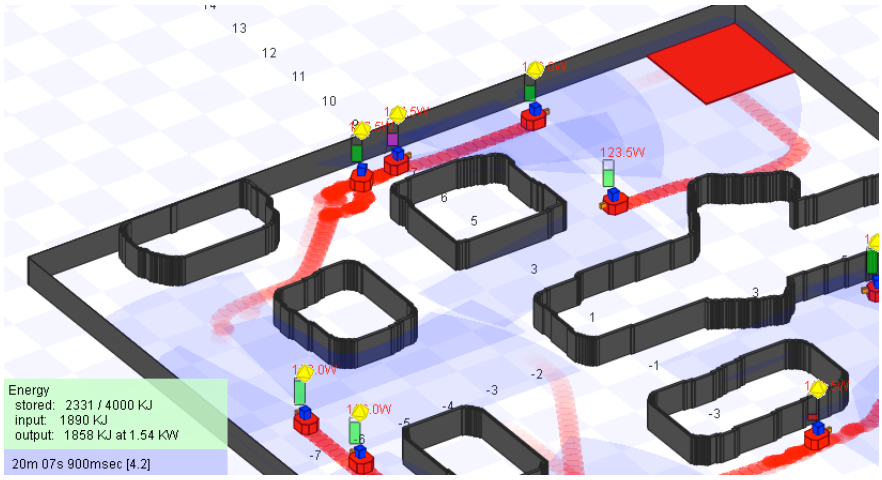
Robotics

Exercise 9.1 Your first robotic explorer

This exercise is a bit different from the previous ones, in the sense that we are going to develop an algorithm that has not been explicitly covered in a lecture. Concretely, it requests you to implement the behavior of a robot that has to explore (visit) an area as large as possible within an unknown space and in a given time. An algorithm like this has many applications: map building, rescue tasks, security works, etc.

The developed algorithm shall be part of a **ROS C++/python node.** It will be tested employing the **Stage simulator**[1]. *Stage is a robot simulator, it provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate. ROS provides a node, called* `stage_ros`[2]*, which wraps the core functionality of Stage, like the utilization of 2D laser scanners or robotic bases yielding odometry information.*



Example of a simulation with Stage showing several robots moving around an environment.

Follow the next steps to end up with a nice robot able to explore places in a *too fast too furious* fashion employing ROS capabilities.

1 http://playerstage.sourceforge.net/doc/Stage-3.2.1/index.html
2 http://wiki.ros.org/stage_ros

1. Getting ready: the provided Virtual Machine and the pre-installed ros-pkgs

For those not willing to install and configure ROS, a virtual machine (Virtual Box) is provided together with this exercise holding a Xubuntu 18.04 OS, which is based on Ubuntu and employs a Xfce interface. It comes with ROS Melodic installed, and a catkin workspace ready to use.

Some indications:

- User of the OS: robotics, password: robotics.
- ROS installation path: /opt/ros/melodic
- Catkin workspace path: /home/robotics/catkin_ws

Note-> If you want to install ROS in your computer, you can follow the detailed guide in: http://wiki.ros.org/melodic/Installation, and install the Desktop-Full version of ROS. If you choose this option, you have to know that the **open_gmapping** package is not available for ROS Melodic in the Ubuntu package repository. That package is a wrapper for GMapping, a SLAM method based on particle filters that we are going to use. However, it's easy to compile it from source. For doing that, execute the following commands:

```
cd ~/catkin_ws/src
git clone https://github.com/ros-perception/openslam_gmapping src/openslam_gmapping
git clone https://github.com/ros-perception/slam_gmapping src/slam_gmapping
cd ..
Catkin_make
```

2. Insight into Stage

Stage comes also pre-installed into the virtual machine. Stage simulates a world as defined in a **.world** file. This file tells stage everything about the world, from obstacles (usually represented via a bitmap to be used as a kind of background), to robots and other objects in the virtual environment. The node stage_ros only exposes a subset of Stage's functionality via ROS. Specifically, it finds the Stage models of type **laser, camera (rgbd) and position**, and maps these models to the ROS topics:

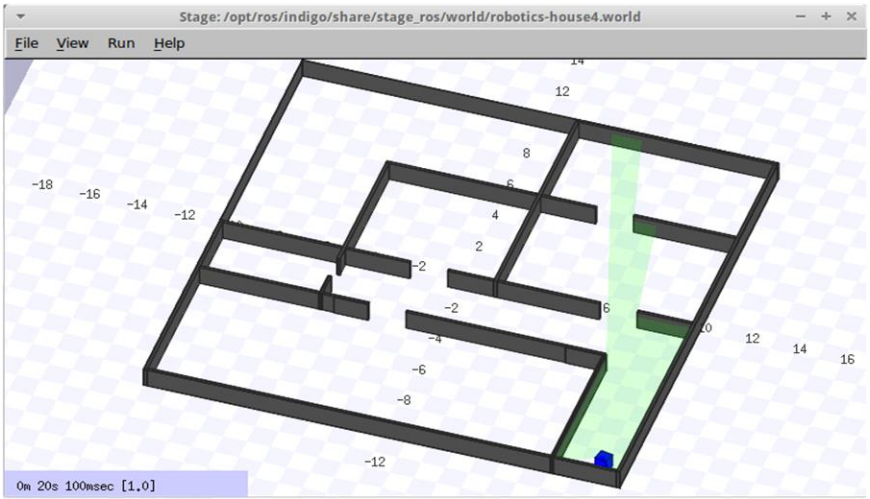
- **laser_scan** (sensor_msgs/LaserScan)
- **image** (sensor_msgs/Image)
- **depth** (sensor_msgs/Image)
- **odom** (nav_msgs/Odometry)
- **base_pose_ground_truth** (nav_msgs/Odometry) **NOT USE!!!**

You can run this node by executing the following command:

```
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-house4.world
```

Note→ Remember that the ROS-MASTER must always be launched before any other node (executing `roscore`):

You will see an interface like the one shown below, where the robot is represented as a blue box, and it is equipped with a 2D laser scanner. Walls are represented as gray, tall rectangles, which define the rooms within the environment to be explored. Spend some minutes taking a look at the *View* options within the Stage interface, since they can be useful at some points of the exercise.



Example of a robot (blue square) and the readings from the laser scanner (green).

3. Some basic movement commands

You can open a new terminal and command the robot directly by publishing in the "**cmd_vel**" topic:

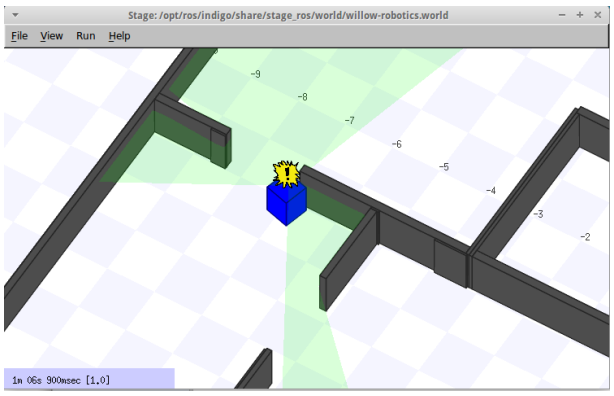
```
rostopic pub /cmd_vel geometry_msgs/Twist '[-0.5, 0, 0]' '[0, 0, 0]' -r 100
```

Since this is a very tedious form of controlling a robot, you can make use of the community and employ different packages that control a robot through the keyboard. For example, you can use the "**keyboard_control**" pkg that is also provided with this document.

You can launch this simple example with the command:

```
roslaunch missions_pkg demo_1_stage_keyboard.launch
```

Take care commanding the robot, it can crash otherwise!



Example of a sad robot.

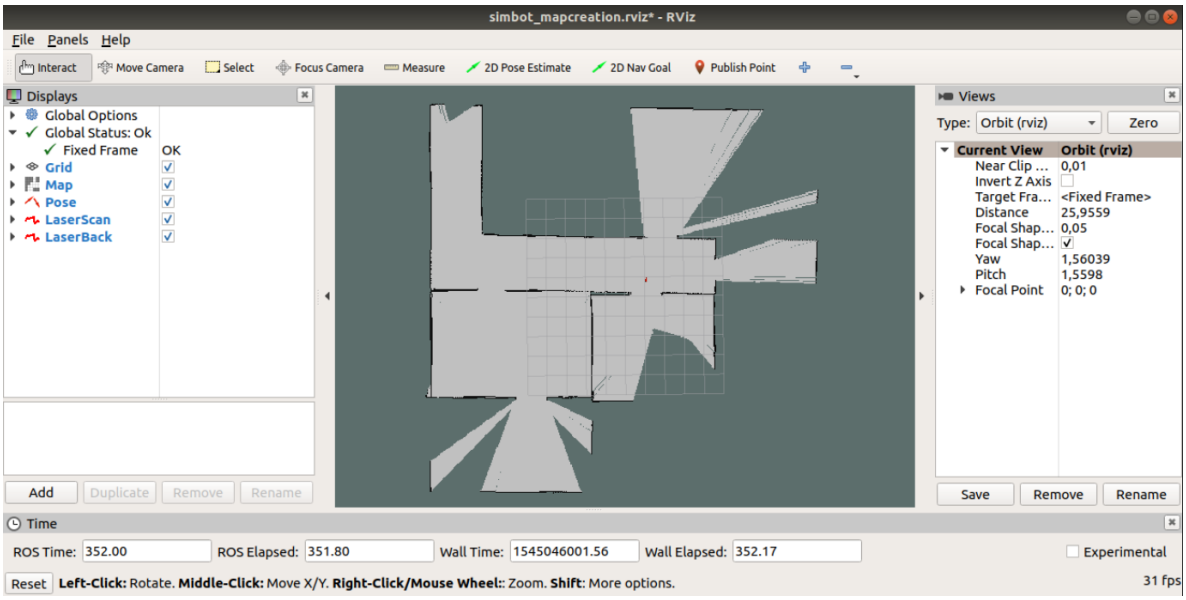
4. SLAM demo

Although we will use Stage as a robotic simulator, it is a good practice to visualize the data coming from the robot outside the simulator. That way, when we use a real robot, the interface will be the same. RVIZ (http://wiki.ros.org/rviz) is a 3D visualization tool for ROS which a lot of pre-configured ROS message types.

As a demo for introducing ROS and RVIZ, let's take a look at a very employed package for SLAM: **gmapping**. This package contains a ROS wrapper for OpenSlam's Gmapping. The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

As a simple exercise, you can load the Stage simulator, then launch the gmapping node, and finally move manually the robot to see how the map builds up while localizing the robot (SLAM). For ease of use, you can use **roslaunch** (http://wiki.ros.org/roslaunch) a tool for easily launching multiple ROS nodes, as well as setting parameters on the **Parameter Server**. For example, you can launch this demo with the command:

```
roslaunch missions_pkg demo_2_slam.launch
```



Example of a map under construction.

5. The robotic explorer

6-1 – Analyzing the *starting* code. A number of code snippets (in C++) are shown below, which belong to the *starting* code that you have to extend. Explain their functionality one by one. Recall that you can also develop **Python** code.

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/LaserScan.h>
#include <stdlib.h>
```

Importing the classes ROS, Twist for moving the robot, the laser in LaserScan and the libraries for std, essential in cpp.

```
int main(int argc, char **argv)
{
  // Initialize the ROS system and become a node.
  ros::init( argc , argv , "exploring") ;

  ROS_INFO_STREAM("Robotic explorer node running and initialized! Let's have some
fun!");

  ros::NodeHandle nh;

  // Create a subscriber object
  ros::Subscriber sub = nh.subscribe("/base_scan", 1000, processScanCallback);

  // Create a publisher object
  ros::Publisher pub = nh.advertise <geometry_msgs::Twist>("/cmd_vel" , 1000);
```

Inicialization of the main, where it inicialize the ROS system and become a node the program, then the NodeHandler, and for last it susbscribe to the laser and it will publish the Twist we will do after.

```
  // Seed the random number generator.
  srand ( time (0) ) ;

  // Loop at 2Hz until the node is shut down or during 5 minutes
  ros::Rate rate (2) ;
  ros::Time begin = ros::Time::now();

  while ( begin.toSec() == 0 )
    begin = ros::Time::now();

  double ellapsed_time = 0;
```

Inicialization of the random number generalization and the timer.

```
  while ( ( ros::ok () ) && ( ellapsed_time < 60*5 ) )
  {
    // Create and fill in the message. The other four
    // fields , which are ignored by stage, default to 0.

    geometry_msgs::Twist msg;
```

```
    msg.linear.x = 2*double( rand() ) / double(RAND_MAX);
    msg.angular.z = ( 4*double(rand()) / double(RAND_MAX) ) - 2;

    // Publish the message.

    pub.publish(msg) ;

    // Send a message to rosout with the details .
    ROS_INFO_STREAM("Sending random velocity command: "
         << " linear=" << msg.linear.x
         << " angular=" << msg.angular.z ) ;

     // Wait until it 's time for another iteration .
    rate.sleep ();
    ros::spinOnce(); // give time to receive /base_scan messages

    ros::Time current = ros::Time::now();
    ellapsed_time = (current - begin).toSec();
    ROS_INFO_STREAM("Ellpased time: " << ellapsed_time );
  }
}
```

Until ros have a error or the time exceeds the 5 minuts limit, it will loop. In the loop is going to change the velocity and the rotation of the robot randomly, then will publish this change to ROS and will inform the changes in the values. Then is going to stop to give time to receive the message. Lastly, we will update the time.

```
void processScanCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  int n_ranges = msg->ranges.size();
  double nearest = *(std::min_element(msg->ranges.begin(),msg->ranges.end()));
  ROS_INFO_STREAM("I've a total of " << n_ranges << " measurements to process! Are you
ready? Nearest=" << nearest );
}
```

This function correspond to the subscribed laser where will see what is capturing the laser, how many samples and which is the nearest of all of them.

6-2 – Now it's time to develop the algorithm for the robot to explore unknown environments! **Describe the solution adopted/implemented**. You can extend the code above, located at: `/home/robotics/catkin_ws/src/robotic_explorer/src/robotic_explorer_node.cpp`, following the next indications:

- You are allowed to modify the code, not only extend it. The only thing that you have to keep is the execution time limit of 5 minutes, i.e., the robot has 5 minutes to explore/visit as many rooms as possible.
- You can get inspiration from the internet. The efficient exploration of an unknown environment by a mobile robot has been widely studied in the literature, so there are many works addressing it. If you do it, include the inspirational works into the bibliography of the exercise's deliverable.
- Hint: you can subscribe to any topic published by the **stageros** node, with the exception of **base_pose_ground_truth.**

The code can be edited with your favorite text editor or IDE. There is information on the internet about how to work with different IDEs to develop a ROS package/node.

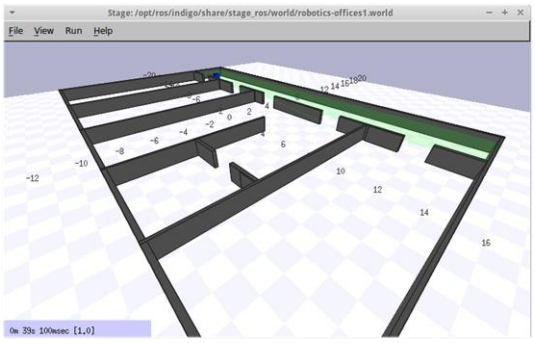
Example of commonly used IDES are QtCreator or KDevelop for C++, or PyCharm or Spyder for Python.

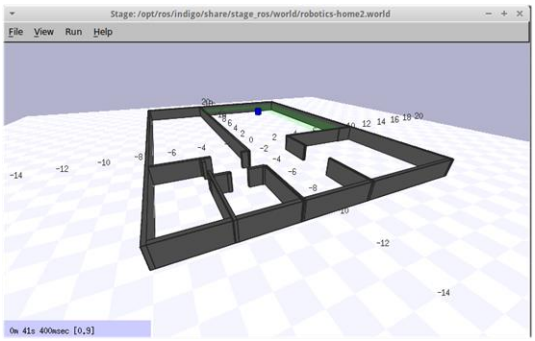
The first code (robotic_node_explorer_1) utilizes a heuristic algorithm which it will go to the farthest site. Also tries to avoid the walls and turns backs if it is in a room, which is interpreted as the values haves the same value as the mean. However, this is a bad algorithm because it is easily to be stuck in a minimum local.
The second code changes and try to organize the samples of the laser, group the same ones and do a mean of then and follow with one have the shortest distance. It is a little better than the last one, but it is still easy to be stuck in a minimum local.
A major improvement would be to receive the map of the SLAM and try to do an A* to the points where we do not have any information.

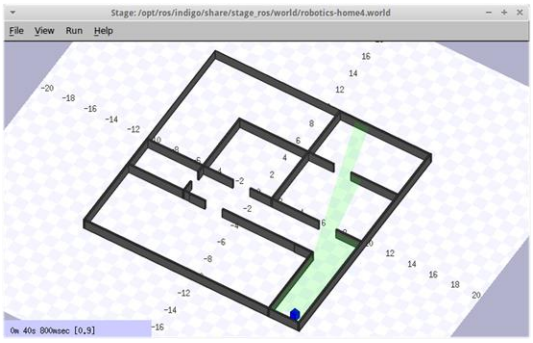
6-3: There are 5 different environments to test your robotic explorer. You can launch them with the following commands (see bellow), or by reusing the “roslaunch” files provided:

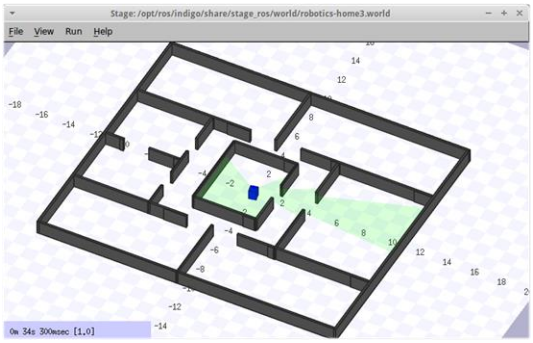
```
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-house1.world
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-house2.world
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-house3.world
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-house4.world
rosrun stage_ros stageros $(rospack find missions_pkg)/world/robotics-offices1.world
```

These images illustrates some of these environments:









Execute your **robotic_explorer_node** for each of the 5 provided environments and count the number of rooms visited by the robot in each of them. Include the initial room as a visited one, and if a room is visited twice or more, then count it only once. Include this information in the deliverable.

Both explorer:
Room1 => 2 rooms
Room2 => 1 room
Room3 => 1 room
Room4 => 1 room
Officine1 => 1 room