. The window specifies the same rectangle, but in logical coordinates. When we do the painting, we specify points in logical coordinates, and those coordinates are converted into physical coordinates in a linear algebraic manner, based on the current window–viewport settings.

By default, the viewport and the window are set to the device’s rectangle. For example, if the device is a $320 \times 200$ widget, both the viewport and the window are the same $320 \times 200$ rectangle with its top-left corner at position (0, 0). In this case, the logical and physical coordinate systems are the same. The window–viewport mechanism is useful to make the drawing code independent of the size or resolution of the paint device. For example, if we want the logical coordinates to extend from (-50, -50) to (+50, +50), with (0, 0) in the middle, we can set the window as follows:

```cpp
painter.setWindow(-50, -50, 100, 100);
```

The (-50, -50) pair specifies the origin, and the (100, 100) pair specifies the width and height. This means that the logical coordinates (-50, -50) now correspond to the physical coordinates (0, 0), and the logical coordinates (+50, +50) correspond to the physical coordinates (320, 200). This is illustrated in Figure 2.9.

Let now see at the world transform. The world transform is a transformation matrix that is applied in addition to the window–viewport conversion. It allows us to translate, scale, rotate, or shear the items we are drawing. For example, if we wanted to draw text at a $45^\circ$ angle, we would use this code:

```cpp
QTransform transform;
transform.rotate(+45.0);
painter.setWorldTransform(transform);
```
If we specify multiple transformations, they are applied in the order in which they are given. For example, if we want to use the point \((50, 50)\) as the rotation’s pivot point, we can do so by translating the window by \((+50, +50)\), performing the rotation, and then translating the window back to its original position:

```cpp
QTransform transform;
transform.translate(+50.0, +50.0);
transform.rotate(+45.0);
transform.translate(-50.0, -50.0);
painter.setWorldTransform(transform);
painter.drawText(pos, tr("Foo"));
```

### 2.3.3 Signals and slots

Signals and slots are one of the central features of Qt; they are used for communication between objects.

A signal is emitted when a particular event occurs. Qt’s widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt’s widgets have many pre-defined slots, but it is common practice to subclass widgets and add one’s own slots so that one can handle the signals that he or she is interested in.

The signals and slots mechanism is type safe in fact the signature of a signal must
match the signature of the receiving slot. Signals and slots are loosely coupled because a class which emits a signal neither knows nor cares which slots receive the signal. Qt’s signals and slots mechanism ensures that if one connects a signal to a slot, the slot will be called with the signal’s parameters at the right time. All classes that inherit from QObject or one of its subclasses can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to other objects and the emitter object does not know or care whether anything is receiving the signals it emits. Slots can be used for receiving signals, but they are also normal member functions furthermore as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it; this ensures that truly independent components can be created with Qt. A single slot can be connected to as many signals as one wants, and a signal can be connected to as many slots as one needs. It is even possible to connect a signal directly to another signal and in this case the second signal will emit immediately whenever the first is emitted.
A code example of the signals and slots mechanism is the following: first we declare a class that inherits from QObject,

```cpp
#include <QObject>
class MyClass : public QObject {
    ... 
    public:
    ... 
    public slots:
        void setValue(int value);
    signals:
        void valueChanged(int newValue);
    private:
        int x;
};
```

Here there is a possible implementation of the `MyClass::setValue()` slot:

```cpp
void MyClass::setValue(int value) {
    ... 
    x = value;
    emit valueChanged(value);
}
```

The `emit` line emits the signal `valueChanged()` from the object, with the new value as argument.

```cpp
MyClass a, b;
QObject::connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
```

Two `MyClass` objects were created and the first object’s `valueChanged()` signal was connected to the second object’s `setValue()` slot using `QObject::connect()`, so that when `a.setValue(...)` is called the `valueChanged()` signal is emitted and the `setValue()` slot function is called for the object `b`.  

66
2.3.4 Event system

Events are generated by the window system or by Qt itself in response to various occurrences. For example when the user presses or releases a key or mouse button, a key or mouse event is generated; when a window is shown for the first time, a paint event is generated to tell the newly visible window that it needs to draw itself. Most events are generated in response to user actions, but some, such as timer events, are generated independently by the system. Qt widgets emit signals when something significant occurs. Events become useful when we write our own custom widgets or when we want to modify the behaviour of existing Qt widgets [12]. Events should not be confused with signals. As a rule, signals are useful when using a widget, whereas events are useful when implementing a widget. In Qt, an event is an instance of a QEvent subclass. Qt handles more than a hundred types of events, each identified by an enum value.

Many event types require more information than can be stored in a plain QEvent object; for example, mouse press events need to store which mouse button triggered the event as well as where the mouse pointer was positioned when the event occurred. This additional information is stored in dedicated QEvent subclasses, such as QMouseEvent.

Another common type of event is the timer event. While most other event types occur as a result of a user action, timer events allow applications to perform processing at regular time intervals. Timer events can be used to implement animations, or simply to refresh the display.

When an event occurs, Qt creates an event object to represent it by constructing an instance of the appropriate QEvent subclass, and delivers it to a particular instance of QObject (or one of its subclasses) by calling its event() function. This function does not handle the event itself; based on the type of event delivered, it calls an event handler for that specific type of event, and sends a response based on whether the event was accepted or ignored.

The normal way for an event to be delivered is by calling a virtual function. For example, QPaintEvent is delivered by calling QWidget::paintEvent(). This virtual function is responsible for reacting appropriately, normally by repainting the widget. Hence by reimplementing such function in our QWidget subclass, we can access and control
the QPaintEvent event delivered; this can be done for each type of event handler function. If one wants to replace the base event handler function of a class, he or she must re-implement everything. However, if one only wants to extend the base class's functionality, then he or she has to implement what he or she wants and call the base class to obtain the default behaviour for any cases one do not want to handle.

Sometimes an object needs to look at, and possibly intercept, the events that are delivered to another object. The QObject::installEventFilter() function enables this by setting up an event filter, causing a nominated filter object to receive the events for a target object in its QObject::eventFilter() function. An event filter gets to process events before the target object does, allowing it to inspect and discard the events as required. To remove an existing event filter one can use the QObject::removeEventFilter() function.

When the filter object’s eventFilter() implementation is called, it can accept or reject the event, and allow or deny further processing of the event. If all the event filters allow further processing of an event (by each returning false), the event is sent to the target object itself. If one of them stops processing (by returning true), the target and any later event filters do not get to see the event at all. Let us now look at an example:

```cpp
bool MyFilterObject::eventFilter(QObject *object, QEvent *event) {
    if (object == target && event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            ... 
            return true;
        } else
            return false;
    }
    return false;
}
```

68
The above code shows a way to intercept Tab key press events sent to a particular target widget. In this case, the filter handles the relevant events and returns true to stop them from being processed any further. All other events are ignored, and the filter returns false to allow them to be sent on to the target widget, via any other, if any, event filters that are installed on it.

One may want to make an application that is capable of creating and sending its own events. This can be done in exactly the same ways as Qt’s own event loop does by constructing suitable event objects and sending them with `QCoreApplication::sendEvent()` or `QCoreApplication::postEvent()`.

`sendEvent()` processes the event immediately. When it returns, the event filters and/or the object itself have already processed the event. For many event classes there is a function called `isAccepted()` that tells the function whether the event was accepted or rejected by the last handler that was called.

`postEvent()` posts the event on a queue for later dispatch. The next time Qt’s main event loop runs, it dispatches all posted events, with some optimisation. For example, if there are several resize events, they are compressed into one. The same applies to paint events: `QWidget::update()` calls `postEvent()`, in order to eliminate flickering and increase speed by avoiding multiple repaints.

### 2.3.5 Threads

Just as multitasking operating systems can do more than one thing concurrently by running more than a single process, a process can do the same by running more than a single thread. Each thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently. One thread can run the GUI, while a second thread does some I/O, while a third one performs calculations [13].

Conventional GUI applications have one thread of execution and perform one operation at a time. If the user invokes a time-consuming operation from the user interface,
the interface typically freezes while the operation is in progress. To avoid this annoyance, one may use multithreading as a solution.

In a multithreaded application, the GUI runs in its own thread and additional processing, like the aforementioned time-consuming task, takes place in one or more other threads. This results in applications that have responsive GUIs even during intensive processing. When running on a single processor, multithreaded applications may run slower than a single-threaded equivalent due to the overhead of having multiple threads. But on multiprocessor systems, which are becoming increasingly common, multithreaded applications can execute several threads simultaneously on different processors, resulting in better overall performance.

The most used functions in Qt to program with threads are **QThread**, **QMutex**, **QSemaphore** and **QWaitCondition**. Let us have a look at each of them.

A **QThread** instance represents a thread and provides the means to start() a thread, which will then execute the reimplementation of **QThread::run()**. The run() implementation is for a thread what the main() entry point is for the application. All code executed in a call stack that starts in the run() function is executed by the new thread, and the thread finishes when the function returns. **QThread** emits signals to indicate that the thread started or finished executing [29]. To create a thread, one has to subclass **QThread** and re-implement its run() function. For example:

```cpp
class MyThread : public QThread {
    ...
    protected: void run();
};
void MyThread::run()
{ ... }
```

Then, one have to create an instance of the thread object and call **QThread::start()**. The function will return immediately and the main thread will continue. The code that appears in the run() reimplementation will then be executed in a separate thread. In GUI applications, the main thread (the thread that executes main()) is also called the GUI thread because it is the only thread that is allowed to perform GUI-related
operations.

A common requirement for multithreaded applications is that of synchronising several threads. To accomplish this task the QMutex, QSemaphore and QWaitCondition classes come to our aid. While the main idea with threads is that they should be as concurrent as possible, there are points where threads must stop and wait for other threads. For example, if two threads try to access the same global variable simultaneously, the results are usually undefined.

QMutex provides a mutually exclusive lock, or mutex. At most one thread can hold the mutex at any time. The class provides a lock() function that locks the mutex. If the mutex is unlocked, the current thread seizes it immediately and locks it; otherwise, the current thread is blocked until the thread that holds the mutex unlocks it. Otherwise, when the call to lock() returns, the current thread holds the mutex until it calls unlock(). The QMutex class also provides a tryLock() function that returns immediately if the mutex is already locked.

QSemaphore is a generalisation of QMutex that protects a certain number of identical resources. In contrast, a mutex protects exactly one resource. The following two code snippets show the correspondence between QSemaphore and QMutex:

```cpp
QSemaphore semaphore(1);  // QMutex mutex;
semaphore.acquire();       // mutex.lock();
semaphore.release();       // mutex.unlock();
```

By passing 1 to the constructor, we tell the semaphore that it controls a single resource. The advantage of using a semaphore is that we can create QSemaphore objects passing to the constructor numbers greater than 1 and then call acquire() multiple times to acquire many resources.

QWaitCondition allows a thread to wake up other threads when some condition has been met. One or many threads can block waiting for a QWaitCondition to set a condition by calling wakeOne() or wakeAll().

71
3 MODEL AND IMPLEMENTATION OF THE ROBOT SIMULATOR

In this chapter are shown the methods that were used to implement the realisation of the project. The following sections illustrate the implementations of the physical model using MATLAB, Simulink and the Real Time Workshop and the implementation of the C++ code using the Qt libraries.

Let us now overview the various step that led to the achievement of the project. The work-flow of the project started from the analysis of the specifications of the problem to be solved. The issue was to test virtually, on a graphic interface, the behaviour of the dynamics of a marine robot in a simulated marine environment.

The work-flow evolved into the modelling of the physical behaviour of the movements of the robot in the water. The baselines to simulate the marine environment were mainly the following: initially the speed and the acceleration of the robot had to be simulated having as inputs the speed of the two propellers; the second task to do was to simulate the angular acceleration and the angular speed of the robot considering the angle of the rudder and the difference of speed of the two propellers that make the robot turn. These two phases of the work were modelled on the MATLAB framework using the Simulink toolbox. After the creation of the model, with the help of the Real Time Workshop, it was converted into C++ code and this code was integrated with the graphic interface hand-made code.

From this step the work-flow passed through the phase of the implementation of the graphical interface to show the simulation of the physical model built in the previous
step. The source code to realise the graphic interface is composed by two different parts: a server and a client side. The job of the client side is essentially to read the input data (i.e. the rotation speeds of the two propellers and the angle of the rudder) from the robot via a CAN wire or from a socket connected to the robot’s control software, to call the automatically made C++ functions to simulate the model and finally to send the results of these functions to the server side. These series of operations happen on average every 50 milliseconds. The server side creates the graphic interface using the Qt libraries and updates the status and the data of the robot every time it receives data from the client.

The final step consisted in the testing the simulator interface connecting the program previously realised directly with the robot or connecting such program with the software that controls the robot.

### 3.1 Simulator overview

Let us now see the simulator’s structure and capabilities. First of all the simulator can be in two different configurations:

**hybrid** let the simulator use the real data that come from the robot; in this way both the software and the hardware parts of the robot and their correct interaction can be tested.

**virtual** all the robot’s data are generated virtually by software in order to test the correctness of the software that runs usually on the robot and to perform simulations that could be too damage prone for the hardware if tested in the hybrid way.

The simulator is composed mainly by two parts. The physical model of the dynamics of the robot and of the marine environment (i.e. sea currents and winds), built with Simulink, and the graphical interface that mediates between the Manager process and the code generated by the Real Time Workshop from the model (Figure 3.1).

The Manager process, as written in section 2.1,
reads the data from the CAN bus, if the simulator is in the hybrid configuration hence the Manager process runs on the robot’s microprocessor, or reads the data from sockets when it runs on the same PC of the simulator when the latter is in the virtual configuration;

- receives the mission commands (e.g. waypoints) or robot’s data (e.g. position, speed, propellers speeds, rudder’s angle) from the simulator;

- dispatches such commands to the different processes.

As said before the simulator has two types of modalities, the hybrid and the virtual ones. In Figure 3.2 there is the representation of the hybrid configuration:

**Radio** manages the communication via radio.
**Manager** manages the commands that come via radio and CAN bus, besides all the robot’s data.

**Navigation** is divided in Global Path Planning and Path Following and creates the messages to be sent to the motors via the CAN bus.

**OA** Obstacle Avoidance module that utilises the data coming from the sensors (devices).

**GUI** is the graphical interface of the simulator, from which there is the possibility to define the position of the robot, the navigation goal point(s), the map, the possible obstacles etc., communicates via radio to the robot. The communication with the robot is performed via an appropriate software module (Wireless Device) that communicates via a TCP socket because the radio device is installed in a different place respect to the one where the robot is.
**Devices** in this configuration the robot’s devices (laser and sonar) can be simulated or real, in the first case the communication is done via Wi-Fi and therefore with a TCP Socket.

**Motor** in this configuration the motors can be simulated or real, in the first case the communication medium is the CAN bus.

**Dynamics** simulates the dynamics of the robot and communicates via CAN bus.

The virtual configuration is depicted in Figure 3.3, to be noticed that all the communications happen via socket since all the processes are executed on the same computer:

![Diagram](image)

Figure 3.3: The virtual configuration.

**GUI** is the graphical interface of the simulator.

**Devices** simulates the robot’s laser and sonar.
Motor simulates the robot’s motors, this simulation can be performed in a simple way, i.e. redirecting only the commands that come from the Navigation module or otherwise simulating the dynamics of the motors.

Dynamics simulates the dynamics of the robot.

In summary, the simulator has the following properties:

- capability of testing the robot in various conditions, normal behaviour or simulating hardware failures;
- hybrid or virtual simulations like explained above;
- capability of testing the robot’s dynamics in a changing marine environment;
- capability of testing the obstacle avoidance and the path following algorithms.

In the next sections of this chapter we will see more closely the implementation and the properties of the aforementioned components of the simulator.

3.2 The Simulink models

3.2.1 Robot dynamics

The first task to do was to model the physical behaviour of the marine robot in a water environment. In our case the physical phenomenon to be described in a mathematical schema, is the movement of the robot in open waters.

The inputs of the model are the speed of rotation of the two propellers and the angle of the rudder. As output the model generates the speed, the angular speed and the angular acceleration of the robot and the robot’s speed computed by the flowmeter sensor. The main block model (Figure 3.4) is divided in two inner blocks called motor_dynamics and calcU_R. The block that computes the outputs is calcU_R and it takes as inputs the variables nR and nL that represent respectively, the speed of rotation of the right propeller and the speed of rotation of the the left propeller in rounds per minute,
\textit{deltaRudder} that represents angle of the rudder in radians and \textit{r\_in}, the angular velocity in degrees over seconds at the \textit{(i-1)}\textsuperscript{th} instant, i.e. the instant before the block computations are performed. The input variables \textit{nR}, \textit{nL} and \textit{deltaRudder} can be obtained directly from the main block inputs if the \textit{dynamic\_motors} variable is set to false or otherwise as the outputs of the other inner block \textit{motors\_dynamics}. In the \textit{motors\_dynamics} block is computed the real dynamics of the motors, that when input with some values for \textit{nR}, \textit{nL} and \textit{deltaRudder} they output different values due to their hardware implementation.

Let us now see the \textit{calcU\_R} inner block as in Figure 3.5. The input variables \textit{nR} and \textit{nL} are taken as inputs for the S-function \textit{modelloSurge-MEX} that calculates the speed of the robot using the following formula as in [14]:

![Simulink model](image.png)

Figure 3.4: The whole Simulink model.
\[(m + m_a) \frac{du}{dt} = X_u u + X_{uu} |u| + X_{uuu} u |u|^2 + X_{uuuu} u |u|^3 +
+ K_{nn} (nL |nL| + nR |nR|) + X_{u2\delta} u^2 \delta^2\]

where \(u\) is the surge speed, \(m\) and \(m_a\) are the mass and the added mass, \(X_u, X_{uu}, X_{uuu}, X_{uuuu}\) are the surge drag coefficients, \(K_{nn}\) is the thrust coefficient \(X_{u2\delta}\) is the braking term induced by the rudder, \(nL\) and \(nR\) are the left/right propellers speeds in rpm and \(\delta\) is the rudder angle in radians.

Figure 3.5: The Simulink model for the robot’s speeds simulation.

The output of the S-function is the velocity \(u\) of the robot. It is used together with \(nR, nL, \text{deltaRudder}\) and \(r\text{\_in}\) as input for the Embedded MATLAB function \(\text{calc\_rDot}\). As we told before \(nR\) and \(nL\) are the speed of rotation of the right and the left propeller respectively, \(\text{deltaRudder}\) is the angle of the rudder and \(r\text{\_in}\) is the angular velocity at the \((i-1)^{th}\) instant, i.e. the instant before the function \(\text{calc\_rDot}\) is called. The Embedded MATLAB function calculates the angular acceleration \(rDot\) using the following formula:

\[I_z \cdot rDot = (K_{rr} \cdot r + K_r \cdot |r| \cdot r + K_{ud} \cdot u^2 \cdot \text{deltaRudder})\]

Where \(I_z\) is the moment of inertia and the added mass, \(K_r\) and \(K_{rr}\) are the linear
and quadratic drag coefficients, $K_{ud}$ a corrective factor because of the rudder effect and $\delta_{\text{Rudder}}$ is the angle of the rudder expressed in radians.

After $r_{\text{Dot}}$ is calculated its value is integrated in time to retrieve the angular velocity $r_{\text{out}}$ at the $i^{th}$ instant of time. The outputs of this part of the model are $r_{\text{Dot}}$ and $r_{\text{out}}$. The latter of these values is reintroduced in the model as the input value $r_{\text{in}}$ at the following instant of calculation. In addition of these elements, in the Simulink schema there is also a section to emulate the function of the flowmeter. The flowmeter is a device for measuring velocity of a flow, in our case the water that passes under the robot. With this tool we can obtain the real velocity of the robot respect to water. The flowmeter updates the velocity with a greater frequency as the robot moves faster and updates it with lower frequency when the robot moves slower. This behaviour is computed with an Embedded MATLAB function, that takes as input arguments the clock time intervals, to calculate when to update the speed of the flowmeter, and the velocity $u$ of the robot that was output from the S-function modelSurge-MEX. The output of this function is obviously the velocity measured by the flowmeter ($u_{\text{flow}}$) and is output with a frequency in a directly proportional way respect to the speed.

All these functions were converted into C++ functions using the MATLAB's Real Time Workshop tool.

### 3.2.2 Sea currents and winds models

As we said in chapter 1.3.2 sea currents can be simulated as a random walk, that is time integration of white noise. This kind of simulation can be extended to the forces that act on the robot resulting from the summation of the forces of the sea currents and those of the winds. In Figure 3.6 there is the Simulink model for such simulation; as one can see there are two blocks that generate a Gaussian white noise, one is to compute the intensity of the forces and the other one the direction, in fact both of the noises are integrated with limits; the first one has as upper limit 4 and lower limit 0, they are the values in knots that the sea currents, resulting from the interaction with the winds, can have; the second integrator has for limits 0 and 360 and these are the
values in degrees for the angle of the direction of the simulated forces.

Figure 3.6: Simulink model for the sea currents and winds forces behaviour in time.

The winds, as the sea currents, can be simulated as a random walk. The Simulink model for such simulation contains two blocks that generate a Gaussian white noise, one is to compute the intensity of the wind and the other one its direction, then both of the noises are integrated with limits; the first one has as upper limit 30 and lower limit 0, they are the values in knots that the winds can have; the second integrator has for limits 0 and 360 and these are the values in degrees for the angle of the direction of the winds.

As for the other Simulink model, also these two are translated by the Real Time Workshop into C++ code, so that they can be used by the code of the simulator.
3.3 Source code implementation

As we previously mentioned the source code is divided into two different logical parts. There is the so called client, that reads data directly from the robot via a CAN bus if the simulator is in its hybrid configuration or from a socket if the configuration is virtual, then it computes the calculations described in the Simulink model using the functions generated by the Real Time Workshop and sends the outputs to the other program (the so called server) via socket. The server receives the data from the client and updates the robot state and therefore the state of the graphical interface. In the server part there is also implemented the simulation of the two robot sensors, the laser and the sonar; besides this there is the implementation of the simulation of the water currents, winds and waves of the virtual environment where the robot’s movements are supposed to be simulated.

Let us now analyse in details these two parts.

3.3.1 Client side source code implementation

Here we discuss how the client program works. The program takes as arguments the host which it has to create a socket with and the port number upon which it has to create the socket. At the beginning of the code is called the \textit{model\_initialize} function, which was generated by the Real Time Workshop to perform the initialisation of all the data structures needed to execute the code generated from the Simulink model. After this all the variables useful for the computation are initialised (i.e. the velocity $u$, the velocity $u$ of the flowmeter, the speed of rotation of the two propellers, the angle of the rudder, the angular acceleration $\dot{r}$ and the angular velocity $r$); the socket \textit{serverSocket} is created with the desired host at the desired port, to communicate with the server side and it is connected to the server. Then if the simulator configuration is the hybrid one, the connection with the CAN bus is made otherwise a socket (\textit{motorSocket}) to communicate with the Navigation process is created; the $r\_in$ parameter of the Simulink model is initialised to zero (see the model description at chapter 3.2), the position of the robot and its angles of rotation are read from the socket connected to the server side of the simulator and a first series of messages, containing the data just read from the server
socket, the current speed of the robot and its current angular speed, is sent onto the CAN wire or onto an other socket (dynamicSocket) previously created that is connected with the Manager process.

Now is set the variable dynamic_motors of the Simulink model that controls which type of dynamics for the motors will be executed.

After this, if the simulator is in the virtual configuration, a new thread is created responsible for reading the data from the motorSocket socket, otherwise these data are read from the CAN wire, in both configurations the reading is performed inside an infinite loop.

Such data are, the angle \( \delta \) of the rudder, the speed of rotation \( nL \) of the left pro-
peller and the speed of rotation $nR$ of the right propeller. Afterwards these read values are passed as arguments of the inputs of the model, literally into the data structure $model_U$. Now we can call the function $model_step$ to compute a step of the calculation of the model and read the output values of the model from the data structure $model_Y$. The output values are the velocity $uMod$ of the robot, the velocity $uFlow$ of the flowmeter, the angular acceleration $rDot$ and the angular velocity $r_{out}$. To reckon the angular speed at the next time step we should now set the variable $r_{in}$ of the model with the output value of the angular speed at the previous step, that is $r_{out}$. We, now, send all these values onto the socket to the server side ($serverSocket$) that answers with the values of the robot’s position and angles of rotation that are sent among the current speed of the robot and its current angular speed onto the CAN wire or onto the $dynamicSocket$ socket, and the loop block starts again its computation.

When the program is terminated from an external event, for example the closure of a socket, it exits the infinite loop and calls the $model_terminate$ function to perform the shut-down of the model’s code.

### 3.3.2 Server side source code implementation

Let us begin from the main function. This is the starting point of the execution of the program. Here the main Qt widget is created and begins the initialisation of all the variables thus the entire graphics is set up. Soon after this part seven threads are created: one thread is the responsible of the calculations needed for the simulation of the laser sensor, the second thread is made to create the simulation of the sonar sensor, other two threads are in charge of the simulation of the marine currents and the winds respectively, one is responsible for the communication via socket with the client and for the update of the robot’s data, the last two are made to communicate via socket with other software that manages the control of the robot. Before analysing the threads and functions that are part of the program we will see the most important data structures of it.

#### 3.3.2.1 Data structures

In this paragraph are listed all the major data structures used in the code. Four different struct exist in the program.
The sea currents and winds are implemented using a `struct` called `current`, whose fields are the following: `posX` and `posY` to specify the position of the arrow icon that represent the current or the wind along the X and Y axes respect to the origin of the map; `intensity` of the current or wind expressed in knots; `direction` of the arrow that simulates the direction of the current or wind in degrees from North; `type` of the object, i.e. “W” if it is a wind arrow or “C” if it is a current arrow.

Boats and shallows are implemented via a `struct` as well, it is called `boat` and it is so composed: `posX` and `posY` to determine the position of the boat or shallow respect to the origin of the map; `direction` of the boat fore or of the semi-axis along the length of the ellipse that represents the shallow viewed from above, expressed in degrees from north; `length` of the boat or the ellipse in metres; `width` of the boat or the ellipse in metres; `height` of the boat in metres (only for the boats); `speed` of the boat in metres over seconds (only for the boats).

The `current` `struct` is used in the `currents` and `winds` arrays to memorise all the marine currents and winds values for the entire map. The `boat` `struct` objects instead are collected in the `boatArray` array to have the control of all the boats created by the user and in the `shallowsArray` array to identify all the shallows added by the user onto the map.

The other two important `structs` are `Robot` and `Wave`.

The first one is probably the most relevant data structure of the project since it holds all the properties of the robot that are updated in real-time. The following are the fields of this `struct`: `posX` and `posY` refer as usual to the position of the robot in the map expressed in pixels from the upper-left corner of the map; `width` and `length` of the robot in metres; `deltaRudder` is the angle between the current position of the robot’s rudder and its rest position measured in degrees; `u` is the surge speed of the robot measured in metres over seconds; `nL` and `nR` are respectively the number of rounds per minute of the robot’s left and right propeller; `r` and `rDot` are the robot’s angular velocity and angular acceleration; `psi` is the yaw angle of the robot in radians from north; `uFlow` is the surge velocity measured by the robot’s flowmeter; `pitch` and `roll` angles of the robot respect to the horizontal plane expressed in radians; `direction` of the robot while moving in degrees respect to North; `lat` and `lon` represent the latitude and the longitude.
of the current robot’s position.

Wave holds the characteristics of the sea wave at the robot’s position to simulate the interaction of our robotic boat with it. These are the variables of this object: amplitude of the wave in meters; length of the wave in meters; frequency of the wave in radians over seconds; direction of the wave respect to North measured in radians.

An other important component of the project is the array of integers \_arrLaser in which are stored all the distances measured by the laser for every quarter of degree with a range of 270 degrees; thus the array contains 1081 locations filled with a distance in millimetres that goes from 0 to 15000 that is the maximum distance to which the sensor can “see”.

There is a similar structure for the sonar sensor called sonarMatrix, as the name says it is a matrix formed by the sonar’s scanlines as columns and by the sonar’s bins of each scanline as rows. A scanline is the minimum amount of the angular scansion performed by the sonar during its rotation around the centre of the section of circle to be scanned, meanwhile a bin is the minimum section of the scanline that represents an area of the scanline.

Every item of the matrix is an integer and it has a value of 0, if in the location represented by that item there is no object, otherwise it has a positive value, in our implementation we have chosen 100.

The last of the most interesting data structures to be described is the pathArray array in which are stored all the points of the map that are crossed by the robot in its movements.

We are now going to see from close the details of the various threads that compose the program.

3.3.2.2 Graphics thread

As we said above, the main function firstly creates the main Qt widget that is a proper graphical window. Such window is designed in the file with .ui extension. This file is written in the XML format and describes the layout of the window and all its contained widgets. This window template is graphically divided in nine relevant parts:
• The background that is covered by the map of the area of work of the marine robot, in our case the sea and the coastline at Livorno and its surroundings, but it could be a map of any part of the world.

• The upper left side of the window is occupied by the Command panel (Figure 3.8). In here there are all the widgets that activate the commands that the simulator user can utilise.

![Command panel overview](Figure 3.8: Command panel overview.)

There are four checkboxes to show or hide the arrow icons for the marine currents and for the winds, to show or hide the path that the robot covered while moving and to show or hide the net that covers the map, useful to easily recognise the distances between objects on the map. As default all the checkboxes are checked, this means that all the icons and lines are shown onto the map. Below these checkboxes there is an other checkbox that if checked enables the user to set up manually the direction and the speed of the winds of the map. To set up these values there are two spinboxes, one for the speed and the other for the direction; as default the checkbox is not checked and the two spinboxes are not enabled.
There are also a checkbox and two spinboxes to manually set up the sea currents of the map; the default settings are the same as for the wind manual set up frame. Under this frame there are the buttons to perform the zoom in and zoom out actions.

Under these buttons there is another one that when pressed let the user set a point or a series of points onto the map where the robot will move to. When this button is pressed two couples of radio buttons are activated; the first couple is used to set up the value for a follow-line navigation or a waypoint navigation and the second one is for the choice of a flag for the moveTo message (see the last point of this list) that could be ADD_WP to add a new waypoint to the list or NEW_WP to clear the list of waypoints and start a new one.

On the right side of this panel there is a checkbox and two spinboxes to manually set up the waves independently by the wind intensity; the default settings are the same as for the wind and sea currents manual set up frame.

Again under, there are other two buttons, one let the user centre the map on the robot position (the default choice), meanwhile the second one has the function of letting the user move the map in the position he or she wants.

Just under the previous mentioned buttons there are other two of them; these when pressed have the function to switch the kind of map visualised in the simulator. If the left of these two buttons is pressed the white and black map is shown, where the white colour represents water and the black colour represents rocks or in general mainland (this option is the default one). If the right button is pressed the map becomes a satellite view of the same area.

In the bottom part of the panel there are two icons that if dragged on the map let the user add shallows or boats respectively onto the map.

- On the upper right side of the window there is a second panel the Properties one. Inside such panel there is a table with two columns and several rows. The left column shows the property name and the right column its value. The properties are those of the robot, in real time, it means that they are updated every time the graphics is refreshed. Here is the list of such properties:
- Position of the robot along the X axis measured in metres respect to the centre of the map.
- Position of the robot along the Y axis measured in metres respect to the centre of the map.
- The yaw angle (ψ) of the robot respect to North in degrees.
- The surge velocity of the robot respect to water (u) expressed in metres over seconds.
- The speed of rotation of the left propeller (nL) expressed in rounds per minute.
- The speed of rotation of the right propeller (nR) expressed in rounds per minute.
- The angular velocity of the robot (r) expressed in degrees over seconds.
- The angular acceleration of the robot (r) in degrees over squared seconds.
- The angle of the rudder (δ rudder) respect to its default position measured in degrees.
- The surge velocity of the robot measured by the flowmeter (u flowmeter) in metres over seconds.
- Latitude of the position of the robot in degrees.
- Longitude of the position of the robot in degrees.
- The pitch angle of the robot respect to horizon (θ) in degrees.
- The roll angle of the robot respect to horizon (ϕ) in degrees.
- The direction of movement of the robot respect to North measured in degrees.

• At the bottom of the window in the left corner there is the graph of the laser sensor (Figure 3.9). It is a portion of circle and the length of every rays of this circle are directly proportional to the distance computed by the laser from its origin point to an obstacle if there is any.
In the last corner left (i.e. the bottom right one) there is the graph of the sonar sensor (Figure 3.10). Also this one is a portion of circle and the objects that the sonar can “see” are painted in this graph onto its background colour.

At half of the window’s width at the bottom there is a graph that represents the sea wave at the position of the robot according to the current values of the wind or of the wave itself if it is manually set up. The wave has a sinusoidal form and moves horizontally, above it there is a red line that represents the rays of the laser viewed from aside that change their slope depending on where the starting point of the laser is located onto the wave. The wave is drawn using the `QPainterPath` object and its `cubicTo` function that enables the drawing of Bézier curves.

Other two parts are two little popup windows that appear when the shallow or the boat icons are dragged onto the map. In these windows there is a table with two
columns; in the left one there are the names of the characteristics of the shallow or boat and in the right one there are the values of these characteristics to be set by the user. These fields to be filled are represented as spinboxes. At the bottom of these windows there are the accept and the cancel buttons to perform respectively the creation of a new shallow or boat on the map, or its cancellation.

The following are the characteristics of the two windows.

- In the shallow creation popup window there is:
  * Direction of the semi-axis along the direction of the length of the ellipse that represents the shallow viewed from above, expressed in degrees from North.
  * The length of the ellipse in metres.
  * The width of the ellipse in metres.

- In the boat creation popup window there is:
  * Direction of the boat fore expressed in degrees from North.
  * The length of the boat in metres.
  * The width of the boat in metres.
  * The height of the boat in metres.
  * The speed of the boat in metres over seconds.

- Lastly there is an other popup window that is shown when the user wants to let the robot move to a certain point of the map or let the robot move along a certain line on the map. It contains all the parameters in order to create a *moveTo* message that will be sent to the Maneuver software block (see section 3.3.2.8). The message commands a follow-line navigation or a waypoint navigation. In the follow-line navigation the robot follows a path along a line between Waypoint 1 (WP1) and Waypoint 2 (WP2). In the waypoint navigation the robot follows a line between its current position and the received waypoint (WP2). The *moveTo* message presents two different modalities set by one dedicated flag field: ADD_WP and
NEW_WP. In the ADD_WP the received waypoint (or line) is added to a buffer in the Navigation process where all the waypoints to be reached are stored. In the NEW_WP modality the buffer is cleaned and the current waypoint (or line) is set to the one received.

The popup window contains the following data:

- WP1 x- and y-coordinates in metres.
- WP2 x- and y-coordinates in metres.
- A radio button to choose how to set up the navigation parameters (default values or user defined values).
- Cruise speed of the robot while reaching the waypoint in metres over seconds.
- Minimum speed of the robot while reaching the waypoint in metres over seconds.
- A radio button to indicate if the speeds are relative to the water or to the ground.
- The maximum commanded rotational speed to the robot in degrees over seconds.
- The tolerance radius within which the robot considers a waypoint reached in metres.
- The heading the robot has to have when reaching the waypoint in degrees.
- The tolerance in the commanded heading destination in degrees.

In the main function is called the init function that is responsible for the initialisation of all the variables and fields of the program. Here are loaded all the maps, are set all the zooming properties, a timer is initialised for the time involving functions, signals are connected to slots, other variables to initialise the state of the interface are set and the properties panel is filled for the first time. The most relevant initialisations are the connection of four slots to the timeout event triggered by the timer every 50 milliseconds. Such slots are updateRobotPosition, moveWave, moveBoat and calcTangentToWave.
The first one is used to update all the values regarding the robot positioning, heading, speeds, angles that are shown in the Properties panel and to insert in the pathArray array all the points of the map where the robot passed across; such array is doubled in capacity every time it is filled.

In the moveWave function are executed the following commands. First it is computed the distance $d_r$ that the robot covered over the wave since the last time this function was called: knowing the space covered by the robot in a step of the simulation time included the thrust of the sea currents $s\text{Covered}$ (see section 3.3.2.7 on how it is evaluated) and the angular difference between the direction of the robot and the direction of the wave $\Delta$, such distance is computed in this way:

$$d_r = s\text{Covered} \cdot \cos(\Delta)$$ (3.1)
secondly it’s computed the space $d_w$ that the wave covered in its direction since the last time this function was called, knowing its length $\lambda$ and its angular frequency $\omega$:

$$d_w = \frac{\lambda}{2\pi/\omega} = \frac{\lambda \cdot \omega}{2\pi}$$

the total distance $d$ that the robot covered onto the wave since the last time this function was called is simply

$$d = d_r + d_w \tag{3.2}$$

The next function called at every timeout event trigger is `moveBoat`. Here it is reckoned the distance in pixels covered, since the last time this function was called, by all the boat icons that where added on the map. There is a for loop iterating through the `boatArray` array that contains all the data about the added boats. In each iteration of the loop it is computed the distance in meters of the boat respect to its direction; knowing the speed of the boat $v$ and the time elapsed since the last call to the function $t$, easily we know the space covered $s = v/t$, now we apply the following formula to discover the pixel moved respect to the X axis and to the Y axis of the map:

$$posX = s \cdot \sin (\psi_b) / R \quad \text{and} \quad posY = s \cdot \cos (\psi_b) / R$$

where $\psi_b$ is the direction of travelling of the boat and $R$ is a constant representing the ratio between one pixel of the map and one metre (in our case it is 1.14). These two results are added to the current position of the boat respectively on the X axis and on the Y axis. When a boat reaches a position occupied by land or a position outside the map it turns by 180 degrees and moves at the same speed as before.

Lastly we are now going to describe the `calcTangentToWave` function, in which all of the parameters for the management of the robot position over the waves are computed. Let us think, in a simplistic but quite realistic way, of a wave as a sinusoid (Figure 3.12). We know the position of the robot on the wave along the X axis and we call it $xRobotPos$, this variable has as value the distance covered by the robot onto the wave as stated in equation 3.2, summed with the initial position of the robot on the wave. In addition we know the amplitude of the wave $A$ and the length of the wave $\lambda$ (see 3.3.2.6), so we can compute the spatial frequency of the wave $k = 2\pi/\lambda$. In this way
the equation of the sine curve that represents the wave is [5]:

$$f(x_{\text{RobotPos}}) = y_{\text{RobotPos}} = A \cdot \sin (k \cdot x_{\text{RobotPos}})$$

(3.3)

and $y_{\text{RobotPos}}$ is the position of the robot on the wave along the Y axis (i.e. the height coordinate). Now, we would like to know the slope at the robot position on the wave to see the behaviour of the pitch and roll angles of the robot and therefore the behaviour of the laser sensor. As for every function the slope is calculated using the first derivative in the point we are analysing:

$$f'(x_{\text{RobotPos}}) = \text{slope} = A \cdot k \cdot \cos (k \cdot x_{\text{RobotPos}})$$

the pitch and roll of the robot are computed as follows:

$$pitch = \arctan (slope \cdot \cos (\Delta)) ;$$

$$roll = \arctan (slope \cdot \sin (\Delta))$$
where $\Delta$ is the angular difference between the heading of the robot and the direction of the wave. At this point we can compute the position of the laser sensor over our robot and therefore the starting point of every laser ray, in this way; knowing the height of the robot above the water (in our case 0.5 metres) and knowing the slope of the wave at the robot’s position, as for every line segment on a Cartesian plane we can write that the distance between the robot position on the wave, $RP = (x_{\text{RobotPos}}, y_{\text{RobotPos}})$, and the point where the laser is placed on the robot, $LS = (x_{\text{LaserStart}}, y_{\text{LaserStart}})$, is the height of the robot above water $h = 0.5$ and the slope of the line that connects the points $RP$ and $LS$ is $m = -\frac{1}{\text{slope}}$, since such line is perpendicular to the tangent to the wave. So,

\[
\Delta x = x_{\text{LaserStart}} - x_{\text{RobotPos}}; \\
\Delta y = y_{\text{LaserStart}} - y_{\text{RobotPos}}
\]

\[
\begin{align*}
\begin{cases}
  h &= \sqrt{(\Delta x)^2 + (\Delta y)^2} \\
  m &= \frac{\Delta y}{\Delta x}
\end{cases} \\

\Rightarrow \begin{cases}
  \Delta y &= m \cdot \Delta x \\
  h &= \frac{\Delta y}{\Delta x} \Rightarrow h = \frac{\Delta y}{\Delta x} = \sqrt{(\Delta x)^2 + m^2 \cdot (\Delta x)^2} \\
  \Delta x &= \frac{h}{\sqrt{m^2 + 1}} \Rightarrow \\
  x_{\text{LaserStart}} &= x_{\text{RobotPos}} + \frac{h}{\sqrt{m^2 + 1}}
\end{cases}
\end{align*}
\]

(3.4)

and

\[
\begin{align*}
\Delta y &= m \cdot \frac{h}{\sqrt{m^2 + 1}} \Rightarrow y_{\text{LaserStart}} &= y_{\text{RobotPos}} + \frac{m \cdot h}{\sqrt{m^2 + 1}}
\end{align*}
\]

(3.5)

in the case of negative slope:
\[ \Delta x = x_{\text{RobotPos}} - x_{\text{LaserStart}}; \]
\[ \Delta y = y_{\text{RobotPos}} - y_{\text{LaserStart}} \]

and therefore

\[ x_{\text{LaserStart}} = x_{\text{RobotPos}} - \frac{h}{\sqrt{m^2 + 1}}; \quad (3.6) \]
\[ y_{\text{LaserStart}} = y_{\text{RobotPos}} - \frac{m \cdot h}{\sqrt{m^2 + 1}} \quad (3.7) \]

Of course all these formulae are valid in the case \( \text{slope} \neq 0 \), otherwise \( x_{\text{LaserStart}} = x_{\text{RobotPos}} \) and \( y_{\text{LaserStart}} = y_{\text{RobotPos}} + h \).

After this view over the function called at the timeout trigger we can now see the other slot functions present in the program. In the \textit{init} function are connected several signals, especially the \texttt{clicked()} ones, to slots referring to the buttons, checkboxes and icons of the Commands panel. In the following paragraphs we will see the functionalities of these widgets and their implementations.

**Hide/Show checkboxes**  The first four widgets from on high are checkboxes that are used to hide or show some properties on the map if checked or not.

The first one enables the showing of the sea current arrows that are the icons used to simulate the direction and the speed of the current in several places of the map. In total these arrows are forty and are disposed in eight columns of five arrows each equally spaced out from each other to cover the most of the area of the map.

The second one has the same function of the previous except for showing or hiding the arrows representing the winds. On the map there are fifty-four wind arrows disposed in nine columns of six each, equally spaced out from each other to cover the most of the area of the map and in a way that they do not overlap with the sea current arrows.
The third of these widgets if checked shows the net which the map is covered with. This net is formed by thin light grey lines that cross the map from left to right and from up to down; there are twenty-three vertical lines and sixteen horizontal ones to form squares of fifty by fifty pixels. The function of this net is only to better visualise distances between, directions and positions of objects on the map.

The last checkbox of this group performs the showing or hiding of the path that the robot covered during its movements. It is represented by a series of red connected lines onto the map. To create these lines in the paintEvent function the items of the pathArray array, that are points, are scrolled and for each couple of them a line is drawn.

To perform the actions described before in the paintEvent function it is simply examined every of these checkboxes and if one is checked the icons or lines are drawn otherwise they are not.

**Zooming**  
The two buttons dedicated to the zoom functions when clicked are connected to two slot functions; such functions are called zoomIn and zoomOut. As it is understandable from their names the first one performs the zooming in of the map and all the icons on it, the second one performs the zooming out of the same items. In the program there is a constant called zoomRatio that identifies the increase or the decrease ratio of the dimensions of the zoomed items each time the button to zoom in or to zoom out is pressed; this constant has a value of 1.25 that means that the zoomed items are enlarged or shrunk by 25% of their dimensions.

To set the dimensions of each item depending on the zoom level, their dimensions at normal zoom level are multiplied by \( \text{zoomRatio}^{\text{zoomLevel}} \) where \( \text{zoomLevel} \) is a variable that is increased by one each time the zoom in button is pressed and is decreased by one each time the zoom out button is pressed and has value zero as default and at normal zoom level.

Using the simulator the user can zoom in and zoom out also using the scroll wheel of the mouse. This functionality is fulfilled reimplementing the wheelEvent function of the QWidget class. This function is called every time a QWheelEvent is triggered (i.e. every time the mouse wheel is moved); such event object contains the number of steps the wheel rotated, at every step of the wheel the zoomLevel variable is increased.
or decreased by one depending if the wheel was rotated forwards away the user or backwards toward the user.

**View buttons** In this paragraph is described the implementation of the other five buttons of the Commands panel.

The first one we analyse is the one that let the user drag the map. Clicking on it the slot function `handMode` is called; here a boolean, called `handCursor`, that indicates that this button was pressed is set to true, the cursor icon is changed into an open hand and when the left mouse button is pressed it is changed into a closed hand. In the `mousePressEvent` event handler function is intercepted the event of a mouse button pressed and if the `handCursor` boolean is true an other boolean, `beginDrag`, is set to true and the point in the simulator, where the mouse cursor was when the left mouse button was pressed, is memorised into a variable. In the `mouseMoveEvent` event handler function are intercepted all the movements of the mouse cursor hence it is verified if the `beginDrag` boolean is true and if so the origin of the map is updated using the following formula:

\[
O_x = O_{x_{old}} + (pos_x - d_x);
O_y = O_{y_{old}} + (pos_y - d_y)
\]

where \(O\) is the map origin, \(pos\) is the current position of the mouse cursor and \(d\) is the position of the mouse cursor when the drag started; this last parameter is updated with the value of \(pos\) in the `mouseMoveEvent` function, so \(pos - d\) is the distance between the current cursor position and the position where the cursor was the last time it was moved.

The second button has the function of centring the viewport of the simulator at the robot’s position. Clicking on it the slot function `homeMode` is called; here the boolean `handCursor` is set to false and the cursor icon is changed into the default one (i.e. an arrow). In the `paintEvent` function the new origin of the map is calculated in this way:
\begin{align*}
O_x &= \frac{width}{2} - (robot_{posX} \cdot zoomRatio^{zoomLevel}), \\
O_y &= \frac{height}{2} - (robot_{posY} \cdot zoomRatio^{zoomLevel})
\end{align*}

where width and height are the width and height of the simulator window, \(robot_{posX}\) and \(robot_{posY}\) are the position of the robot in pixels respect of the origin along the X axis and the position of the robot in pixels respect of the origin along the Y axis.

The third button let the user set a goal point on the map that the robot should reach. When this button is pressed the moveToMode slot function is called and in here a boolean, flagCursor, is set to true and a flag is set as the cursor icon. In the mousePressedEvent function when the boolean flagCursor is true, the point in the map where the user clicked is registered in a variable and a popup window shows up, asking the user various parameter to be set to indicate the navigation algorithm of the robot the behaviour of the robot itself to reach the goal. If the OK button of the popup window is pressed the flag icon is set onto the map and the message to be sent to the navigation algorithm is prepared. Otherwise, if the Cancel button of the popup window is pressed nothing happens.

The other two buttons are responsible of the map background view. One when pressed activate a slot function that changes the image that represents the map with a white and black topographic map, the other one when pressed is connected to a slot function that changes the image with a satellite picture of the same area of the topographic map.

**Adding boats and shallows features** At the bottom of the Command panel there are two icons, one represents a shallow water and the other one a boat. When the first one of this icon is dragged onto the map it is moved at the mouse cursor position and when the mouse button is released the shallows popup window appears. This drag and drop feature is done by reimplementing the eventFilter event handler function. This function filters every event is triggered by one of the components of the main widget; in this case in the reimplementation of the function it is checked the type of event and the object that triggered it, if is a MouseButtonPress event and the object is the boat icon the
boolean `iconBoatDragged` is set to true, in the case that the event is `MouseButtonPress` and the object is the shallow icon the boolean `iconShallowsDragged` is set to true. If the event is a `MouseMove` and one of the boolean `iconBoatDragged` or `iconShallowsDragged` is true a `QDrag` object is created with associated a `QMimeData` object that contains as text an identifier “B”, if the dragged icon is the boat one, “S” if it is the shallow one. The `QDrag` object contains also as pixmap the pixmap of the icon dragged.

Furthermore in this function is checked again the event type controlling if it is a `MouseButtonRelease`, in this case the boolean `iconBoatDragged` is set to false, if the triggering object was the boat icon otherwise `iconShallowsDragged` is set to false. In the `dropEvent` event handler function, that filters the events when an object is dropped somewhere in the main widget, is checked the `QMimeData` object of the event that contains the same items of the one in the `QDrag` object created in the `eventFilter` function; if such object contains as text the identifier “B”, it means that the boat icon was dropped and that the boat creation popup window has to be shown, otherwise if it contains an “S”, the shallow creation popup window is unfold.

These two popup windows have a Cancel and an OK button each and each of such buttons have associated a slot function for their clicked signal. In the case of the slot functions connected to the Cancel buttons, the popup window is simply hidden; instead for the OK button slot function a new boat is added to the `boatArray` array if the popup window was the one of boat creation or a new shallow is added to the `shallowsArray` array if the popup window was the one of shallow creation and in both cases the window is hidden as well.

**Tooltip** A graphical element implemented during the development of the program is the tooltip for the various icons present on the map. The tooltip feature is shown after the user moves the mouse cursor over an icon of the map, such as the wind and sea currents arrows and the boats. It is implemented as a black rectangle with information about the icon written in white. Let us now see first how the tooltip for the current and wind icons works.

In the `mouseMoveEvent` function there is a for loop scrolling all the sea current and the wind icons and if the mouse cursor is near the icon a boolean called `toolTipBool`
is set to true, in a current variable called toolTipCurrent is memorised the current or
the wind data of the icon where the mouse is passing over, and the QPoint toolTipPos
variable takes the value of the current position of the mouse cursor. To see if the mouse
cursor is near an icon the following condition must be true:

\[ \text{manhattanLength}(\text{pos} - \text{currPoint}) < \text{current}_\text{size} \cdot \text{zoomRatio}^{\text{zoomLevel}} \]

where the manhattanLength function returns the sum of the absolute values of x and
y coordinates, traditionally known as the “Manhattan length” of the vector from the
origin to the point resulting from the subtraction \(\text{pos} - \text{currPoint}\); where \(\text{pos}\) represents
the position of the mouse cursor and \(\text{currPoint}\) the position of the current or wind
arrow icon; \(\text{current}_\text{size}\) is the size of the current or wind icon calculated depending on
its intensity.

In the paintEvent function it is checked the boolean toolTipBool, if it is true the
tooltip is drawn in the position contained in the variable toolTipPos and with the data
collected in the variable toolTipCurrent as tooltip message.

Similarly for the boats tooltip, the position of the tooltip and the data of the boat
are hold into variables, in the mouseMoveEvent function and in the paintEvent function
the tooltip is drawn considering the data stored in those variables.

**Boat dragging** As we previously told boat icons can move autonomously on the
map, but they can even be replaced by the user that is allowed to drag them. In this
paragraph we will see how this is implemented. We will start our analysis from the
mouseMoveEvent function; in here it is checked if the mouse cursor is near a boat using
a similar formula as the one for the tooltips, in this case the mouse cursor is changed
and a boolean, called moveBoatCursor, is set to true. Now, in the mousePressEvent
function this boolean is verified and if true means that the user clicked on the boat,
so an other boolean, beginDragBoat, indicating that the drag of the boat started is set
to true and in the meanwhile the current position of the mouse cursor is recorded in
a variable. Returning in the mouseMoveEvent function, the boolean beginDragBoat is
controlled, if true the boat coordinates are changed considering the difference of the
point where the cursor was when the user pressed the mouse button and the current cursor position. The applied formulae are the following:

\[
\begin{align*}
\text{boatSelected}_{posX} &= \text{boatSelected}_{posX_{old}} + \frac{(pos_x - \text{distanceFrmLastMouseEvnt}_x)}{\text{zoomRatio}^{\text{zoomLevel}}} \\
\text{boatSelected}_{posY} &= \text{boatSelected}_{posY_{old}} + \frac{(pos_y - \text{distanceFrmLastMouseEvnt}_y)}{\text{zoomRatio}^{\text{zoomLevel}}}
\end{align*}
\]

where \(\text{boatSelected}_{posX}\) and \(\text{boatSelected}_{posY}\) are respectively the coordinates along the X and Y axes of the boat that is dragged, \(pos_x\) and \(pos_y\) are the current position along the two axes of the mouse cursor and \(\text{distanceFrmLastMouseEvnt}_x\) and \(\text{distanceFrmLastMouseEvnt}_y\) are the coordinated along the two axes of the point where the mouse button was pressed and is updated to the current mouse position each time the above formulae are calculated.

In the mouseReleaseEvent function simply the boolean \beginDragBoat\ is set to false.

### 3.3.2.3 Laser simulation thread

In this section we will see how the laser sensor is simulated in our virtual environment. As previously mentioned in the main function of the project the thread that takes care about the simulation of the laser is created and executed. This thread is an extension of the class QThread and so when it is started, the run function is called. This function is a simple infinite loop in which there is a function that performs the calculations to obtain the laser array of the distances from objects and the time that this computation takes to be done, is clocked; if this time is minor than the real laser update time (i.e. 250 milliseconds) the difference in time between the length of the computation and the real laser update time is slept by the thread.

To compute all the distances of every laser ray from the objects, the algorithm has been divided in more phases. First are computed the distances between the robot and the coastlines considering only the horizontal distances. Then if some of the rays intersect the coastline, those distances are calculated also in the vertical dimension considering the wave amplitude at the robot position on it and the laser ray slope.
depending on its direction respect to the one of the wave. In addition to this the
distances are calculated also from boats first in the horizontal plane and then in the
vertical one if some rays have intersected a boat in the horizontal plane. Eventually
the distances are reckoned from waves too.

Coastline distances Let see how is implemented the algorithm to calculate the dis-
tances from coastlines. Initially we have to understand where a coastline is on the map;
it is quite trivial since considering the black and white map, coastlines are those part
coloured in black. The algorithm scans every pixel in a rectangular area around the
robot. Such rectangle is made in this way:

\[
\begin{align*}
\text{startPx}_x &= \text{robot}_{posX} - \frac{\text{laserMaxDistance}}{R}; \\
\text{startPx}_y &= \text{robot}_{posY} - \frac{\text{laserMaxDistance}}{R}; \\
\text{width} &= 2 \cdot \frac{\text{laserMaxDistance}}{R}; \\
\text{height} &= 2 \cdot \frac{\text{laserMaxDistance}}{R};
\end{align*}
\]

where \(\text{startPx}_x\) is the x-coordinate of the top-left corner point of the rectangle,
\(\text{startPx}_y\) is the y-coordinate of the top-left corner point of the rectangle, \(\text{width}\) and
\(\text{height}\) are the rectangle width and height respectively, \(\text{laserMaxDistance}\) is the maxi-
mum distance the laser can “see” an object (in our case 15 metres) and \(R\) is the ratio
between one pixel of the map and one metre as said before. The scan is performed using
a nested loop to scan each line of the rectangle and for every line each pixel that forms
that line, for each pixel is controlled its colour and if it is darker than a dark grey the
pixel is stored in an array of \(\text{QPoint}\); to know this fact the pixel’s colour representation
in the ARGB (alpha, red, green and blue channels) quadruplet on the format \#AARRGGBB
must be less than 0xFFFCFCFCF. This array is used to know which points of
the map could be intersected by a laser ray, in fact it is computed the segment line that
represents the laser ray and checked if each of those black pixel points of the array lie
on such line.
Hence, to compute the line we consider the start and end point of it. The first one of these points is the centre of the robot from where, for simplicity without affecting the calculation in a wrong way, it is supposed to be located the laser sensor, so with coordinates \( \text{robot}_{\text{posX}} \) and \( \text{robot}_{\text{posY}} \) along the X and Y axes. The other point is computed knowing the starting point of the laser and the angle of the ray direction respect to North. So the following formula is adopted:

\[
\begin{align*}
\text{endLaserPnt}_x &= \text{robot}_{\text{posX}} + \sin(\Delta) \cdot \frac{laser_{\text{MaxDistance}}}{R}; \\
\text{endLaserPnt}_y &= \text{robot}_{\text{posY}} - \cos(\Delta) \cdot \frac{laser_{\text{MaxDistance}}}{R}
\end{align*}
\]

where \( \text{endLaserPnt} \) is the point on the map that represents the end of the laser ray, \( \Delta \) is the angle between the ray direction and North, considering the X axis travelling in ascending order from left to right of the map and the Y axis from up to down of the map.

Then for each ray of the laser after knowing the start and the end of it, a loop iterating through the black pixel points of the map is performed and in each iteration is checked if the point lies on the segment line that represents the laser ray. To see this, these formulae have been considered:

- In the case of a segment line parallel to the Y axis

\[
d = \text{blackPnt}_x - \text{startLaserPnt}_x
\]

(3.8)

- In the case of a segment line parallel to the X axis

\[
d = \text{blackPnt}_y - \text{startLaserPnt}_y
\]

(3.9)

- In general for all the other cases
where blackPnt is the black pixel point of the map that has to be controlled if lying on the laser ray segment line, $m$ is the slope of the line, $q$ the $y$-intercept of the line and $d$ is the distance from the point to the line, calculated as in [9]. To understand if the point lies on the line we simply control if $d < \varepsilon$ where $\varepsilon$ is a little value for example 0.5. If this condition is true, are computed the distances from the black point to the start of the laser segment line and from the black point to the end of the laser segment line using the Pythagorean theorem:

$$d_{\text{start}} = \sqrt{(\text{startLaserPnt}_y - \text{blackPnt}_y)^2 + (\text{startLaserPnt}_x - \text{blackPnt}_x)^2}$$

$$d_{\text{end}} = \sqrt{(\text{endLaserPnt}_y - \text{blackPnt}_y)^2 + (\text{endLaserPnt}_x - \text{blackPnt}_x)^2}$$

now if $d_{\text{start}} < \text{minDist} \land d_{\text{start}} \geq 0 \land d_{\text{end}} \leq \text{laserMaxDistance}$ is true, the minimum distance, $\text{minDist}$, from a black point to the robot takes the value of $d_{\text{start}}$, considering $\text{minDist}$ initialised as the maximum distance the laser sensor can see, before the loop iterating through the black pixels.

If there is intersection between a point of the map belonging to land, hence a distance less than the maximum one is computed for that line segment, we will check if, considering the slope of the laser rays caused by the waves, this distance has to be changed in value.

**Coastline distances with waves** In this phase initially is taken in consideration the angle between the line segment that represents the laser ray and the direction where
the wave comes from respect to the robot position, modulo 360 degrees. This angle, called $\Delta$ has the function of computing a coefficient to be multiplied to the slope of the laser ray that travels along the longitudinal axis of the robot to know the actual slope for the laser ray we are considering.

The coefficient is simply $coeff = \cos(\Delta)$ and therefore the slope of the ray we are taking in consideration respect to the horizontal plane is $slope_r = slope_{\perp\text{wavecrest}} \cdot coeff$, where $slope_{\perp\text{wavecrest}}$ is the slope respect to the horizon of the laser ray that is exactly perpendicular to the crest of the wave, that is the ray that has the same heading respect to North as the direction where the wave comes from.

To understand this fact we can analyse the progress of the variable $coeff$, it has the highest value, 1 when $\Delta = 0$, then decreases passing through the value of 0 when $\Delta = \frac{\pi}{2}$, in fact in this case the slope is 0 because the laser ray is exactly parallel to the wave crest, $coeff = -1$ when $\Delta = \pi$, because the laser ray in this position has the exactly heading of the one of the wave crest thus has a slope with opposite value of the one at $\Delta = 0$, from this point $coeff$ increases crossing the value of 0 at $\Delta = \frac{3}{2}\pi$, in fact also in this case the slope is 0 because the laser ray is exactly parallel to the wave crest and eventually $coeff$ reaches the value of 1 again at $\Delta = 2\pi$.

Once we have computed the slope of the laser ray respect to the horizon, we have to calculate the distance from the coastline along the ray direction. Considering Figure 3.13, we know the distance in horizontal from the robot position to the coastline, $min\text{Dist}$, here called $\Delta x$, we know the slope, $slope_r$, here called $m$, hence, applying the Pythagorean theorem we obtain the oblique distance $d_{obl}$:

$$
\begin{align*}
\begin{cases}
d_{obl} &= \sqrt{(\Delta x)^2 + (\Delta y)^2} \\
m &= \frac{\Delta y}{\Delta x}
\end{cases}
\Rightarrow
\begin{cases}
\Delta y &= m \cdot \Delta x \\
d_{obl} &= \frac{\Delta x}{\sqrt{(\Delta x)^2 + m^2 \cdot (\Delta x)^2}}
\end{cases}
\Rightarrow
d_{obl} &= \Delta x \cdot \sqrt{m^2 + 1}
\end{align*}
$$

now, if $d_{obl} > laser\text{MaxDistance}$, $d_{obl}$ takes the value of $laser\text{MaxDistance}$ otherwise we can compute $\Delta y = m \cdot \Delta x$ and control if this value summed to the height of the starting point of the laser ray above sea level, computed in 3.5 or in 3.7, is greater than the height of the coastline, that in our case is considered 1 metre. If so the actual distance is set to $laser\text{MaxDistance}$, otherwise the actual distance is considered as $d_{obl}$.
Boat distances  To calculate the distances from the boats, as for the coastlines, the algorithm is divided in two parts; initially are considered the distances on the horizontal plane, then if there are interceptions between the laser rays and at least one boat, those distances are computed also considering the waves influence on the laser rays.

So, at the beginning all the boats data present in the boatArray array are scrolled in a for loop. For each boat the positions of its vertices are mapped in the world coordinate system using the map function of the QTransform object, so that are in agreement with the coordinates system used for the robot position. Boats have seven vertices hence are represented in a heptagonal shape and knowing the vertices we can know the functions that describe the segment lines that connect each couple of such vertices and control if there are intersections between them and the laser rays. In the function it is adopted a simplified 2D version of the 3D ray tracing algorithm used in [6], quite similar to the one in [7]. Let us consider image 3.14, we can say that line $a$ is the laser ray and line $b$ is a segment line of the boat shape.

Now, the equations of the two lines are:

$$Pa = P1 + u_a(P2 - P1);$$
Figure 3.14: Example of two intersecting lines.

\[ Pb = P3 + u_b(P4 - P3) \]

Solving for the point where \( Pa = Pb \) gives the following two equations in two unknowns \( (u_a \text{ and } u_b) \)

\[ x_1 + u_a(x_2 - x_1) = x_3 + u_b(x_4 - x_3) \]

and

\[ y_1 + u_a(y_2 - y_1) = y_3 + u_b(y_4 - y_3) \]

solving gives the following expressions for \( u_a \) and \( u_b \)

\[ u_a = \frac{(x_4 - x_3) \cdot (y_1 - y_3) - (y_4 - y_3) \cdot (x_1 - x_3)}{(y_4 - y_3) \cdot (x_2 - x_1) - (x_4 - x_3) \cdot (y_2 - y_1)}; \]

\[ u_b = \frac{(x_2 - x_1) \cdot (y_1 - y_3) - (y_2 - y_1) \cdot (x_1 - x_3)}{(y_4 - y_3) \cdot (x_2 - x_1) - (x_4 - x_3) \cdot (y_2 - y_1)} \]

Substituting either of these into the corresponding equation for the line gives the intersection point. For example the intersection point \((x,y)\) is:

\[ x = x_1 + u_a(x_2 - x_1); \quad (3.13) \]

\[ y = y_1 + u_a(y_2 - y_1) \quad (3.14) \]
• if the denominator for the equations for $u_a$ and $u_b$ is 0 then the two lines are parallel.

• if the denominator and numerator for the equations for $u_a$ and $u_b$ are 0 then the two lines are coincident.

• the equations apply to lines, if the intersection of line segments is required, like in our case, then it is only necessary to test if $u_a$ and $u_b$ lie between 0 and 1. Whichever one lies within that range then the corresponding line segment contains the intersection point. If both lie within the range of 0 to 1 then the intersection point is within both line segments.

If there is an intersection point, the algorithm computes the distance between the laser starting point, $(robot_{posX}, robot_{posY})$, and the intersection point, $(intersPnt_x, intersPnt_y)$, using the Pythagorean theorem

$$d_{horiz} = \sqrt{(robot_{posX} - intersPnt_x)^2 + (robot_{posY} - intersPnt_y)^2} \cdot R$$  \hfill (3.15)

as usual $R$ is the ratio between one pixel of the map and one metre. This equations are applied for every line segment of the boat shape or until two intersections are found for that laser ray, since the shape of the boat is convex, each laser ray can intersect the boat borders zero or two times. If such distance is less than the maximum distance the laser can see (i.e. 15 metres), then the algorithm computes the distances considering the height of the laser onto the robot, the height of the robot onto the wave and the height of the boat.

**Boat distances with waves** We apply a similar solution as for the computation of the coastline distances with waves. The slightly difference is in the fact that the boats float, hence its height respect to horizon varies. To know where the boat is onto the wave such that we can obtain its height, we compute the distance from the robot along the direction of the wave, considering the distance on the horizontal plane (3.15) multiplied by the cosine of the angle between the direction of the laser ray and the direction where
the wave comes from \((d_{\text{side}} = d_{\text{horiz}} \cdot \cos(\Delta))\). Knowing the equation that represents the wave (3.3) we can compute the position on the wave respect to the horizon of the boat in such way:

\[
y_{\text{BoatWave}} = A \cdot \sin(k \cdot (x_{\text{LaserStart}} + d_{\text{side}}))
\]

where \(x_{\text{LaserStart}}\) is computed in 3.4 or in 3.6 and is the position of the laser onto the wave in the x-coordinate. Now we can apply the same algorithm as for the coastline distances, \(\text{slope}_r = \text{slope}_{\perp \text{wavecrest}} \cdot \cos(\Delta)\), \(d_{\text{obl}} = d_{\text{horiz}} \cdot \sqrt{m^2 + 1}\). If \(d_{\text{obl}} > \text{laserMaxDistance}\), \(d_{\text{obl}}\) takes the value of \(\text{laserMaxDistance}\) otherwise we can compute \(h_{\text{ray}} = m \cdot d_{\text{horiz}}\) and control if this value summed to the height of the starting point of the laser ray above sea level, \(y_{\text{LaserStart}}\) (3.5, 3.7), is greater than the height of the boat summed with the position of the boat on the wave (\(y_{\text{BoatWave}}\)).

\[
h_{\text{boatOverHoriz}} = (y_{\text{BoatWave}} + \text{boatheight});
\]

if \(h_{\text{ray}} + y_{\text{LaserStart}} > h_{\text{boatOverHoriz}}\) is true, the actual distance is set to \(\text{laserMaxDistance}\), otherwise the actual distance is considered as \(d_{\text{obl}}\).

**Distances from wave** The last kind of object the laser rays can intersect are the sea waves. To compute the distances that the laser measures, the following algorithm has been adopted. The angle between the laser ray and the direction where the wave comes from is called \(\Delta\); to compute the intersection between the laser ray and the sea wave we have to solve the following equation system:

\[
\begin{align*}
y_{\text{Inters}} &= m \cdot x_{\text{Inters}} + q \\
y_{\text{Inters}} &= A \cdot \sin(k \cdot x_{\text{Inters}})
\end{align*}
\]

hence the next equation, with \(x_{\text{Inters}}\) as unknown, has to be solved

\[
m \cdot x_{\text{Inters}} + q = A \cdot \sin(k \cdot x_{\text{Inters}})
\]
, since such equation cannot be solved analytically we adopt the following algorithm
in order to solve it. Firstly we compute the position of the end point of the laser ray
in this way, similarly as in 3.4, 3.5, 3.6 and 3.7:

\[
\begin{align*}
\text{if } \cos(\Delta) > 0 \\
& \quad x_{\text{Laser End}} = x_{\text{Laser Start}} + \frac{\text{laser Max Distance}}{\sqrt{m^2 + 1}} \\
& \quad y_{\text{Laser End}} = y_{\text{Laser Start}} + \frac{m \cdot \text{laser Max Distance}}{\sqrt{m^2 + 1}} \\
\text{else} \\
& \quad x_{\text{Laser End}} = x_{\text{Laser Start}} - \frac{\text{laser Max Distance}}{\sqrt{m^2 + 1}} \\
& \quad y_{\text{Laser End}} = y_{\text{Laser Start}} - \frac{m \cdot \text{laser Max Distance}}{\sqrt{m^2 + 1}}
\end{align*}
\]

then, the algorithm scrolls in a for loop the line every \( \varepsilon = 0.1 \) metres from the start
point of the laser ray to its end, computing:

\[
\begin{align*}
\text{yLaser} & = m \cdot (i - x_{\text{Laser Start}}) + y_{\text{Laser Start}} \\
\text{yWave} & = A \cdot \sin(k \cdot i)
\end{align*}
\]

where \( i \) is the \( i^{th} \) analysed point along the x-coordinate,

\[
\text{if } |y_{\text{Laser}} - y_{\text{Wave}}| < \frac{\varepsilon}{2} \text{ then } x_{\text{Inters}} = i, y_{\text{Inters}} = y_{\text{Laser}} \approx y_{\text{Wave}}, \text{ so the}
\]
distance from the ray laser starting point and the sea wave, applying the Pythagorean
theorem, is:

\[
d_{\perp \text{wave crest}} = \sqrt{(x_{\text{Inters}} - x_{\text{Laser Start}})^2 + (y_{\text{Inters}} - y_{\text{Laser Start}})^2}
\]

In this way we have reckoned the distance between the laser ray that is perpendicular
to the wave crest. In order to compute the actual distance for the given laser ray,
considering image 3.15,
if \( \cos(\Delta) = 0 \), \( d_{\text{wave}} = \text{laserMaxDistance} \) since the angle between the laser ray and the wave crest is 90° so the ray is parallel to the wave crest and will never intersect the wave, otherwise, in trigonometry a cathetus of a right-angled triangle is equals to the hypotenuse times the cosine of the adjacent angle [10], so the hypotenuse is equals to the cathetus divided by the cosine of the adjacent angle, in our case \( d_{\text{wave}} = \frac{d_{\text{wave crest}}}{\cos(\Delta)} \).

![Diagram](image)

Figure 3.15: Laser rays angles with wave from above.

Eventually, when computed all these types of distances for each laser ray, the shortest one is considered as distance measured by the laser and inserted in the \(_{\text{arrLaser}}\) array of integers.

### 3.3.2.4 Sonar simulation thread

In this section is described how the sonar sensor is simulated in our virtual environment. In a way completely similar to the one for the laser simulation, in the main function of the project the thread that takes care about the simulation of the sonar is created and executed. This thread is an extension of the class QThread and so when it is started,
the run function is called. Such function is an infinite loop in which there is a function that performs the calculations to obtain the sonar matrix containing positive values if an objects is seen by the sensor at the position specified by the matrix cell index and zeroes if there are not seen objects for that position.

Besides this, the time that this computation takes to be done, is clocked; if this time is minor than the real sonar update time (that depends on the range of the sonar angle and the length of the scan) the difference in time between the length of the computation and the real sonar update time is slept by the thread.

Also this algorithm is divided in more parts, there is one that controls the presence of land in the nearby of the robot, the second part checks if there are boats near the robot and finally the sonar simulator checks the presence of shallows in the surroundings of the robot.

Waves are not considered for the sonar because they move the pitch and roll of the robot by angles whose magnitude is too small to be influential for the behaviour of the sonar.

**Land recognition** As for the simulation of the laser, initially an array is filled with all the black pixels (points that represents land) of the map in the surrounding of the robot. Then for each scanline of the sonar it is checked if such black points lie on the line that represents the scanline; the algorithm used is the same as for the laser sensor simulation (formulae 3.8, 3.9 and 3.10). Now, if a point lies on the line, we compute the distances of the point from the start, $d_{\text{start}}$, and from the end, $d_{\text{end}}$, of the scanline using exactly the same formulae as in 3.11 and 3.12.

Considering $\text{sonarCurrentDistance}$ as the current distance up to which the sonar can recognise objects, such that it has a value of 25 metres when the robot is travelling with a speed less than 1.2 m/s, otherwise 50 metres, if $d_{\text{start}} < \text{sonarCurrentDistance} \land d_{\text{start}} \geq 0 \land d_{\text{end}} < \text{sonarCurrentDistance}$ is true, the cell of the sonar matrix corresponding to the $d_{\text{start}}$ distance for that scanline is filled with the value 100. The cell number is calculated as

$$\text{cellNum} = \frac{d_{\text{start}} \cdot \text{sonarBinsInScan}}{\text{sonarCurrentDistance}} = \frac{d_{\text{start}}}{\text{sonarBinsInScan}} \cdot \frac{\text{sonarBinsInScan}}{\text{sonarCurrentDistance}}$$

(3.16)
where $sonarBinsInScan$ is the number of bins for each scanline (usually 100), hence

$$sonarMatrix[i][cellNum] = 100$$

where $i$ is the index of the scanline; since a point of the map can be greater than the area covered by a bin of the sonar also the surrounding cells of the matrix are filled with the value 100:

$$sonarMatrix[i][cellNum + 1] = 100;$$
$$sonarMatrix[i][cellNum - 1] = 100;$$
$$sonarMatrix[i][cellNum + 2] = 100;$$
$$sonarMatrix[i][cellNum - 2] = 100;$$
$$sonarMatrix[i][cellNum + 3] = 100;$$
$$sonarMatrix[i][cellNum - 3] = 100.$$

**Boats recognition**  
Despite sonar sensor scan under the water level it can “see” also boats since they have a part under the water. To simulate this type of recognition from the sonar, the next algorithm has been used. Each boat contained in the boatArray array is controlled, for each of them every segment line of their shape is computed and using the formulae of 3.13 and 3.14, it is checked if the line that represents the sonar scanline intersects one of the segment lines that constitutes the borders of the boat. When two interception points, $a$ and $b$, are found for that boat, the distances from the sonar starting point, $sonarPnt$, (i.e. the position of the robot) and the intersection points are computed like usual using the Pythagorean theorem:

$$d_a = \sqrt{(sonarPnt_y - a_y)^2 + (sonarPnt_x - a_x)^2};$$  
$$d_b = \sqrt{(sonarPnt_y - b_y)^2 + (sonarPnt_x - b_x)^2}.$$
and the corresponding sonar matrix cells are computed in this way:

\[ \text{cellNum}_a = \frac{d_a}{\text{sonarBinsInScan}} \times \frac{\text{sonarCurrentDistance}}{\text{sonarBinsInScan}} = \frac{d_a \cdot \text{sonarBinsInScan}}{\text{sonarCurrentDistance}}; \]
\[ \text{cellNum}_b = \frac{d_b}{\text{sonarBinsInScan}} \times \frac{\text{sonarCurrentDistance}}{\text{sonarBinsInScan}} = \frac{d_b \cdot \text{sonarBinsInScan}}{\text{sonarCurrentDistance}}. \]

hence every cell between \( \text{cellNum}_a \) and \( \text{cellNum}_b \) for that scanline is filled with the value 100.

Unluckily there are some particular cases where only one intersection point is found; this happens when a part of the boat is inside the cone of visualisation of the sonar and the rest is outside. To simulate the behaviour of the sonar for these situations the algorithm simply checks if the starting point of the sonar scanline or the ending point of the scanline we are analysing lies inside the shape of the boat. To do this it has been used the Qt function \text{containsPoint}(\text{const QPointF} &\text{point}, \text{Qt::FillRule fillRule}) of the \text{QPolygonF} class, that returns true if the given point is inside the polygon according to the specified fillRule, otherwise returns false [29]. If the starting point of the scanline lies inside the boat shape, the missing cell number \( \text{cellNum}_x \) (with \( x = a \lor b \)) is set to the value of 0 that means that it is the first cell of the scanline, otherwise if the ending point of the scanline lies inside the boat polygon \( \text{cellNum}_x = \text{sonarBinsInScan} - 1 \), that means that the cell is the last one of the scanline.

**Shallows recognition** One of the most important properties of the sonar is that it can recognises every object under water in particular rocks or other kind of shallows. In our simulator shallows can be added by the user onto the map and they are represented as ellipses. To perform the recognition of such objects the program acts as following, in a similar way as for boats recognition; there is a for loop iterating through the shallows objects contained in the \text{shallowsArray} array and it is verified if there are interceptions between the line segments that take the role of the sonar scanlines and the ellipses that form the shallows. To facilitate the computation of the intersection point between a line and an ellipse, the coordinate system is centred at the centre of the ellipse, so that
the following system has to be solved in order to obtain the intersection point:

\[
\begin{aligned}
\begin{cases}
y = m \cdot x + q \\
\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1
\end{cases}
\end{aligned}
\]  

(3.17)

where \( m \) is the slope of the line computed as \( m = \frac{\text{scanlineEnd}_y - \text{scanlineStart}_y}{\text{scanlineEnd}_x - \text{scanlineStart}_x} \). \( \text{scanlineEnd} \) and \( \text{scanlineStart} \) are respectively the end point of the scanline and the starting point of the scanline, translated to the new coordinate system;

\( q \) is the the y-intercept of the line \( q = \frac{\text{scanlineStart}_y - \text{scanlineStart}_y}{\text{scanlineStart}_x - \text{scanlineEnd}_x} \).

\( \text{scanlineStart}_x \);

\( a \) is the horizontal semi-axis of the ellipse \( a = \text{width}/2 \), \( \text{width} \) is the ellipse width (i.e. the horizontal axis);

\( b \) is the vertical semi-axis of the ellipse \( b = \text{height}/2 \), \( \text{height} \) is the ellipse height (i.e. the vertical axis);

solving the system equation in \( x \):

\[
\begin{aligned}
\frac{x^2}{a^2} + \frac{(m \cdot x + q)^2}{b^2} &= 1 \\
\Rightarrow \quad \frac{x^2}{a^2} + \frac{m^2x^2 + 2mqx + q^2}{b^2} &= 1 \\
\Rightarrow \quad b^2x^2 + a^2m^2x^2 + 2mqa^2x + a^2q^2 &= a^2b^2 \\
\Rightarrow \quad \left( m^2 + \frac{b^2}{a^2} \right)x^2 + 2mqa^2x + a^2q^2 - b^2 &= 0 \\
\Rightarrow \quad x_{1,2} &= \frac{-mq \pm \sqrt{(mq)^2 - \left( m^2 + \frac{b^2}{a^2} \right) \cdot (q^2 - b^2)}}{m^2 + \frac{b^2}{a^2}}
\end{aligned}
\]

hence

\[
y_{1,2} = m \cdot x_{1,2} + q.
\]

In the case of a scanline parallel to the x axis, the system (3.17) becomes

\[
\begin{aligned}
\begin{cases}
y = y_0 \\
\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1
\end{cases}
\end{aligned}
\]  

(3.18)
\[ x_{1,2} = \pm \frac{b}{a} \cdot \sqrt{b^2 - y_0^2}, \quad y_{1,2} = y_0; \]

in the other case of a scanline parallel to the y axis, such system becomes

\[
\begin{align*}
  x &= x_0 \\
  \frac{x^2}{a^2} + \frac{y^2}{b^2} &= 1
\end{align*}
\]

and \( y_{1,2} = \pm \frac{a}{b} \cdot \sqrt{a^2 - x_0^2}, \quad x_{1,2} = x_0. \)

After this is done, we can compute the distances from the robot position and the interception points using the Pythagorean theorem and see if such distances are inside the scanline length; if a distance is greater than the scanline length the sonar matrix cell for that interception point is considered \( \text{sonarBinsInScan} - 1 \) otherwise it is computed as in (3.16) and all the cells of that scanline between the cells that represent the position of the two intersection points are filled with the value 100.

Of course, as for the boat recognition, there are some particular cases to handle with, for example if both the intersection points are outside the range of the sonar for a particular scanline, the algorithm checks if the starting point of the scanline and/or its end point lie inside the ellipse; to know this fact we consider the line parallel to the x axis that passes through the point we are analysing, we find the intersection points between this line and the ellipse like in (3.18) and control if the x coordinate of the point lies between the x coordinate of the two intersection points. If both the starting point and the end point of the scanline lie inside the ellipse, all the cells of that scanline are marked with the value 100, otherwise if only the starting point lies inside the ellipse we consider filling the cells from 0 to the one of the other intersection point, lastly if only the end point lies inside the ellipse we consider filling the cells from the one of the other intersection point to \( \text{sonarBinsInScan} - 1 \).

### 3.3.2.5 Sea currents and winds forces generator thread

Now let us described how the behaviour of the forces applied to the robot by the sea currents and the winds are simulated. As for the previously mentioned threads, in the main function of the project the thread that simulates the behaviour of the sea currents is created and executed. As it is written in 3.3.2.2 the sea currents and winds forces can
be set up also manually from the user, so the first thing that the currents simulation thread does is controlling if such forces are set up manually, if not it can continue the execution of its code.

1. The \texttt{model\_initialize} function is called and then \texttt{model\_step} is called a random number of time in order to output different intensity and direction of the forces each time the algorithm starts.

2. Now a loop starts and it is executed until the currents generator is set up manually by the user; inside this loop the \texttt{model\_step} function is called once and the difference of its output and the previous one is computed, we will call this difference $\Delta_{int}$ and $\Delta_{dir}$ respectively for the output intensity and for the output direction.

3. The first element of the array of currents \texttt{myCurrents} is filled with the data output from the model for its intensity and direction and its position is set at the point \( P = (133, 133) \) of the map.

4. All the other elements of the \texttt{myCurrents} array are filled with a position each time 133 pixels far from the previous element of the array and are filled with an intensity and a direction that is the result of a random number summed to the intensity or to the direction of the previous element of the array. The intensity of the current is limited between 0 and 4 knots and the direction is limited between 0 and 360 degrees.

5. After all the currents’ data are set up the algorithm computes the intensity and the direction of the currents at the position of the robot. To do this a loop iterating through the currents of \texttt{myCurrents} array is performed and it is checked which are the four currents that surround the robot, (Figure 3.16) and their distances from the robot; say \texttt{arrowUpLeft} the current at the left above the robot, \texttt{arrowUpRight} the current at the right above the robot, \texttt{arrowDownLeft} the current at the left under the robot, \texttt{arrowDownRight} the current at the right under the robot and \texttt{distArrowUpLeft}, \texttt{distArrowUpRight}, \texttt{distArrowDownLeft}, \texttt{distArrowDownRight} their distances from the robot, the following formulae are reckoned in.
order to find the intensity and the direction of the currents at the robot’s position:

\[
\text{distSumInv} = \frac{1}{\text{distArrowUpLeft}} + \frac{1}{\text{distArrowUpRight}} + \frac{1}{\text{distArrowDownRight}} + \frac{1}{\text{distArrowDownLeft}}
\]

\[
\text{currAtRobotPos_{int}} = \left( \frac{\text{arrowUpLeft_{int}}}{\text{distArrowUpLeft}} + \frac{\text{arrowUpRight_{int}}}{\text{distArrowUpRight}} + \frac{\text{arrowDownLeft_{int}}}{\text{distArrowDownLeft}} + \frac{\text{arrowDownRight_{int}}}{\text{distArrowDownRight}} \right) \cdot \frac{1}{\text{distSumInv}}; \quad (3.19)
\]

\[
\text{currAtRobotPos_{dir}} = \left( \frac{\text{arrowUpLeft_{dir}}}{\text{distArrowUpLeft}} + \frac{\text{arrowUpRight_{dir}}}{\text{distArrowUpRight}} + \frac{\text{arrowDownLeft_{dir}}}{\text{distArrowDownLeft}} + \frac{\text{arrowDownRight_{dir}}}{\text{distArrowDownRight}} \right) \cdot \frac{1}{\text{distSumInv}} \quad (3.20)
\]

where \( \text{currAtRobotPos_{int}} \) and \( \text{currAtRobotPos_{dir}} \) are the intensity and the direction of the sea currents at the position of the robot. Here it is computed also the x and y coordinates components of the vector that represents the current thrust on the robot (we will talk about this vector in 3.3.2.7). Such components are reckoned in this way:

\[
\text{vCurrX} = \frac{(\text{currAtRobotPos_{int}} \cdot \sin (\text{currAtRobotPos_{dir}}))}{R}; \quad (3.21)
\]

\[
\text{vCurrY} = -\frac{(\text{currAtRobotPos_{int}} \cdot \cos (\text{currAtRobotPos_{dir}}))}{R} \quad (3.22)
\]

where \( R \) is the ratio between one pixel of the map and one metre.

6. At every current contained in the \texttt{myCurrents} array is set as type the “C” code to
identify them as sea currents, the thread is let sleep for 10 seconds, the intensity and the direction of the current at the first position of the array are summed with a tenth of $\Delta_{int}$ and $\Delta_{dir}$ respectively and the loop begins again from step 4. After 10 iterations the loop exits, the old intensity and direction are updated with the new ones and the outer loop is executed from step 2.

7. If the set up of the currents is done manually the outer loop exits, the \textit{model\_terminate} function is called and the thread every 100 milliseconds checks if the currents set up is become automatic instead of manually in order to restart the loop from step 1.

3.3.2.6 Winds generator thread

The thread that generates the wind intensities and directions automatically works pretty much the same as the one for the sea currents generation.

In the main function of the project the thread that simulates the behaviour of the winds is created and executed. As it is written in 3.3.2.2 the winds can be set up also manually from the user, so the first thing that the winds simulation thread does is

---

Figure 3.16: The robot icon and the four currents icon that surround it.
controlling if such forces are set up manually, if not it can continue the execution of its code.

1. (Same as for the sea currents simulation) The `model_initialize` function is called and then `model_step` is called a random number of time in order to output different intensity and direction of the winds each time the algorithm starts.

2. (Same as for the sea currents simulation) Now a loop starts and it is executed until the winds generator is set up manually by the user; inside this loop the `model_step` function is called once and the difference of its output and the previous one is computed, we will call this difference $\Delta_{int}$ and $\Delta_{dir}$ respectively for the output intensity and for the output direction.

3. (Same as for the sea currents simulation) The first element of the array of winds `myWinds` is filled with the data output from the model for its intensity and direction and its position is set at the point $P = (66.5, 66.5)$ of the map.

4. (Same as for the sea currents simulation) All the other elements of the `myWinds` array are filled with a position each time 133 pixels far from the previous element of the array and are filled with an intensity and a direction that is the result of a random number summed to the intensity or to the direction of the previous element of the array. The intensity of the wind is limited between 0 and 30 knots and the direction is limited between 0 and 360 degrees.

5. After all the winds’ data are set up the algorithm computes the intensity and the direction of the winds at the position of the robot. To do this a loop iterating through the winds of `myWinds` array is performed and it is checked which are the four winds that surround the robot and their distances from the robot; the formulae to compute the data at the robot’s position are the same as for the sea currents generation (3.19, 3.20).

The difference from the other generator is at this point, since the winds are responsible of the creation of waves, here all the properties of the waves are computed as in [5]:

122
\[
\begin{align*}
\text{wave direction} & = (\text{windAtRobotPos}_\text{dir} + 180) \pmod{360}; \\
\text{windAtRobotPos}_\text{MS}_{\text{int}} & = \text{windAtRobotPos}_{\text{int}} \cdot \text{KMS Ratio}; \\
\text{wave amplitude} & = A = \frac{0.21 \cdot \text{windAtRobotPos}_{\text{MS}_{\text{int}}}}{g} \cdot \frac{1}{2}; \\
\text{wave frequency} & = \omega_0 = \frac{0.88 \cdot g}{\text{windAtRobotPos}_{\text{MS}_{\text{int}}}}; \\
\text{wave period} & = T = \frac{2 \cdot \pi}{\text{wave frequency}}
\end{align*}
\]

where \(\text{windAtRobotPos}_\text{dir}\) and \(\text{windAtRobotPos}_{\text{int}}\) are the direction and the intensity of the wind at the robot’s position, \(\text{KMS Ratio}\) is the ratio between one knot and one metre over second that is equal to 0.514444444 and \(g\) is the acceleration of gravity (9.80665 m/s); to compute the length of the wave as in [8]:

\[
\begin{align*}
\text{depth} & = 20\text{(metres)}; \\
G & = \left(\frac{2 \cdot \pi}{\text{wave period}}\right)^2 \cdot \frac{\text{depth}}{g}; \\
F & = G + \frac{1}{1 + 0.6522 \cdot G + 0.4622 \cdot G^2 + 0.0864 \cdot G^4 + 0.0675 \cdot G^5}; \\
\text{wave length} & = \lambda = \text{wave period} \cdot \sqrt{\frac{g \cdot d}{F}}
\end{align*}
\]

6. (Same as for the sea currents simulation) At every wind contained in the \texttt{myWinds} array is set as type the “W” code to identify them as winds, the thread is let sleep for 10 seconds, the intensity and the direction of the wind at the first position of the array are summed with a tenth of \(\Delta_{\text{int}}\) and \(\Delta_{\text{dir}}\) respectively and the loop begins again from step 4. After 10 iterations the loop exits, the old intensity and direction are updated with the new ones and the outer loop is executed from step 2.
7. (Same as for the sea currents simulation) If the set up of the winds is done manually the outer loop exits, the model_terminate function is called and the thread every 100 milliseconds checks if the winds set up is become automatic instead of manually in order to restart the loop from step 1.

3.3.2.7 Client communication and robot’s data update thread

An other important thread launched in the main function is the one that let the graphics thread communicate with the client so that the various data of the robot can be updated. Firstly in this thread a socket is created to communicate with the client; then the data of the robot regarding its position and its angles of rotation are sent onto the socket in order to let the client send them to the navigation algorithm (see section 3.3.1).

After this, an infinite loop is created in which are read from the socket, the data values of the propellers’ speeds, the rudder angle, the velocity of the robot, the rotational velocity and acceleration and the velocity of the robot output from the flowmeter sensor. All these values are stored in the Robot struct and the new position of the robot is computed in this way:

the space covered by the robot is reckoned using the trapezoidal rule of integration [11], knowing the robot’s speed at the previous instant of time \( u_0 \), the robot’s speed at the current instant of time \( u_1 \) and the difference in time between the two instants \( \Delta t \) therefore \( s = \frac{(u_0+u_1)\Delta t}{2} \);

the current angle of heading \( \psi \) is updated in this way, knowing the robot’s rotational speed at the previous instant of time \( r_0 \), the robot’s rotational speed at the current instant of time \( r_1 \) and the difference in time between the two instants \( \Delta t \), using the same trapezoidal rule of integration, \( \psi = \psi_{\text{old}} + \frac{(r_0+r_1)\Delta t}{2} \);

the space covered by the robot along the X and Y axes is so calculated,

\[
\begin{align*}
    s_{\text{Pixel}} & = \frac{s}{R}; \\
    s_{\text{Pixel}_x} & = s_{\text{Pixel}} \cdot \sin(\psi) + v_{\text{Curr}X} \cdot \Delta t; \\
    s_{\text{Pixel}_y} & = -s_{\text{Pixel}} \cdot \cos(\psi) + v_{\text{Curr}Y} \cdot \Delta t
\end{align*}
\]
where $R$ is the ratio between one pixel of the map and one metre, $v_{CurrX}$ and $v_{CurrY}$ are the components of the vector representing the speed of the sea current at the robot’s position in metres over seconds along the X and Y axes evaluated in the formulae 3.21 and 3.22. So the real amount of space covered by the robot including the thrust of the sea currents (used in formula 3.1) is computed as, 

$$s_{Covered} = \sqrt{s_{Pixel_x}^2 + s_{Pixel_y}^2} \cdot R;$$

eventually the actual direction of the robot considering the thrust of the sea currents is calculated as,

$$\Delta x = s_{Pixel_x};$$

$$\Delta y = -s_{Pixel_y};$$

if $\Delta x < 0$

$$robot_{direction} = \left[ 2\pi + \frac{\pi}{2} - \left( \pi + \arctan \left( \frac{\Delta y}{\Delta x} \right) \right) \right] \mod 2\pi$$

else if $\Delta x > 0$

$$robot_{direction} = \left[ 2\pi + \frac{\pi}{2} - \arctan \left( \frac{\Delta y}{\Delta x} \right) \right] \mod 2\pi$$

else

$$robot_{direction} = 0$$

Before starting a new iteration of the loop the values of the robot speeds and angles are sent onto the socket in order to let the client send them again to the navigation software.

3.3.2.8 Maneuver communication thread

This thread is responsible of the communication with the Manager software block in order to send it the moveTo messages, so that the latter software block can send the data included in such message to the Navigation software block that can in turn convey
the commands about the new propellers’ speeds and the rudder’s angle to the simulator client so that the robot can reach the desired position.

Our thread controls if the moveTo message (created in the moveTo popup window, see section 3.3.2.2) is changed, therefore it can be sent; since this communication is connection-less a TCP socket is created every time a message has to be sent and closed after it has been sent. After the socket is closed the thread sleeps for 100 milliseconds and then checks again if the moveTo message has changed, beginning so a new iteration of an infinite loop.

3.3.2.9 Laser data communication thread

To communicate the laser data with the Obstacle Avoidance (OA) software block there is a thread responsible for this. First of all it is checked if the simulator is in the virtual or in the hybrid configuration, for the first one a UDS socket is created otherwise a TCP one is made. Then an handshake between our thread (LDCT) and the OA thread begins:

- OA sends a request to initialise the communication.
- LDCT sends a message stating the it is ready to start the communication.
- OA asks for the laser parameters.
- LDCT answers with a struct filled with such parameters; the struct is the following

```c
typedef struct{
    int num_laser_data; // (value = 1081)
    float angle_res;    // (value = 0.25)
    int countMax;       // (value = 1080)
    int countMin;       // (value = 0)
    float laser_min_angle;    // (value = -135)
    float laser_max_angle;    // (value = 135)
    int count_zero;         // (value = 540)
} laser_data;
```
float max_range; // (value = 60 m)
float range_res; // (value = 0.001)
float scanning_frequency; // (value = 40)
} Laser_Params;

and the variables of such struct are initialised as those depicted in the above listing between round brackets.

- OA asks for the data of the laser.
- LDCT answers with arrLaser, the array of distances computed by the laser thread simulator (see section 3.3.2.3). The loop iteration begins again from the previous point.
4 PROGRAM VALIDATION

To test the program various trials have been performed in order to validate the results. The tests were the following:

- robot navigation via autopilots commands without sea currents;
- robot navigation via autopilots commands with sea currents;
- robot navigation to reach a waypoint without sea currents;
- robot navigation to reach a waypoint with sea currents;
- virtual laser and sonar sensors test without waves;
- virtual laser and sonar sensors test with waves;
- all the above mentioned tests, except the last two, were performed both in the hybrid simulator configuration and in the virtual one.

All the tests were performed on a Acer Aspire 5720 with dual processor at 1.5 GHz and 2GiB of RAM memory. The operating system was a Linux Ubuntu 10.04 (lucid).

4.1 Robot navigation via autopilots commands without sea currents

During this trial the following input data values were defined at the starting of the test:

- Robot’s cruise speed at 1.5 m/s
- Rudder’s angle of the robot at 0°
Robot’s starting heading at $0^\circ$ (from North)

As we can see from Figure 4.1 in which there are the properties panel of the simulator and the path covered by the robot during the first test, the output values after reaching a steady state are exactly 1.5 m/s for the robot speed, the rudder angle is at $0^\circ$, the heading ($\psi$) of the robot is at $0^\circ$ from North and of course its direction too, since there are no currents set to disturb the robot’s movement. We can also notice that the robot moved along a straight path (the red line) while moving and always pointing to North.

![Properties Panel and Path](image)

Figure 4.1: Robot navigation via autopilots commands without sea currents.

During the test were logged the values for the time responses of the program to update the values of the robot properties read from the client side that in turn read them from the autopilots software. The constraint in order to let the software work is to perform this task in 50 ms. The data logged, for the virtual simulation, had an average of 50.41 ms with a maximum peak of 54 ms and a standard deviation of 0.54 ms (Figure 4.2); during the test performed in the hybrid configuration the data were the following: average of 49.99 ms with a maximum peak of 60 ms and a standard deviation of 2.27 ms. These results show that the simulator respects the constraint with a small and non influential error of 0.82% on average, in the virtual mode, and it is almost perfect in the hybrid one.
4.2 Robot navigation via autopilots commands with sea currents

The input data values for the initial state of the simulator were the following:

- Robot’s cruise speed at 1.5 m/s
- Rudder’s angle of the robot at 0°
- Robot’s starting heading at 0° (from North)
- Sea currents set up at a direction of 90° (i.e. from West to East)
- Sea currents set up at a velocity of 2 knots (~1.03 m/s)

In Figure 4.3 there are the properties panel of the simulator and the path covered by the robot during the second test, as for the first test, the output values after reaching a steady state are, for the robot speed, 1.5 m/s, 0° for the rudder angle, the heading (ψ) of the robot is at 0° from North but the actual direction of the robot is at 34.45°, this is due to the effect of the sea currents that moved eastwards the robot while it was trying to go in the northern direction; this fact led the robot to move along a straight line as in the first test, because the currents had a fixed value, but in a diagonal way.

Regarding the time of updating the robot’s value almost the same results of the first trial were achieved; the average during the virtual simulation was 50.39 ms with a maximum peak of 54 ms and a standard deviation of 0.53; for the hybrid configuration
50.02 ms was the average, with a maximum peak of 88 ms and a standard deviation of 3.10.

4.3 Robot navigation to reach a waypoint without sea currents

In this test the user had to set onto the map a waypoint that the robot had to reach. The sea currents were not set up. The following values were outlined:

- Robot’s cruise speed at 1.5 m/s
- Robot’s minimum speed at 1.0 m/s
- Maximum heading speed 0 °/s

As one can see from Figure 4.4 the robot followed the shortest path from its starting point to the goal point it had to reach, the cruise speed of 1.5 m/s was respected and it did not go under the 1 m/s until the motors stopped after the goal waypoint was
reached; the angle of the rudder during the displacement of the robot at a steady state varied less than 1° and its direction changed between 24° and 26°.

![Figure 4.4: Robot navigation to reach a waypoint without sea currents.](image)

The update times were comparable to the other two trials previously performed: for the virtual configuration, 50.39 ms on average with peaks of 56 ms and a standard deviation of 0.60; for the hybrid one the values were almost the same as in the previous trial with a slightly higher standard deviation of 4.04 ms.

### 4.4 Robot navigation to reach a waypoint with sea currents

Again for this test the user had to set onto the map a waypoint that the robot had to reach. The sea currents were set up too. The following values were outlined:

- Robot’s cruise speed at 1.5 m/s
- Robot’s minimum speed at 1.0 m/s
- Maximum heading speed 0°/s
- Sea currents set up at a direction of 90° (i.e. from west to east)
- Sea currents set up at a velocity of 2 knots (~1.03 m/s)

Analysing the data expressed in the table shown in Figure 4.5, at steady state the robot travelled correctly at 1.5 m/s and its direction was towards the waypoint (i.e. at 236.09°) meanwhile the heading ($\psi$) was at 258.58°, this difference is due to the effect of the sea currents on the motion of the robot. The path covered was not very straight especially in the beginning because the robot’s speed was much minor respect to the current’s speed so it was dragged for a while until it reached a higher speed in order to contrast the sea current. After the waypoint was reached the motors of the propellers stopped and the robot was dragged again by the currents in the east direction.

![Figure 4.5: Robot navigation to reach a waypoint with sea currents.](image)

Also in this test the time values were very similar to the previous ones: 50.40 ms on average with peaks of 55 ms and a standard deviation of 0.54, in the case of virtual simulation; the same results as the previous test in the case of hybrid configuration.
4.5 Virtual laser and sonar sensors test without waves

For this trial it has been decided to do not let the robot move, so its speed was set up at 0 m/s, the rudder’s angle of the robot at 0°, the robot’s starting heading at 0° (from North). Besides the constraint of the 50 ms for each update of the robot's data, for this test there were other two of them regarding the sensors we had to test. In order to simulate correctly the response times of the real sensors, the laser sensor had to be updated every 250 ms meanwhile for the sonar sensor the update time had to be in function of the range of the angle it was set to “see” and the length and the number of the beams; in our case the beams were 100 with length of 25 metres each and an angle magnitude of 180°, from this input values the sonar response time was computed as 3333 ms.

As mentioned in chapters 3.3.2.3 and 3.3.2.4 if the time elapsed for the computation is minor than the real sensor update time the difference is slept by the thread.

To analyse the update time for the sensors there have been added boats to the map and investigated the maximum number of boats present in order to let the sensor simulators obtain the desired results under the constraint times. This can be achieved looking at Figure 4.6 and 4.7 for the laser sensor and the sonar sensor respectively.

![Figure 4.6: Laser update time versus number of boats.](image)

For the laser sensor the update time increases linearly in the number of the boats and reaches the limit of 250 ms at about 42 boats added on the map. This increase of
the update time is due to the laser simulator algorithm described in 3.3.2.3 that has to analyse all the boats’ borders in order to compute the possible intersections with the laser rays.

Figure 4.7: Sonar update time versus number of boats.

Also for the sonar sensor the update times increase almost linearly in the number of the boats added to the map, but the values are much inferior than the constraint time for this type of sensor; the maximum reached is 65 ms.

4.6 Virtual laser and sonar sensors test with waves

This test was performed with the same input values as the previous one i.e.:

- robot’s speed at 0 m/s
- the rudder’s angle of the robot at 0°
- the robot’s starting heading at 0° (from North)

Furthermore the wind was manually set up by the user in order to have a constant value for it and therefore a constant sea wave. The data of the wind were the following:

- 18 knots of intensity
- 0° of direction (i.e. from South to North)
Consequently the sea wave generated by such wind had the following properties:

- period of 6.7 seconds
- amplitude of 0.92 metres
- length of 67.64 metres

In Figure 4.8 we can see at the results for the update time of the laser sensor. This time the update times are slightly higher than in the previous test for each boat added to the map, this is caused by the added amount of time the algorithm had to spend computing the interceptions between the laser rays and the sea wave. We can notice that the constraint of 250 ms is respected until about 38 boats have been added to the map.

![Figure 4.8: Laser update time versus number of boats (with waves).](image)

Regarding the sonar update time we can consider Figure 4.9 for the results of this test. We can notice that the behaviour of the update time function is pretty similar to the one of Figure 4.7 (i.e. without the presence of waves). In fact as stated in chapter 3.3.2.4 waves do not have any effect on the sonar behaviour.

### 4.7 Results discussion

All the trials led to good results especially in the simpler environmental conditions as without the presence of sea currents, winds or obstacles. The results were acceptable...
also in the cases of movements with sea currents included in the simulation though. The only cases in which the simulator does not respect the constraints is when using the sensors simulators, in particular the laser, and a big amount of objects are added onto the map; as we saw 42 in the case of absence of waves, 38, waves simulation included. Generally speaking a normal usage of the simulator imply the use of no more than 20 objects added to the map, in this case the simulator respects easily the time constraints.

During all these tests the RAM memory occupation and the CPU load of work have been monitored. Eventually an average of about 50 MiB of RAM was occupied by the server process, meanwhile just 180 KiB were occupied by the client process. Concerning the CPU usage the server process reached an average of 20% of the maximum work-load and the client process less than 1%.

This values show that the application could be run on almost all modern computers, even if it may slow down the execution of other programs running on the same machine especially if they make use of a great amount of the RAM memory and CPU resources.

Figure 4.9: Sonar update time versus number of boats (with waves).
5 CONCLUSIONS

During the development of this thesis I had to realise a graphical simulator whose function was to emulate the environment of the working area of a marine robot and the behaviour of the dynamics of the robot itself.

In order to elaborate the physical model of the robot the MATLAB framework was used among some of its various toolkits like Simulink and the Real Time Workshop. Meanwhile to create the graphical interface the Qt graphical libraries, based on the C++ programming language, were utilised.

The simulator can be used in two different configurations:

- the hybrid configuration,
- the virtual configuration.

The first one lets the simulator read and use some real data of the robot and send data to it, in order to test not only the virtual responses to the simulated environment but also to control the right functioning of the robot’s hardware.

In the second configuration the simulator communicates only with other software processes that usually run on the robot’s microprocessor but in this case they run on the same PC where the simulator is running. This is a completely safe environment thus the simulated robot’s dynamics can be stressed and tested very accurately without any problem of damaging the robot’s devices. Furthermore the correctness of the software modules like the Obstacle Avoidance or the Navigation algorithms can be tried out.

The simulator has many options to emulate a lot of different kind of situations the robot may encounter in the real environment. There is the possibility to set up sea currents up to 4 knots; the user can set up winds up to 30 knots in order to create
realistic waves; furthermore the user can even create his or her own sea waves changing their fundamental characteristics (i.e. their period, amplitude and length). These options lead to a complete flexible simulator that could be customised by the user in a very easy way to perform many kind of tests in different ways, in a simulated marine environment with many different characteristics between each test.

Besides these features the simulator lets the user add moving objects on the map (i.e. boats) and underwater obstacles like shallows, to control the correct behaviour of the robot’s sensors (laser and sonar) and the obstacle avoidance algorithm.

The simulator lets the user set waypoints the robot have to reach to test the navigation algorithm developed for the robot.

As we have seen in the program validations, the simulator responds satisfactorily to the time constraint it has to respect, at least until a number of around 40 obstacles are added to the map, but for a normal use of the simulator (i.e. adding around 10-20) boats and/or shallows the simulator respect such constraints perfectly.

As future developments the simulator could be extended by letting the user add more kind of objects onto the map in order to emulate in a even more realistic way the marine environment; the dynamics of the robot could be easily improved or changed, since the Simulink model besides the Real Time Workshop let these kind of expansions be achieved investing just a little effort and time; for example if necessary could be even added the simulation of the hardware errors or failures of the robot’s devices; the emulation of the wind’s physics for the robot’s sail could be implemented in the MATLAB framework and therefore easily added as a feature to the graphical simulation.
Bibliography


**Online material**


[34] http://www.cyberbotics.com/