Programming Alien-Hexa robot with ros

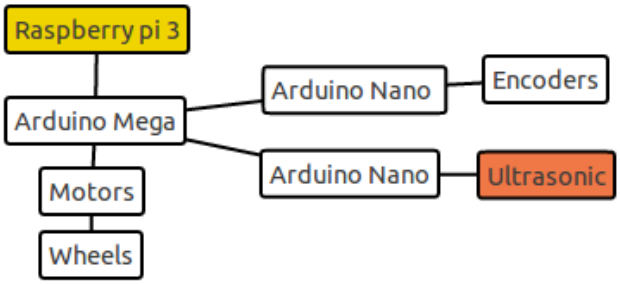
Ahmed Nashaat – Ahmed Radwan – AbdelRahman Mahmoud- Gehad Essam
Aerospace engineering department
University of science and technology Zewailcity
Giza- Egypt

INTRODUCTION

A mobile robot is an automatic machine that is capable of locomotion. Mobile robots have the capability to move around in their environment based on their mission and are not fixed to one physical location. Most of them are used for research education. They also have become more commonplace in commercial and industrial settings. Hospitals and warehouses have been using autonomous mobile robots to move materials for many years. Design, simulation and manufacturing of a mobile robot for mapping, navigation and motion planning will be discussed in the paper with details.

LAY OUT

To integrate and apply SLAM including mapping and motion planning. A computer is needed with Ubunto OS and ROS distribution installed. Raspberry pi 3 which is a card-sized computer is used with Ubunto mate installed and ROS kinetic installed on Hexabot. Microcontrollers used with Raspberry are a couple of ArduinoNano. First ArduinoNano controls the motors and gets encoder data. Second ArduinoNano is used to run six ultrasonic sensors used for mapping the environment. An additional Arduino mega used for integrating the two Nanos connected to them through serial communication. Connecting ArduinoNano to ArduinoMega has advantages of higher memory and better baud and uploading rates. Also, using. Two separate ArduinoNano has an advantage of higher safety in case of failure of one of the microcontrollers, The whole system doesn't fail.



Rosserial Library is used to publish and subscribe to topics through serial communication over a serial port.

Rosserial have multiple packages: Arduino, windows, mbed and tivac. In Hexabot, Rosserial Arduino is used. Customized ROS-Arduino code is uploaded to board and a node is being run to subscribe to topics published from Arduino serial or even subscribing topics to Arduino in form of commands like velocity command.

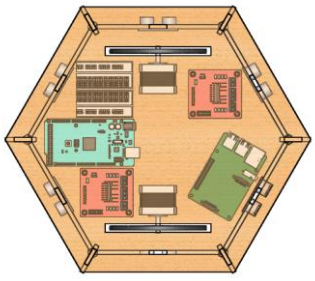
I. Hardware design and manufacturing

Design requirements

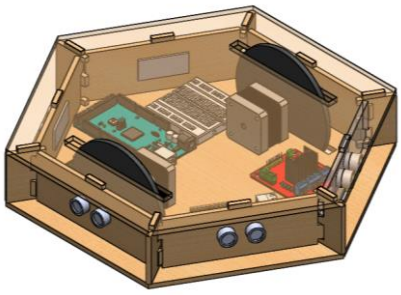
In order to move from simulation to testing phase, we have to have a hardware platform that able to perform the task. A mobile robot was design with mobility and sensing features with onboard CPU to perform localization, mapping and motion planning. Solidworks was used to create the CAD model and during the manufacturing process. A Laser cutter machine was used to create the structure parts. The parts then assembled together using glue, epoxy and screws.

First CAD model

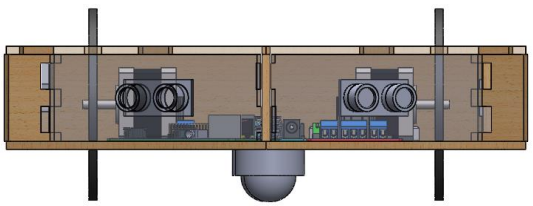
The first model was a hexagonal shape that has an ultrasonic sensor on each side (6 ultrasonic sensors). Two stepper motors were used, connected with a custom made wheels to provide the robot with the required motion. Two caster wheels were used to keep the robot balanced.



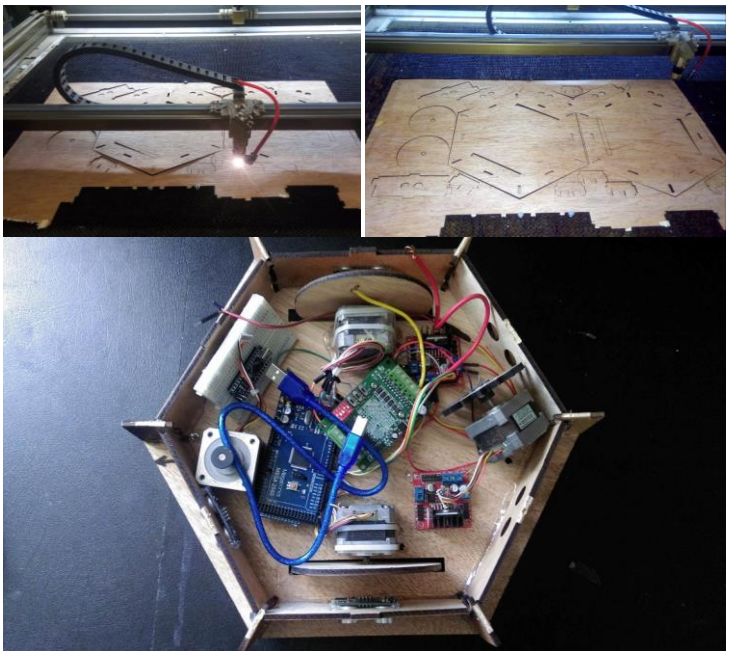
Top view of the first CAD model



Isometric view of the first CAD model



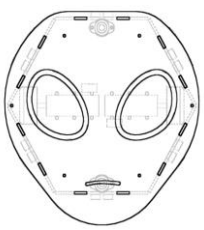
Front view of the first CAD model



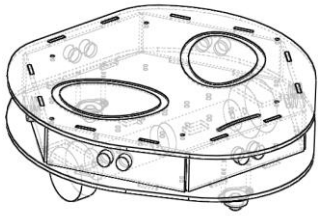
Fabrication Steps: Laser cutter machine then assembly

Second CAD model

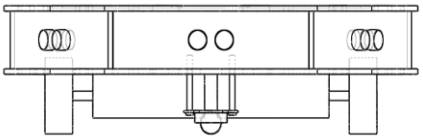
The second Design was bigger and stiffer than the first one. The following tow figure shows the new CAD design and its final after fabrication.



Top view of the second



Isometric view of the second CAD model



Front view of the second CAD model

Problems and troubleshooting

We missed the motor sizing step that costed us a second iteration of design and fabrication process. The stepper motors were perfectly suitable for the task as it has a full observability of the position, but the torque produced was not enough to move the robot. Those steppers were the strongest in the market and yet less than the mission needs, the market limits enforced us to use DC motors with encoders. This new system with DC motors and encoders required a major changes in the structure. Wood and laser cutter operation cost were relatively cheap compared to the cost of ordering strong stepper motors from abroad. So a choice of manufacturing another robot with DC motors is cheaper and will take less time.

The new design solved many problems reported by team members. The following table summarize each problem and its solution in the new design

problems	solution
Component accessibility is hard	More space was dedicated for each component
Motors fixation is not firm	Holes were made around the motor to insert zip ties that keep the motors in place
The structure is not perfectly rigid (some parts can move relative to each other)	Screws were used with glue
The front and back are identical which confuses team members during testing	The top cover is asymmetric shape which enable us to eyeball the orientation rapidly
Caster wheels were not in the same plane of the main wheels	Longer screws were used
Weak stepper motors	High torque DC motors with encoders

Final model

II. Simulation and programming using ROS:

A. Motion planning libraries

1. (SBPL) - Search based planning. “not studied”.
2. (CHOMP) – Trajectory optimization based. “not studied”
3. Moveit! - RRT algorithm, but only implemented for manipulators not mobile robots.
4. Descartes - The APIs are completely unstable.
5. Navigation stack – works for mobile robots & studied planning algorithms like A* & Dijkstra

Navigation stack was chosen as we studied the planning algorithms available and there is a good documentation for it to refer to if we faced any problem during implementation.

B. Nav-stack requirements

Navigation stack has both hardware & software requirements that must be met to work properly.

Hardware requirements

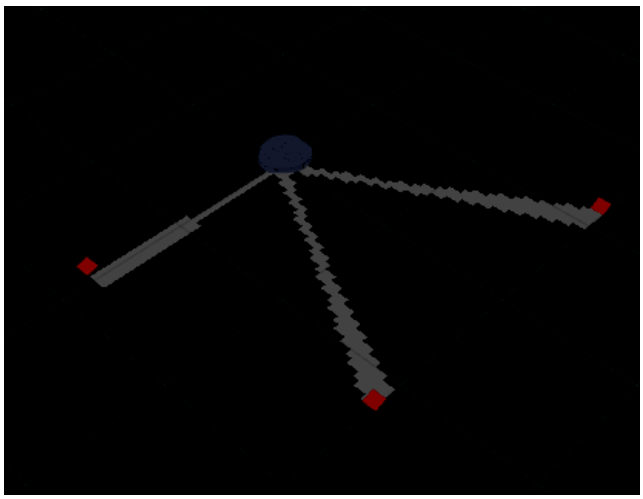
1. Just differential drive or holonomic robots.
2. Laser scanner mounted on the robot for mapping of the environment.
3. Nearly square or circular robots. Theoretically it works with robots with any shape, but it was tested only on square and circular robots, this is the reason of this requirement.

Software requirements:

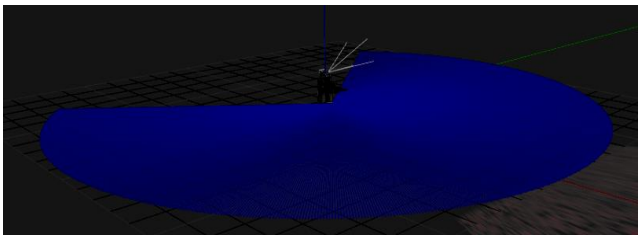
1. Odometry data.
2. Laser scanner data or point cloud data for mapping.
3. The transformation between the frames.
4. A controller node to transform the output velocities of the navigation stack (geometry_msgs/Twist type) into motor velocity commands.

Resolving the requirements:

- Laser scanner is not available, it is substituted with ultrasonic sensors. However, this is not that efficient as laser scanners gives a cone of measurements, which makes it useful in mapping, but ultrasonic gives a beam of measurements, which makes it very inefficient for mapping. We solved this by placing 6 ultrasonic sensors, one on each face of our robot, and performing rotation and translation during mapping.



Ultrasonics measurements



Laser scanner mounted on PR2 measurments

- Unfortunately, Gazebo has no plugins for ultrasonic sensors. The ultrasonics were modeled in the .urdf file and presented in Gazebo as laser scanners with very low field of view (6 degrees).
- Odometry data: for simulation purpose, the odometry source was chosen to be the world itself to provide the real odometry. Data fusion of the imu & encoders would be done to get odometry data of the real robot.
- Transformation is done using tf library.

C. Using Nav-stack:

We are using two nodes from Nav-stack, which do all the required job by linking to other nodes and packages to complete the required tasks:

1. slam_gmapping:

The first node is “slam_gmapping” which is used for mapping. The node takes laser scan data (ultrasonic in our robot) and the transforms as inputs, and gives a 2D map of the environment as output. The subscribed topics in our case are: “/scan” to get sensor data & “/tf_static” to get transforms.

2. move_base:

This node moves the robot to the desired goal by liking to planner & cost-map packages. It takes a desired goal as an input and outputs velocity commands.
“/move_base_simple/goal” topic to get the desired goal. This topic is published on by Rviz. Following are the different packages linked by this node and taks performed by each one of them:

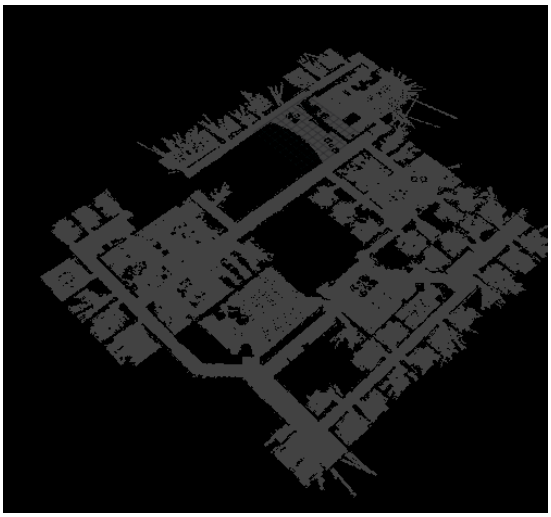
Global_planner package: The main task of this package is to generate a global path from
the robots position to the goal.
Local_planner package: The main task is to make the robot move on small portions of the
global plan.
cost_map2D package: This package outputs two occupancy-grid maps, a global one and a
local one with the specified dimensions and characteristics passed as arguments for “move_base” node.

First procedure:

“Mapping the environment “willow garage for simulation purposes” → saving the map → performing localization on the saved map using “amcl” node → perform motion planning.”

The map is saved using “map_server” node in two files: a .png file and a .yaml file that contains configuration of the map (resolution, origin, occupied threshold, ..etc). The .yaml file is loaded on the map_server which would open the map on Rviz. In this procedure, “slam_gmapping” is used for mapping, “map_server” for saving the map, “amcl” for localization & “move_base” for planning.

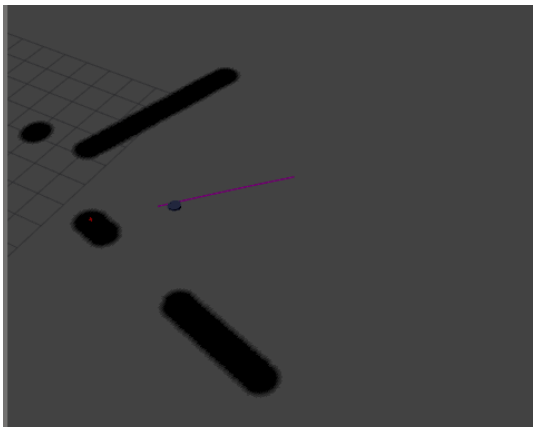
Problems encountered in this procedure:
1. Bad map was generated by the ultrasonics, this caused a problem to AMCL as it needs a well-defined map generated by laser scanners. This was solved by getting an already mapped willow garage environment and loading it to the map_server.



Map of the willow garage

2. The AMCL could not localize the robot in the map, as it works by using laser scanner scans and matches them to the map built with the laser readings. Here we do not provide laser scans, instead ultrasonic scan, which cannot match the map.

Second procedure:
Performing mapping alongside with motion planning, meaning that a goal is set to the robot in unmapped environment and, the robot generates a path “straight line” and follows it. When it detects an obstacle it generates another path that avoids the obstacle. This way mapping is done while approaching the goal. This procedure was implemented when the problem of AMCl emerged and could not find a solution for it. In this procedure we just use “move_base” & “slam_gmapping” nodes.



Procedure2: the robot is mapping the environment while following the generated path

III. Implementation on the real robot

Code and logic

To get a map for the surrounding environment. Six ultrasonic sensors are used launching sequentially to avoid crosstalk. The code uploaded to ArduinoNano

First, we have to include the header file for ultrasonic ping library in order to be able to use the functions built in it,

```
#include <ros.h>
```

Then define trigger and echo pins for each ultrasonic.

```
#define trigL1 9
#define echoL1 8
```

Specify No. of ultrasonic sensors (six) and max range. Ping interval is the interval between triggering ultrasonic sensors as they should be launched sequentially to avoid any interference between rays. The min interval determined by experiment is about 29 milliseconds so 50 milliseconds is chosen to guarantee avoiding the crosstalk.

```
#define SONAR_NUM     6
#define MAX_DISTANCE 300
#define PING_INTERVAL 50
```

Then in the void setup function (which runs only once). Baud rate is defined. High baud rate is safer for avoiding delay in sensor readings especially because they are six. Then for loop is made to set the starting time for each sensor (sequentially launched).

```
void setup() {
  Serial.begin(115200);
  pingTimer[0] = millis() + 75;           // First pi
  for (uint8_t i = 1; i < SONAR_NUM; i++) // Set the
    pingTimer[i] = pingTimer[i - 1] + PING_INTERVAL;
}
```

Then. Ultrasonic are launched. Each ultrasonic has two pins. One for “Trigger” which is used to trigger the ultrasonic to start by setting the pin low then high then low. Another is for “Echo” which is used to send the duration data to the microcontroller. By implementing calculations on duration data, distance in cm can be extracted.

```
void loop() {
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    if (millis() >= pingTimer[i]) {
      pingTimer[i] += PING_INTERVAL * SONAR_NUM;
      if (i == 0 && currentSensor == SONAR_NUM - 1) oneSensorCycle();
      sonar[currentSensor].timer_stop();
      currentSensor = i;
      cm[currentSensor] = MAX_DISTANCE;
      sonar[currentSensor].ping_timer(echoCheck);
    }
```

Printing value on serial which in this case is connected to ArduinoMega

```
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    Serial.print(cm[i]);
    if (i != 6){
      Serial.print("\t");
    }
```

Now, ArduinoMega code is ready to be uploaded

First, include the header files for the ROSserial library to be able to use the functions and messages type included. In this case, although ultrasonic sensors are used, sensor messages of type LaserScan is used to be able to send not just one reading but an array of ultrasonic readings.

```
#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/LaserScan.h>
```

Creating the node handle allows publishing and subscribing to topics and also handles the serial communication stuff. Creating message of type LaserScan named range_msg and publishing this message to a topic named /ultrasound.

```
ros::NodeHandle  nh;
sensor_msgs::LaserScan range_msg;
ros::Publisher pub_range( "/ultrasound", &range_msg);
```

After variables and functions definitions and Setting baud rates for ArduinoMega and the two serial ports connected to Arduino Nano, Node must be initiated and using either advertise to print the published topics or subscribe to send commands.

```
void setup() {
  Serial3.begin(115200);
  Serial2.begin(115200);
  Serial1.begin(115200);
  nh.initNode();
  nh.advertise(pub_range);
```

Then LaserScan parameters are defined. Angle_min and angle_max specify the start angle of the scan and the end angle of the scan. angle _increment is the regular angle between measurements. Header is the acquisition time of the first ray in ultrasonic sensors.range_min and range_max are the minimum range value and the maximum range value.

```
  range_msg.angle_min = -3.009;
  range_msg.angle_max = 3.18;
  range_msg.angle_increment = 1.04;
  range_msg.header.frame_id =  frameid;
  range_msg.range_min = 0.0;
  range_msg.range_max = 2.5;
```

According to ArduinoNano code. Ultrasonic readings are printed on serial3 to which ArduinoNano is connected, Serial3 is read to get these values. After parsing the data array, it can be published to the node running ROS.

```
//receiving from ultrasonics
void r_ultra(){
  if (Serial3.available()) {
    c_ultra = Serial3.read();

    if (c_ultra == '\n') {
      parse_ultra(command_ultra);
      command_ultra = "";
    }
    else {
      command_ultra += c_ultra;
    }
  }
}
```

Where ultra parse function implements the parsing is as following.

```
void parse_ultra(String command_ultra1){
  part1_ultra = command_ultra1.substring(0, command_ultra1.indexOf("\t"));
  command_ultra1.remove(0, command_ultra1.indexOf("\t")+1);
  part2_ultra = command_ultra1.substring(0,command_ultra1.indexOf("\t"));
  command_ultra1.remove(0, command_ultra1.indexOf("\t")+1);
  part3_ultra = command_ultra1.substring(0,command_ultra1.indexOf("\t"));
  command_ultra1.remove(0, command_ultra1.indexOf("\t")+1);
  part4_ultra = command_ultra1.substring(0,command_ultra1.indexOf("\t"));
  command_ultra1.remove(0, command_ultra1.indexOf("\t")+1);
  part5_ultra = command_ultra1.substring(0, command_ultra1.indexOf("\t"));
  command_ultra1.remove(0, command_ultra1.indexOf("\t")+1);
  part6_ultra = command_ultra1.substring(0,command_ultra1.indexOf("\n"));

  ultraL1 = part1_ultra.toFloat();
  ultraL2 = part2_ultra.toFloat();
  ultraR1 = part3_ultra.toFloat();
  ultraR2 = part4_ultra.toFloat();
  ultraF = part5_ultra.toFloat();
  ultraB = part6_ultra.toFloat();
  ultra_all[0]=ultraL1;
  ultra_all[1]=ultraL2;
  ultra_all[2]=ultraR1;
  ultra_all[3]=ultraR2;
  ultra_all[4]=ultraF;
  ultra_all[5]=ultraB;

}
```

After uploading the two codes to Arduinos, a terminal can be opened to receive and transmit to Arduino and echo the topic published which is the vector of ranges. Visualization in Rviz was done.

First, run roscore to prepare the master which will arrange the publishing and subscribing to and from the node

```
roscore
```

Then run the rosserial node which is essential to connect between arduino and ROS and transform arduino sensor data to readable ROS messages. Serial Port connected to arduino should be mentioned in the command.

```
rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Now it's time to know what arduino can publish and what it can subscribe to. This can be done by listing all the available topics

```
rostopic list
```

Available topics obviously include the topic needed which is named "/ultrasound"

```
/rosout_agg
/tf
/ultrasound
```

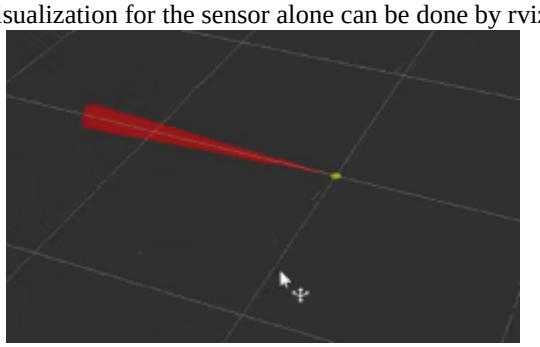
After that, we can echo this topic to see the sensor readings.

```
rostopic echo /ultrasound
```

Values are arranged in a vector of readings

```
ranges: [300.0, 300.0, 35.0, 144.0, 300.0,
```

Visualization for the sensor alone can be done by rviz



```
[ERROR] [WallTime: 1483992890.718902] Error opening serial: could not open port
/dev/ttyACM0: [Errno 2] No such file or directory: '/dev/ttyACM0'
```

Most probably this error occurs when serial port name is wrong. Check serial port name from arduino user interface tools tab.

```
avrdude: stk500v2_ReceiveMessage(): timeout
```

This happens sometimes while uploading arduino codes when arduino serial port is busy this can be for several reasons

- Serial communication device connected to Tx and Rx pins
- ROS running node which uses the por

It can be solved by unplugging the jumpers from Rx and Tx. or terminating the running node of ROS serial by clicking ctrl+c

```
[ERROR] [WallTime: 1483995131.170014] Unable to sync with device; possible link
problem or link software version mismatch such as hydro rosserial_python with gr
oovy Arduino
```

This on the first look seems like there is a kind of mismatch between arduino software and rosru

INDEX A

References

[1] http://answers.ros.org/questions/

[2] http://wiki.ros.org/

[3] Lentin Joseph-Mastering ROS for Robotics Programming-Packt Publishing (2015). Book.