Deliverable D 2.1

Release of the core simulator providing generic functionalities for cooperative underwater robotics compliant with the requirements for the project according to D1.1

Contract number: FP7- 231378 Co3 AUVs

Cooperative Cognitive Control for Autonomous Underwater Vehicles

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number 231378.
**Identification sheet**

<table>
<thead>
<tr>
<th>Project ref. no.</th>
<th>FP7-231378</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>Co3AUVs</td>
</tr>
<tr>
<td>Status &amp; version</td>
<td>[Final] &quot;1.0&quot;</td>
</tr>
<tr>
<td>Contractual date of delivery</td>
<td>month 14</td>
</tr>
<tr>
<td>Actual date of delivery</td>
<td>March 2011</td>
</tr>
<tr>
<td>Deliverable number</td>
<td>D 7.2</td>
</tr>
<tr>
<td>Deliverable title</td>
<td>Release of the core simulator providing generic functionalities for cooperative underwater robotics compliant with the requirements for the project according to D1.1</td>
</tr>
<tr>
<td>Nature</td>
<td>Report</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU</td>
</tr>
<tr>
<td>WP contributing to the deliverable</td>
<td>WP2</td>
</tr>
<tr>
<td>WP / Task responsible</td>
<td>Jacobs</td>
</tr>
<tr>
<td>Editor</td>
<td>Andreas Birk</td>
</tr>
<tr>
<td>Editor address</td>
<td>Jacobs University Robotics, School of Engineering and Science Cmapus Ring 1, 28759 Bremen, Germany</td>
</tr>
<tr>
<td>EC Project Officer</td>
<td>Franco Mastroddi</td>
</tr>
<tr>
<td>Keywords</td>
<td>marine robotics, simulation, visualization</td>
</tr>
</tbody>
</table>

**Abstract (for dissemination):** There is a vastly growing interest in the use of Autonomous Underwater Vehicles (AUV) in complex mission scenarios. But there are still many open research problems and challenges. Also, there are in general high risks and potentially severe consequences of mission failures for many underwater applications. It is hence of high interest to use simulators to test and develop marine robots. The Co\(^3\)-AUv\(^s\) simulator is designed to support research on the cooperative and cognitive side of marine robotics, i.e., it is laid out to simulate different vehicles with rich sensor payloads in complex 3D scenarios. The simulator is implemented using the Bullet Physics Library as the physics engine and the Object-Oriented Graphics Rendering Engine (OGRE) as rendering engine.

**List of annexes (if any)**

- Simulator v 1.0 Overview
- Interfacing the Simulator with the Robotics Operating System (ROS)
- Example exercises (PID control, 2D mapping) to get started with the simulator
Co$^3$-AUV Simulator
version 1.0

Ravi Rathnam, Alexandru Ichim, Andreas Birk
email: eecs-robotics@jacobs-university.de

November 2010
# Contents

1 Co$^3$-AUV Simulator ................................................. 3
  1.1 Overview ....................................................... 3
    1.1.1 Physics World ........................................... 4
    1.1.2 Physics World Listeners ................................. 4
    1.1.3 Sensor World Listeners ................................. 4
  1.2 Coordinate System ............................................ 4
  1.3 Requirements .................................................. 4
    1.3.1 Bullet-2.76 .............................................. 5
    1.3.2 Ogre 1.7 ................................................ 5
  1.4 How To Use the Simulator ...................................... 5
    1.4.1 How to build ............................................ 5
    1.4.2 How to run ............................................... 6
    1.4.3 Config file sections .................................... 6
    1.4.4 World File ............................................... 6
    1.4.5 How to interact with the simulator ..................... 7
    1.4.6 How to use the Graphical User Interface ............... 8
      1.4.6.1 How to Navigate .................................... 8
      1.4.6.2 Changing the settings of the view and camera .... 8
    1.4.7 How to Create a New Robot Model ....................... 9
    1.4.8 How to Create a New World from a Heightmap Image ... 9
      1.4.8.1 Relation between Heightmap and World coordinates 10
  1.5 Directory Structure ............................................ 10
    1.5.1 PhysicsWorld ............................................ 11
    1.5.2 BtOgreConversions ...................................... 11
  1.6 Sensors ........................................................ 11
    1.6.1 OdometrySensor ......................................... 12
    1.6.2 Range2DSensor ......................................... 13
    1.6.3 Range3DSensor ......................................... 14
  1.7 Actuators ....................................................... 14
    1.7.1 Thruster ................................................ 15
  1.8 Class Descriptions ............................................. 15
    1.8.1 PhysicsWorld ............................................ 16
      1.8.1.1 BulletWorld ....................................... 16
      1.8.1.2 PhysicsWorldListener .............................. 16
      1.8.1.3 BufferedPhysicsWorldListener ..................... 16
      1.8.1.4 RobotManager ...................................... 16
      1.8.1.5 Vehicle ............................................ 16
      1.8.1.6 UnderwaterVehicle .................................. 17
      1.8.1.7 BulletObjectsLibrary .............................. 17
    1.8.2 SensorWorld .............................................. 17
      1.8.2.1 SensorWorld ....................................... 17
      1.8.2.2 SensorWorldListener .............................. 17
      1.8.2.3 OdometrySensorWorld .............................. 17
      1.8.2.4 RangeSensorWorld .................................. 18
      1.8.2.5 SensorWorldFactory ............................... 18
    1.8.3 Networking ................................................ 19
Chapter 1

Co$^3$-AUV Simulator

![Screenshot of the simulator](image)

Figure 1.1: Screenshot of the simulator

1.1 Overview

The basic structure of the simulator is shown in figure 1.2. The simulator uses a modular design and the different components are decoupled. Interaction between different components is implemented using the listener pattern. Listeners are registered to the different components at the startup. Whenever, changes are made to a component, it propagates the changes by calling the appropriate function on all registered listeners.
1.1.1 Physics World

The physics world is the core physics simulation. This is implemented using the Bullet Physics Engine. It contains the world, including static and dynamic components (such as vehicles). Whenever, any change occurs in the Physics World, these changes are propagated to the outside using the callbacks on the registered listeners. These changes may include external (such as adding objects to the world), or movement (caused by the dynamics of the world).

1.1.2 Physics World Listeners

As stated above, the changes in the Physics World are propagated using Physics World Listeners. Physics World Listeners are any components that need to be updated about changes in the Physics World. This includes sensor, as well as any visualisation tool that are needed to view the world.

1.1.3 Sensor World Listeners

Sensor World Listeners are analogous to Physics World Listeners, but are meant to for transferring of sensor data. When sensor data is updated, the data is then propagated using the registered Sensor World Listeners. These include any components that require sensor data, including networking components, as well as any viewers of sensor data.

1.2 Coordinate System

The simulator uses the Right-Handed Cartesian Coordinate System. The +X axis points to the front, the +Y axis to the right, and +Z axis to the bottom.

1.3 Requirements

1. Bullet-2.76
2. OGRE 1.7
1.3.1 Bullet-2.76

Bullet-2.76 must be compiled using the cmake system with the following options. Read Bullet installation documentation on how to set the following parameters.

- BUILD_EXTRAS should be set to true
- INSTALL_LIBS should be set to true
- INSTALL_EXTRA_LIBS should be set to true
- USE_DOUBLE_PRECISION should be set to false

After setting these options and installing Bullet, Bullet will be installed on the system. However, the Extras libraries and sources are not installed. This issue has been submitted to the bug developers. To install the required libraries for the simulator follow the following steps.

1. Copy libBulletWorldImporter.a and libBulletFileLoader.a into the bullet install directory. Typically, this should be done with the following commands from the Bullet build directory.

   (a) `cp Extras/Serialize/BulletWorldImporter/libBulletWorldImporter.a /usr/local/lib`
   (b) `cp Extras/Serialize/BulletFileLoader/libBulletFileLoader.a /usr/local/lib`

2. Copy the required header files into the Bullet include directory. Typically, this should be done with the following commands from the Bullet source directory.

   (a) `cp -r Extras /usr/local/include/`

1.3.2 Ogre 1.7

Before installing OGRE-1.7 ensure that the following packages are already installed on your machine.

1. FreeImage-Dev
2. ZZip

After installing these packages, proceed to installing OGRE.

1.4 How To Use the Simulator

1.4.1 How to build

1. Extract the archive provided
2. If Bullet was installed in a location other than /usr/local/lib and /usr/local/include, edit the file /lib/ external/bullet_extras/Makefile.in
3. Go to the folder app/Co3AUVSimulator/main
4. make

The compiled binary should be in app/Co3AUVSimulator/bin

In addition to the main simulator program, the archive also contains a tool to convert OGRE meshes to Bullet collision meshes. These can then be used in the RobotConfigurations section described below (See Section 1.4.3). To compile the converter,

1. Go to the folder app/Co3AUVSimulator/BtOgreConversions/converter
2. make
1.4.2 How to run

The simulator has to be run with the following syntax from app/Co3AUVSimulator/bin directory.

```
./Co3AUVSimulator configfile worldfile
```

A sample configuration and world file are provided in the archive. You can run the simulator with the following command.

```
./Co3AUVSimulator ../resources/config.cfg
../resources/WorldFiles/terrainTitanicWorld.cfg
```

Alternately, if you have a good graphics card and have compiled the cgProgramRenderer plugin with OGRE, you can run

```
./Co3AUVSimulator ../resources/config_new.cfg
../resources/WorldFiles/terrainTitanicWorldNew.cfg
```

To quit the simulator, you can press ‘q’ on the command line. **Closing the GUI does not cause the simulator to stop!**

When the simulator starts, you may be see a blue screen. This is because the terrain cannot be seen in the view. Move the camera back, and you should be able to see the terrain.

1.4.3 Config file sections

The simulator uses libconfig for its configuration files. The configuration file provided with the simulator has the following section.

**RobotConfigurations** The *RobotConfigurations* section shows the list of robot configurations. A robot configuration consists of information about the robot such as its collision mesh(to be used by the physics) as well as the OgreMesh(to be used by the visualisation). In addition to this, each robot configuration has the list of sensors present on the robot. This is in the *Sensors* sections of the robot configuration. The configuration of the individual sensors is described in the Sensors section(Section 1.6) of this document. In addition to sensors, there is also a section for actuators. The configuration of the actuators is defined in the Actuators section (Section 1.7) of this document.

**SensorWorlds** The *SensorWorlds* section describes the sensor worlds that are present. All *SensorWorlds* referenced by the *Sensors* section of the *RobotConfigurations* must be defined in the *SensorWorlds* section. See Section 1.1 for details on the structure of the simulator.

1.4.4 World File

The World File is also a configuration file which describes the world to be used by the simulator. World files describe a terrain as well as a list of static objects present. Sample world files are provided in the resources/WorldFiles folder with the simulator. All parameters in the world file are part of the *StaticWorld* setting. The following parameters need to be set in the world file.

**bulletWorldFile** The .bullet file containing the collision shape of the world

**ogreTerrainFile** The visualization of the terrain of the world. This is a terrain file that is used by OGRE.

**createOceanVolume** Whether to create an ocean volume. When this is set to true, a box is created for the ocean surface, and encloses an area. Note: This is purely for visual effects. On machines without a graphics card, this should be set to false.

**oceanLength** The length of box for the ocean volume. Only used if createOceanVolume is true.

**oceanWidth** The width of the box for the ocean volume. Only used if createOceanVolume is true.

**oceanSurfaceHeight** The height at which the ocean surface should be created. The volume below the height is considered to be underwater.
**createFog**  Whether fog should be created. If set to true, a distance fog is created in the visualisation.

**fogColor**  The colour of the fog. Only used if createFog is set to true.

**fogStart**  The distance from the camera at which the fog starts having an affect

**fogEnd**  The distance from the camera beyond which nothing is visible.

**terrainWorldSize**  This is the size in world coordinates of the terrain being used

**terrainImageSize**  This is the size of the heightmap image in pixels

**useOldTerrain**  This parameter is set to true, if the ogre terrain uses the terrain system introduced in OGRE 1.7. This new terrain system should only be used in computers with good graphics cards and the CgProgramRenderer plugin is compiled.

**objects**  This contains the list of objects that should be added to the world. Each object is defined by a name which is unique. Each object must have the following parameters set.

  - **ogreMeshName**  The mesh to be used for visualisation
  - **bulletCollisionFile**  The file to be used for the bullet collision model
  - **bulletCollisionName**  The name of the collision model in the file
  - **position**  The position of the object in the world.
    - **x**  The x position
    - **y**  The y position
    - **z**  The z position
  - **orientation**  The rotation of the object
    - **rotx**  The rotation around the x axis
    - **roty**  The rotation around the y axis
    - **rotz**  The rotation around the z axis

### 1.4.5 How to interact with the simulator

The simulator can be communicated with using a simple TCP connection and using a text based message format. All commands and messages are terminated by a carriage return and newline character(`\r\n`). By default, the simulator listens on port 3000 for connections. To spawn a robot, use the following command:

```
INIT {Name robotName} {ClassName robotConfigurationName} {Location x,y,z} {Rotation rx,ry,rz}\r\n
```

**Name**  The name of the robot should not have any spaces.

**ClassName**  The ClassName should be defined in the RobotConfigurations section in the config file.

**Location**  The location is in meters

**Rotation**  The rotation is in radians.

When the simulator receives this command, it spawns a robot with the given configuration, and name at the given location and rotation. After that, the data from the sensors is transferred through the TCP connection. The format of the individual sensor messages is described in Section 1.6.

The robots can be driven either with velocity control or force control using the DRIVE command. The format for the drive command is the following:

```
DRIVE {LocalVelocity x,y,z} {LocalAngularVelocity rx,ry,rz}\r\n
```

**LocalVelocity**  The velocity component is in the local frame of the robot.
LocalAngularVelocity  The angular velocity in the local frame of the robot.

\[
\text{DRIVE } \{\text{LocalForce } x,y,z\} \ {\text{LocalTorque } rx,ry,rz}\n\]

LocalForce   The force in the local frame of the robot, acting on the center of mass

LocalTorque   The torque in the local frame of the robot, acting on the center of mass

1.4.6 How to use the Graphical User Interface

The Graphical User Interface allows users to view the world, as well as sensor data. When a robot connects to the simulator, it is added on the list of robots seen on the right of the screen. The list of sensor data available is also shown below the robot. Double clicking on the robot name causes the camera to move relative to the robot, so as to follow the robot. To go back to free moving mode, click on the Reset Camera button. Double clicking on the sensor name, causes the widget connected to the sensor data to be shown.

1.4.6.1 How to Navigate

To navigate in the world, click on the navigation buttons below the scene, and click and drag the mouse inside the scene. Each button performs different navigation manoeuvres for the camera, depending on the mouse movement.

FlyButton Dragging the mouse forward and backwards causes the camera to move forwards and backwards in the world. Dragging the mouse sideways causes the camera to rotate in place.

PanButton Dragging the mouse forward and backwards causes the camera to pitch up and down in place. Dragging the mouse sideways causes the camera to rotate sideways.

SlideButton Dragging the mouse forward and backwards causes the camera to slide up and down in the world. Dragging the mouse sideways causes the camera to slide sideways in the world.

1.4.6.2 Changing the settings of the view and camera

Visual parameters of the world can be changed using the Viewer Preferences Widget which can be accessed from Edit->Scene Preferences.

The following parameters can be set:

Linear Fog End   This defines the distance beyone which objects are no longer visible completely shrouded in fog.

Linear Fog Start   This defines the distance from the camera from which the fog starts having an effect.

Near Clip Distance   Objects closer than this are not drawn on the screen.

Far Clip Distance   Objects beyond this distance from the camera are not drawn.

Camera Speed   This changes the speed of the movement of the camera. Changing this scales the movements of the camera when navigating in the world.
1.4.7 How to Create a New Robot Model

The robot model should follow the same coordinate system as the simulator, i.e. +X axis pointing towards the front of the robot, +Y axis pointing towards the right of the robot, and the +Z axis pointing towards the bottom of the chassis.

1. Firstly create the robot model according to the coordinate system in the modeling tool of your choice (e.g. Blender, 3dsMax), such that the center of mass is at (0,0,0)
2. Export to Ogre.mesh file format.
   (a) The Ogre exporter provided with Blender does not export the ambient component properly. Edit the material file and copy the diffuse values into the ambient component of the material.
3. Copy the files created into resources/OgreResources
4. Go to the folder app/Co3AUVSimulator/BtOgreConversions/converter
   (a) compile
   (b) run the following command
      ```bash
      ./btOgreConverter ogreMeshFileName bulletFileName false bulletCollisionName
      ```
5. Copy the ogreMesh file created into resources/OgreResources
6. Copy the bullet file into resources/BulletResources

1.4.8 How to Create a New World from a Heightmap Image

The World Creator program is present the BtOgreConversions/worldCreator folder. The WorldCreator program allows to create a heightmap/terrain from a grayscale image. There are two options for creating the visualisation for the world. If the computer has a good graphics card and the OGRE CgProgramManager plugin is compiled, you can use the config_new file provided. Otherwise, use the config file provided. The following option has to be set for the world.

imageFile The heightmap image. It should be a grayscale image.
bulletWorldFile The bullet file that is created
terrainFile The ogre file that is created
terrainWorldSize The size of the terrain in world units, i.e. meters
maxHeight The maximum height of the terrain. This corresponds the white on the image file.
maxPixelError This parameter is only used for visualisation.
useOldTerrain Whether to use the new terrain system introduced in OGRE 1.7. This should only be set to false if used on a good graphics card, and CgProgramRenderer plugin compiled.
worldFile The name of the world file to be created. The world file can then be used as the second argument to the simulator.

Run the World Creator with the following command
```bash
./worldCreator configFile
```
After running the command, there are 3 files which are created. The bullet world file, the ogre terrain file, and the world file. Copy these files into the resources/WorldFiles folder.
1.4.8.1 Relation between Heightmap and World coordinates

The world is created such that black pixels correspond to lowest height(maximum depth), and white pixels correspond to maximum height(minimum depth). The difference in heights between the minimum and maximum depth is given by the \( \text{maxHeight} \) parameter in the config file. To convert from image to a world terrain, the following steps are performed:

1. A trimesh is created such that \(+x\) on the image corresponds to \(+x\) in the world, \(+y\) on the image corresponds to \(+y\) on the world. The \( z \) coordinate of the vertex is taken to be the value of the pixel.

2. The trimesh is then scaled by \( \text{terrainWorldSize}/\text{imageSize} \) on the \( x \) and \( y \) axis. The \( z \) axis is scaled by \( \text{maxHeight}/255 \).

3. The trimesh is then rotated by 180 degrees around the \( x \) axis, and then translated along the local \( y \) axis by the world size.

Therefore, a black pixel on the top left corner of the image corresponds to \((0,0,0)\) in world coordinates. An increase in the \( x \) axis on the image corresponds to traveling along the \( x \) axis in world coordinates. An increase in the \( y \) coordinates in the image corresponds to increase in the \( y \) axis in the world. This is easily seen in figure 1.4. Also all points corresponding to white pixels occur have a \( z \) coordinate of \(-\text{maxHeight}\), and all points corresponding to black pixels have \( z \) coordinate 0.

1.5 Directory Structure

The directories in the application are

1. PhysicsWorld
2. SensorWorld
3. main
4. BtOgreConversions
5. networking

The directories are described below.
1.5.1 PhysicsWorld

This is the core of the Simulator. The classes in this folder comprise the Physics Simulation, and all classes that are needed to interact with the simulator. The classes included are

1. BulletWorld
2. RobotManager
3. PhysicsWorldListener
4. BufferedPhysicsWorldListener
5. Vehicle

The classes are described in the next section.

1.5.2 BtOgreConversions

The classes in this folder are required to convert between transforms, vectors, quaternions between OGRE, Bullet, and the jacobs_robotics namespace. In addition to this, it has a tool in the converter folder, which can be used to convert Ogre meshes to .bullet files. After making the OGRE mesh, using Blender or similar tools, it is required to convert the mesh into a bullet file. This can be done using the BtOgreConverter program provided in this folder. The program should be executed as follows.

```
./BtOgreConverter ogreMeshName bulletFileName bmakeTriMesh CollisionName
```

- **ogreMeshName** is the name of the .mesh OGRE File.
- **bulletFileName** is the name of the bulletFile
- **bMakeTriMesh** is set to true if the OGRE mesh should be converted to a triangle mesh. Otherwise it is converted to a convex hull. For dynamic objects the bmakeTriMesh must be set to “false”. For static objects, it can be set to “true”. Collisions to convex hulls is significantly faster than collisions to triangle meshes.
- **CollisionName** This is the name of the collision object. This is used by the Physics world to load the correct collision object from the file.

1.6 Sensors

The sensors in Co3AUUVSimulator are all present in SensorWorlds. The sensor world type is determined by the SensorWorld where they are used. Each sensor has a few common parameters, such as the position, and orientation with respect to the robot base, as well as the rate of scanning. The sensors that a robot is spawned with is dependent on the robot configuration in the configuration file. A typical robot configurations section in the config file looks like
1.6.1 OdometrySensor

The odometry sensor provides information on the pose of the robot. It currently gives the global pose in world units, i.e., meters and radians. The parameters that can be configured for the sensor include its position with respect to the robot base, and its rate of data acquisition. Its section in the configuration file looks like this:

```
OpenFrame1:
    collisionName="OpenFrame";
    collisionFile="OpenFrame.bullet";
    OgreMesh="OpenFrame.mesh";
    mass=10.0;
    Sensors:
        Odometry:
            sensorWorldName="OdometrySensorWorldA";
            OgreMesh="SensorCube.mesh";
            x=0.0;
            y=0.0;
            z=0.0;
            roll=0.0;
            pitch=0.0;
            yaw=0.0;
            scanIntervalMSec=1000;
        RangeSensor:
            sensorWorldName="OdometrySensorWorldA";
            OgreMesh="SensorCube.mesh";
            x=0.0;
            y=0.0;
            z=0.0;
            roll=0.0;
            pitch=0.0;
            yaw=0.0;
            scanIntervalMSec=1000;
            SensorSpecificparameters
    }
}

OpenFrame2:
    collisionName="OpenFrameScaled";
    collisionFile="OpenFrameScaled.bullet";
    OgreMesh="OpenFrameScaled.mesh";
    mass=10.0;
    Sensors:
        Odometry:
            sensorWorldName="OdometrySensorWorldA";
            OgreMesh="SensorCube.mesh";
            x=0.0;
            y=0.0;
            z=0.0;
            roll=0.0;
            pitch=0.0;
            yaw=0.0;
            scanIntervalMSec=500;
```
like:

```cpp
Odometry:{ //the sensor name
    sensorWorldName="OdometrySensorWorldA";
    OgreMesh="SensorCube.mesh";
    x=0.0;
    y=0.0;
    z=0.0;
    roll=0.0;
    pitch=0.0;
    yaw=0.0;
    scanIntervalMSec=500;
};
```

The message format from the sensor is the following:

```
SEN {Type INS} {Name sensorName} {Location x,y,z} {Orientation roll,pitch,yaw} \n```

### 1.6.2 Range2DSensor

The Range2DSensor implements a 2D range sensor similar to a rotating beam range sensor. The X axis of the sensor points to the front, and the sensor rotates from 
\(-\text{fieldOfView}/2\) to \(+\text{fieldOfView}/2\). The ranges or points returned by the sensor are in sensor coordinates. In addition to the base sensor parameters, the following parameters are:

- **maxRange** The maximum range of the sensor. If no objects are encountered within the maxRange, the sensor returns the range as 0.
- **bUsePointClouds** Whether to give the point cloud or the ranges. This is explained below
- **horizontalFieldOfView** The horizontal field of view in radians
- **noBeamsHorizontal** The number of beams of the robot

A standard config file section of this sensor looks like:

```cpp
Scanner1:{
    sensorWorldName="RangeSensorWorldA";
    RangeSensorType="2DRangeSensor";
    OgreMesh="SensorCube.mesh";
    x=0.6;
    y=0.0;
    z=0.0;
    roll=0.0;
    pitch=0.0;
    yaw=0.0;
    maxRange=20.0;
    bUsePointClouds=false;
    scanIntervalMSec=100;
    horizontalFOV=3.141159;
    noBeamsHorizontal=181;
};
```

If the sensor is set to use point clouds then the message looks like:

```
SEN {Type PointCloudSensor} {Name sensorName} {Points x,y,z;x,y,z;.....} \n```
If the sensor is set to not use point clouds then the message looks like

```
SEN{Type RangeScanner} {Name sensorName} {Resolution res} {FOV fov} {Range d1,d2...,dn}
```

### 1.6.3 Range3DSensor

This is a 3D range Sensor which rotates its scanning beam along two axes. Therefore, in addition to the parameters of the RangeSensor2D, the parameters that can be set in this sensor are:

- **verticalFOV** the vertical field of view
- **noBeamsVertical** the vertical number of beams.

A typical config file section for a RangeSensor3D is

```json
RangeSensor:
    sensorWorldName="RangeSensorWorld1";
    RangeSensorType="3DRangeSensor";
    OgreMesh="SensorCube.mesh";
    x=0.0;
    y=-0.3;
    z=0.0;
    roll=0.0;
    pitch=0.0;
    yaw=0.0;
    maxRange=40.0;
    bUsePointClouds=true;
    scanIntervalMSec=200;
    horizontalFOV=1.72;
    noBeamsHorizontal=30;
    verticalFOV=0.57;
    noBeamsVertical=30;
```

The message format for this format is the same as the RangeSensor2D when the bUsePointClouds is set to true. However, when the bUsePointClouds is set to false the message format is

```
SEN {Type RangeScanner3D} {Name sensorName} {HorizontalFieldOfView hFOV} {numberOfBeamsHorizontal n1} {VerticalFieldOfView vFOV} {numberOfBeamsVertical n2} {Ranges d1,d2,..dn}
```

### 1.7 Actuators

In addition to sensors, actuators can also be mounted on the robots. Actuators can be controlled by sending commands in the network interface. Actuators commands are of the type

```
SET {ActuatorName act1 ParameterName par1 ParameterValue val1} {ActuatorName act2 ParameterName par2 ParameterValue val2}
```

Currently, thrusters are the only actuators present in the simulator.
1.7.1 Thruster

Thrusters are actuators which generate a force and turning torque from the position they are mounted. Each robot has a list of thrusters that are mounted in the config file. The section for thrusters in the configuration of the robot in the config file looks like the following:

```plaintext
Thrusters:

   // For computation of thrust from positive voltage.
   forward:
       { //ko+k1*v=thrust ,
         gamma=0.0130;
         kth=5.3558e-04;
         kq=6.9882e-06;
         kappa_0=-1.0212;
         kappa_1=0.8272;
     };

   // For computation of thrust from negative voltage.
   reverse:
       {  
         gamma=0.0136;
         kth=4.3392e-04;
         kq=5.9056e-06;
         kappa_0=-1.2146;
         kappa_1=0.8068;
     };

   // List all thrusters one by one below:
   // The locations are from O_A not G.
   list:
      ( 
         {  
           name="front_left";
           r_OA_P=[0.50, -0.27, 0.28];
           n=[1.0, 0.0, 0.0];
           sigma=-1; //left handed or right handed
         },
      );
```

The `forward` section shows the values when a positive voltage is applied. The `reverse` section shows the values when the negative voltage is applied. The list section shows the list of sensors that are mounted on the robot. Each element in the list has the following parameters:

- **name** The name of the thruster. Must be unique for the robot configuration.
- **r_OA_P** The position of the thruster with respect to OA as defined in the robot configuration.
- **n** The direction in which the thrust is applied when the voltage is positive.
- **sigma** is -1 when the motor is left handed, and +1 when motor is right handed.

The voltage supplied to the thruster can be set through the network using the following message format:

```
SET {ActuatorName thruster1 ParameterName Voltage ParameterValue val}
```

Depending on the voltage, the thrust generated by the vector can be shown as:

```
thrust = \kappa_0 + \kappa_1 \times voltage
```

1.8 Class Descriptions

This section provides brief description of the classes, and important functions. Please read the Doxygen documentation provided for further information.
1.8.1 PhysicsWorld

1.8.1.1 BulletWorld

This is the main physics simulation thread. It runs the physics simulation, and calls callback function on all registered PhysicsWorldListeners. The class uses the Bullet Physics Engine for the physics simulation. Objects can be added to the BulletWorld using the `addVehicle()`, and `addMovableRigidBody()` functions. Static Worlds, can be loaded into the world using the `loadFile()` function. PhysicsWorldListeners are registered to the BulletWorld using the `registerWorldListeners()` function.

1.8.1.2 PhysicsWorldListener

The `PhysicsWorldListener` is used for callbacks from the `BulletWorld`. When an object is added or moved in the simulator, it calls the appropriate callback on the `PhysicsWorldListener`. The `PhysicsWorldListener` are registered to the `BulletWorld` using the `registerWorldListener()` function. The `PhysicsWorldListener` functions are called from the `BulletWorld` thread, so these functions must exit quickly to ensure that the physics simulation does not slow down.

1.8.1.3 BufferedPhysicsWorldListener

The `BufferedPhysicsWorldListener` is an example of the `PhysicsWorldListener` which implements a buffered queue. All changes are stored in a buffer and can be gotten using the `getChanges()` function. The `BufferedPhysicsWorldListener` functions properly only if there is one receiver. Otherwise, some of the changes may be lost. The `BufferedPhysicsWorldListener` is thread-safe.

1.8.1.4 RobotManager

This class can be used to add vehicles to the physics simulation. The `RobotManager` is instantiated with the configuration of the robots. The settings should look like this

```java
OpenFrame: {
    collisionName="OpenFrame";
    collisionFile="OpenFrame.bullet";
    mass=10.0;
    r_OA_G=[0.0, 0.0, 0.25]; //the center of gravity
    r_G_B=[0.0, 0.0, -0.05]; //the center of buoyancy with respect to center of gravity
};
Crawler: {
    collisionName="OpenFrame";
    collisionFile="OpenFrame.bullet";
    mass=10.0;
    r_OA_G=[0.0, 0.0, 0.25]; //the center of gravity
    r_G_B=[0.0, 0.0, -0.05]; //the center of buoyancy with respect to center of gravity
};
```

Therefore, when `addVehicle()`, is called with a certain robot type, the properties of the robot type(such as mass, collisionName, and collisionFile) are taken from the setting, and the robot is instantiated with the correct properties.

1.8.1.5 Vehicle

This is the basic vehicle class that is available in the simulator. It provides a basic physical object, which can be manipulated by applying forces and torques to the center of mass using the `setLocalForces()` function. The forces are applied continuously until the forces are changed. The forces continue to act in the local frame of the robot. When the robot is added to the `BulletWorld`, `BulletWorld` sets the chassis of the vehicle, and calls `updateVehicleForces()` on all vehicles that have been added to the
**BulletWorld.** The *Vehicle* class is used by the *RobotManager* to add Vehicles to the *BulletWorld*. In addition to setting forces to manipulate the vehicle, the class also allows simple velocity control using the `setLocalVelocities()` function. Also, when using velocity control, it should be noted that the velocity set overrides velocities caused by the physics engine. For example, if a collision occurs causing an object to go in a different direction, the velocity is again overridden in the next time step. Therefore, force control is more realistic than velocity control.

### 1.8.1.6 UnderwaterVehicle

This Vehicle class simulates a neutrally buoyant underwater vehicle. Every time step, the *UnderwaterVehicle* applies a force to cancel out gravity and also applies a drag force against the motion of the vehicle. Currently, all vehicles created by *RobotManager* are of this class.

### 1.8.1.7 BulletObjectsLibrary

The *BulletObjectsLibrary* is present to unify the loading of bullet objects in one place. The *BulletObjectsLibrary* initializes the models used by Bullet. This can be used by all other classes to load the correct object when an object is added to the world. The *BulletObjectsLibrary* is initialized using a setting for all objects that need to be loaded. The configuration file section for the *BulletObjectsLibrary* looks like:

```json
object1:{
    collisionFile="objecta.bullet"
    collisionName="objectx"
};
object2:{
    collisionFile="object2.bullet"
    collisionName="objectz"
};
```

After initialization, the correct collision model can be loaded using the `getCollisionShape()` function.

### 1.8.2 SensorWorld

#### 1.8.2.1 SensorWorld

The sensors in the simulator are implemented as *PhysicsWorldListeners*. When the world is updated, the world of the sensor(*SensorWorld*) is updated. Then the sensors can use this data to perform scans to get the data they need. The *SensorWorld* class provides the basic functionality for the sensors. When vehicles are added to the world, it calls the `instantiateSensor()` function to create a sensor of the correct type. When vehicles move, it updates the position of the sensors. It also takes calls the `scanWorld()` function when the sensor needs to perform scans.

#### 1.8.2.2 SensorWorldListener

The *SensorWorldListener* allows *SensorWorlds* to dispatch the data collected. *SensorWorldListeners* can be registered to the *SensorWorld* by calling the `registerSensorWorldListener()` function.

#### 1.8.2.3 OdometrySensorWorld

This *SensorWorld* implements the Odometry Sensor. Sensors are added to the robot type with the `addSensorConfig()` function. The sensor setting should have the following parameters
1.8.2.4 RangeSensorWorld

This SensorWorld implements the range sensors, both 2D and 3D. On receiving updates from the PhysicsWorldListeners, it adds them to its collision data structure, which it uses to perform ray traces. Sensors are added to the robot type using addSensorConfig() function.

1.8.2.5 SensorWorldFactory

The SensorWorldFactory is a helper class which constructs the SensorWorlds from the configuration file. It also adds the sensors to the SensorWorlds by calling the addSensorConfig() function. This class requires the settings both for SensorWorlds and for RobotConfigurations. The settings for the sensorworlds must contain the properties required for the sensor world, most important being the type for the sensor world, e.g.

```
SensorWorlds:
    OdometrySensorWorldA: {
        sensorWorldType = "OdometrySensorWorld";
        uSleepTime = 100000;
    };
    RangeSensorWorldA: {
        sensorWorldType = "RangeSensorWorld";
        uSleepTime = 100000;
    };
```

The configuration for the robots must contain the sensors that are present. Each setting for the sensor must have the name of the SensorWorld it is part of (which is then looked up in the SensorWorlds settings described above, as well, as settings for the specific sensor. An example for the robot configuration required is given below.
Due to the fact that the SensorWorldFactory consists of all the sensors and SensorWorlds, it acts as an interface to all the SensorWorlds. It register all SensorWorldListeners to each of the SensorWorlds it creates, mitigating the need to register each sensor world separately.

### 1.8.3 Networking

The networking classes allow remote interaction with the simulator over the network.

#### 1.8.3.1 SerialiserFactory

The SerialiserFactory is implemented as a SensorWorldListener. Therefore, it can be registered for callbacks when the sensorworlds get new data. It dispatches the sensor data to the correct serialiser based on the robot name. Currently, all the data is serialized to USARSim type message format.

#### 1.8.3.2 USARSimSerialiser

The USARSimSerialiser converts data into the format recognized by USARSim. The Message is of the type

```
SEN {Name SensorName} {Data specific to each message}
```

This allows a seamless transition between USARSim and the JR Simulator. Currently, this is the only serialiser that is implemented.

#### 1.8.3.3 USARSimSimulatorServer

This class acts as a TCP server for USARSim type messages. When this class is started, it is possible to spawn and drive robot over the network using standard USARSim messages. Robots can be spawned using the following command

```
INIT {Name robotName} {ClassName robotConfigurationName} {Location x,y,z} {Rotation rx,ry,rz}
```

**Name** the name of the robot. Must be unique, otherwise the robot is not spawned
**ClassName**  the configuration name as defined in the config file. If the configuration name is not known, the robot is not spawned.

**Location**  The position to spawn the robot, in meters

**Rotation**  The rotation of the robot at spawning time. The units are in radians.

The robots can be driven can be driven either with velocity control or force control using the DRIVE command. The format for the drive command is the following:

```
DRIVE {LocalVelocity x,y,z} {LocalAngularVelocity rx,ry,rz}\n```

**LocalVelocity**  The velocity component is in the local frame of the robot.

**LocalAngularVelocity**  The angular velocity in the local frame of the robot.

```
DRIVE {LocalForce x,y,z} {LocalTorque rx,ry,rz}\n```

**LocalForce**  The force in the local frame of the robot, acting on the center of mass

**LocalTorque**  The torque in the local frame of the robot, acting on the center of mass

### 1.8.4 QtOgreGUI

This files in this directory are related to the Graphical User Interface. This includes visualisation of the world as well as visualisation of sensor data. Currently, the only config file setting required for the GUI is the location of the icons to be used for navigation.

```json
GUISettings: {
  iconsDirectory="../resources/QtResources";
};
```

#### 1.8.4.1 MainInterface

This is the central GUI class. It initializes all the other classes required for viewing sensor data as well as the world. To this purpose it is a `SensorWorldListener`. When new sensor data arrives for a sensor, it redirects the data to the appropriate widget. If a widget does not exist for the sensor data, it creates one. When you add new sensors to the simulator, you would need to modify this class to ensure proper operation. Also, when a robot is removed, you need to modify the `removeRobotSignalReceived()` function.

#### 1.8.4.2 OgreSceneWidget

This is the main OGRE widget responsible for viewing the world. To this purpose it is a `PhysicsWorldListener`. When an object is added to the world, it uses the `OgreObjectsLibrary` to load the correct ogre mesh and to move the robot. Also, it emits an `addedRobotSignal` and `removedRobotSignal` to inform the `MainInterface` when a robot is added or removed. This allows the `MainInterface` to add this information to the list of robots in the simulator.

#### 1.8.4.3 OgreFullWidget

This widget consists of the `OgreSceneWidget`, and buttons to allow for navigation inside the widget. The icons at the bottom of the widget all need to be present in the directory defined in the config file.

#### 1.8.4.4 PointCloudViewer

The PointCloudViewer is used to view the point cloud data from the range sensors.
1.8.4.5 RangeSensor2DViewer

The RangeSensor2DViewer is used to view sensor data from the 2d range sensor.

1.8.4.6 RobotTreeWidget

This widget shows the list of robots connected to the simulator. In addition to this, it shows the list of widgets connected to each robot. When an item connected to the sensor is used, it opens the widget connected to the sensor. If the robot name is clicked, the camera is positioned to follow the robot.

1.9 Proposed Development

1. The camera sensor and sending of camera data over the network is currently under development. Currently, only the USARSim serialiser is present which cannot send binary data. Hence, a new serialiser will be implemented for this purpose.

2. Being able to load .Scene Ogre files that correspond to .bullet world file to load the initial World

3. Currently, the number of sensorworlds and sensor loads on the sensor worlds is defined in the config file. Future releases should automatically allocate sensor worlds depending on the sensor loads, to ensure real time sensor data.

4. We are also looking into making a plugin system for listeners which would allow developers to create listeners and linking without the need to compile the simulator again.

5. Since it is known that many groups have their dynamic model implemented in MATLAB as Simulink blocks, we are looking into an interface to Matlab/Simulink.

6. Add noise to the sensor readings.

7. Simulate a sonar, with multiple readings per beam.
Appendix A

Using Co3AUVSimulator on Virtual Machines / Non-Linux-OS

A.1 Virtual Machines on Windows

A.1.1 Overview

It can be of interest to test or run the simulator on a Windows machine. In general, virtualization solutions like VMware, VirtualBox, and Parallels Desktop are good and efficient ways to provide a virtual Linux machine on a Windows OS. The problem is that the Co3AUVSimulator - or more precisely both the OGRE and Bullet libraries require OpenGL support, i.e., 3D hardware acceleration. This causes problems as it is a topic the virtualization companies are still working on. Here, a rough overview of the current state of the situation is given. Please note the dates reported in each section below; this situation may significantly improve over the next months.

Roughly speaking, this is the situation for the simulator and virtualization on Windows

- VMware works but slower than a dual boot installation
- VirtualBox seems not to work
- Parallels Desktop is not tested yet

Details are given below. Note that things may depend on the exact guest, i.e., Linux version, respectively on the graphics card of the computer used for the installation. Any feedback on additional experiences is very welcome in this respect. Please send in this case an email to eecs-robotics@jacobs-university.de.

A.1.2 VMware

This describes the situation in October 2010.

The set-up was tried with VMware 3.1.0 on a Windows XP host. The guest was Ubuntu 10.04. The hardware was a Laptop with a Nvidia graphics card.

Both installation and running the simulator works. The graphics are a bit slow as VMware seems to use a software emulation for 3D graphics. The topic is in general discussed in VMware forums and there seems to be strong interest to get proper 3D hardware acceleration in VMware for Linux guests. If or when exactly this will be provided is unknown.

A.1.3 ORACLE VirtualBox

This describes the situation in October 2010.

The set-up was tried with VirtualBox 3.2.8 on a Windows XP host. The guest was Ubuntu 10.04. The hardware was a Laptop with a Nvidia graphics card.

The 3D support in VirtualBox seems to be still quite experimental. But it has to be enabled in the virtual machine configuration to be able to install Bullet and OGRE. Installation works but running the simulator fails with an error claiming that there is “No GLX FBCConfig support”. This seems to be a known issue for OGRE and other 3D graphics software like some games on VirtualBox.
Appendix B

Tools

B.1 Blender

Blender is a 3D graphics application that can be used for modelling, and texturing. It can be used to create individual objects, as well as scenes. It can import as well as export standard formats, such as 3ds, autodesk, collada format. Blender also uses the Bullet physics engine internally, for its game engine, which can be used for prototyping robot components, such as flippers. In addition to these standard exporters, exporters to OGRE mesh and scenes are also available.

It should be noted that in Blender, the meshes used in the scene are not reused, but rather duplicated. The significance of this is explained in section B.1.2.

B.1.1 Blender to OGRE Mesh Exporter

The exporter to OGRE meshes is available and seems to be stable. It is also part of the package manager of standard linux distributions. It can export to the .mesh and .material format used by ogre. However, Blender does not seem to export the materials properly. The ambient colour is always set to grey, so if the OGRE scene has ambient light, the objects appear to be grey.

B.1.2 Blender to OGRE Scene Exporter

Blender to Ogre Scene exporter can be found in the ogre wiki. The Dot Scene Format used for ogre scenes is an XML format, which shows the position of nodes, and the objects connected to each node. When exporting from Blender, first the scene is exported. Then one must use the OGRE Mesh Exporter to export the meshes individually. However, as explained earlier, blender does not seem to have the concept of object types. Just object names. Thus each object in the scene needs to be exported as a separate mesh. So, if there are 20 instances of the same object in the same, there are 20 meshes exported, instead of one.

B.2 OGITOR

OGITOR is an OGRE scene editor. OGITOR allows us to create scenes with meshes and objects. It also has plugins for loading water meshes(hydrax) as well as Sky boxes.

B.3 OGRE DotScene Framework(DSI)

The DotScene Framework(DSI) is used a software package that is used to import the .dotscene file exported by blender. The code was written in 2007 and no longer compiles, so changes had to be made to a few functions and includes to compile with the latest version of OGRE and Blender.
B.3.1 TinyXML

This software package is a requirement for the Ogre DotScene Framework, and is available from sourceforge. It is a simple xml parsing library which is used by the DSI. It needs to be compiled with STL support in order to be used by DSI. After compiling the binaries, it is convenient to merge the binaries into one library using the following command:

```
gcc -shared -Wl,-soname,libtinyxml.so -o libtinyxml.so tiny*.o -lc
``` 

B.4 GDAL 1.7

GDAL allows reading and writing Bathymetry Attributed Grid (BAG) files. This is non-proprietary file format for storing and exchanging bathymetric data developed by the Open Navigation Surface Working Group. It is used by the National Oceanic and Atmospheric Administration (NOAA) to represent underwater heightmaps. BAG files have two or three image bands representing Elevation (band 1), Uncertainty (band 2) and Nominal Elevation (band 3) values for each cell in a raster grid area.
How to connect to the Co3-AUVSimulator using the Jacobs ROS Package

October 6, 2010

1 Introduction
This document describes the co3auv_sim_robot package in the co3auv_sim stack. This package spawns a robot and publishes the sensor data from the robot. This document assumes basic knowledge of ROS. Some quick start info on ROS including installation guidelines are provided in the appendix of this document.

2 Published Topics
The sensor data is published in the topic
- /co3auv_sim_robot/ROBOT_NAME/SENSOR_TOPIC for sensor data
- /tf for sensor frames.
The specific topics published are
1. /co3auv_sim_robot/ROBOT_NAME/odo_pose for odometry data
2. /co3auv_sim_robot/ROBOT_NAME/range2d_scan for the 2d range scanner data
3. /co3auv_sim_robot/ROBOT_NAME/range3d_scan for the 3d range scanner data

3 Subscribed Topics
The robot can be controlled by publishing to the following topics
1. /co3auv_sim_robot/ROBOT_NAME/local_angular_velocity for controlling the angular velocity of the robot in the local frame
2. /co3auv_sim_robot/ROBOT_NAME/local_velocity for controlling the translational velocity of the robot in the local frame
3. /co3auv_sim_robot/ROBOT_NAME/local_force for controlling the translational force in the local frame
4. /co3auv_sim_robot/ROBOT_NAME/local_torque for controlling the rotational torque in the local frame
5. /co3auv_sim_robot/ROBOT_NAME/voltages for directly controlling the voltages applied to the thrusters.

4 How to use
The package provides a launch file, co3auv_sim_robot.launch, which spawns the robot at a default location provided in the smallauv.yaml config file. The launch file can directly be included in another launch file by setting the AUV_NAME argument. An example of such a launch file is provided below.

```xml
<launch>
  <include file="$(find co3auv_sim_robot)/co3auv_sim_robot.launch">
    <arg name="AUV_NAME" value="robot1"/>
  </include>
  <param name="/co3auv_sim_robot/robot1/robot1_node/initial_location/x" value="55.4"/>
  <param name="/co3auv_sim_robot/robot1/robot1_node/initial_location/y" value="84.4"/>
  <param name="/co3auv_sim_robot/robot1/robot1_node/initial_location/z" value="-20.5"/>
</launch>
```
The steps required to connect to the simulator are the following:
1. Start the simulator. (Instructions provided in the manual of the simulator)

2. Create your own launch file including the launch file `co3auv_sim_robot.launch` and setting the `AUV_NAME` argument.

3. Change any parameters if required.

4. roslaunch your launch file. Look at `co3auv_robot_joy.launch`.

A Installing ROS

This is very easy. Instructions can be found at [http://www.ros.org/wiki/cturtle/Installation/Ubuntu](http://www.ros.org/wiki/cturtle/Installation/Ubuntu).

It is recommended that you get the "Base", i.e., ROS plus robot-generic stacks like navigation and GUI components.
PID Controller Assignment

October 14, 2010

1. Download Co\textsuperscript{3}-AUV simulator repository by running the following commands in a terminal
   
   \texttt{git clone http://10.70.15.2/jrsimulator.git}
   \texttt{cd jrsimulator}
   \texttt{git checkout v0.3}

   Compile the simulator.

2. Update your Jacobs ROS repository by running \texttt{git pull} in any sub-directory of \texttt{jacobs-ros-pkg}.
   Build \texttt{co3auv_sim_robot} package using \texttt{rosmake}

3. Create a new ROS package with the following dependencies \texttt{co3auv_sim_robot} (for communicating with the robot in the simulator), \texttt{small_auv_control} (for Volts message), \texttt{geometry_msgs} (for PoseStamped message), \texttt{roscpp} (for ROS C++ API), \texttt{std_msgs} (for standard messages e.g. \texttt{std_msgs/Float64} can be used to publish a single 64-bit floating-point number)

4. Create a launch file which spawns a robot in the simulator (for details see JacobsROS.pdf provided in one of the previous lab sessions)

5. Move the robot forward with the following command in a terminal.
   \texttt{rostopic pub /co3auv_sim_robot/robot1/voltages small_auv_control/Voltages \{-10.0, -10.0, 10.0, 10.0\} \["front_left", "front_right", "back_left", "back_right"]}

6. Write a node in your package which publishes voltage messages to make a robot move forward

7. Add your node to the launch file such that it publishes voltages to the correct topic using the \texttt{remap} tag. Launch the file.

8. Add subscriber to your node for the PoseStamped message type and modify launch file to receive the position from the robot. Use \texttt{rostopic list} to get the correct name.

9. Extend pose callback to calculate speed and add a publisher for it. The callback should be called with 1 Hz frequency. For more details read about callbacks and spinning in ROS wiki. Publish the speed within the callback.

10. Write a PID controller class which tries to keep a constant forward velocity by setting the voltages.

11. Use the controller in the pose callback and publish calculated voltages. The target speed and P, I, D should be parameters of your node and should be set in the launch file.

12. Use \texttt{rxplot} to plot voltages in one subplot, and target and current speed in the other. Tune the parameters of the controller. You can use \texttt{--period} option to set period displayed in \texttt{rxplot} window. To plot the values from the beginning you should start \texttt{rxplot} before launching your PID controller node.
1. Update your Jacobs ROS repository by running `git pull` in any sub-directory of `jacobs-ros-pkg`.

2. Go to `jir_toro` package and checkout SVN repository of Toro optimizer:
   ```
   svn co -r 28 https://openslam.informatik.uni-freiburg.de/data/svn/toro/trunk toro_svn.
   ```
   Then build the package by running `rosmake`. You can build the documentation for the package using `doxygen toro.dox` in the `jir_toro/toro_svn/doc` directory.

3. Install the `graph_mapping` stack (`sudo apt-get install ros-cturtle-graph-mapping`) and read the documentation `http://www.ros.org/wiki/graph_mapping`.

4. Use your code from the PDL homework to make the robot explore the environment in the simulator. Alternatively you can use a joystick or a keyboard to drive the robot. For that, see `co3auv_robot_joy.launch` and `co3auv_robot_key.launch` in `jacobs-ros-pkg/apps/co3auv_sim/`.

5. In the laser scan callback, use the Karto scan matcher `http://www.ros.org/wiki/karto_scan_matcher` to register the laser scans. You can find the details about correlative scan-matching in [1], [2].

6. Store the results from the scan-matcher in the pose-graph data structure from `jir_toro` package. In parallel, store the laser scans for each node. Look into `toro.cpp` and the `TreePoseGraph2::load` method (in `posegraph2.cpp`) for an example of creating and filling the Toro pose-graph.

7. Visualize the pose-graph after each scan-matching operation. For that represent the pose-graph as an occupancy grid using `nav_msgs::OccupancyGrid overlayClouds(...)` declared in `occupancy_grid_utils/ray_tracer.h`. Look into the `laser_geometry` package (`http://www.ros.org/wiki/laser_geometry`) for the conversion between the `LaserScan` and `PointCloud` types.

8. Stop the robot after some time. Detect and add loop-closing constraints to the pose-graph.

9. Use the `jir_toro` package, which contains the Toro optimizer (`http://www.openslam.org/`), to minimize the error introduced by the constraints.

10. Visualize the pose-graph after the optimization.

11. If you have chosen to drive the robot with keyboard or joystick try running it with the code from the PDL homework.

References
