My Lidar

ESE 519 - Fall 2012
Team:
Nirmiti Mantri
Runyu Ma
Xiaochen Pan

Acknowledgement

We would like to express our very great appreciation to Dr Rahul Mangharam for his valuable and constructive suggestion during the planning and development of this project. His willingness to give us time so generally has been very much appreciated.
We would also like to thank the Teaching Staff of ESE 519 for their valuable inputs and constant guidance.
And last but definitely not the least a big thank you to our friends from ESE 519. They were a constant source of support and motivation. They have helped us through every phase of the project.

Contents

Abstract

In this project we present the use of SLAM, simultaneous location and mapping, for generating 2D map of the environment by a robot car in real time. The 2D map stores for each cell of a given grid a posterior about the amount the corresponding cell is covered by an obstacle. Further the robot car can autonomously navigate in this environment using the generated map. The robot can void obstacles in the environment and can navigate successfully from the point of start to the user-specified destination. The entire system can be controlled from a remote laptop. This application can be used to generate maps of unknown environment and then use autonomous robots for rescue or safety operations as per its capacity.

2. Introduction

We have implemented a robot car, which can generate a very precise(thanks to Laser)2D map of the unknown environment. SLAM is a technique of simultaneous mapping and localization. We have incorporated grid based SLAM technique that aids the robot in storing the location of obstacles in the environment. After it generates the map the robot uses this self-generated map of the environment to navigate from the start point to the user-specified end point. At the heart of the system we have Beagleboard, which runs ROS and has the sensors interfaced to it. This data is sent in real time to a remote laptop that generates and displays the map, executes the navigation algorithm and then notifies the Beagleboard about the commands that should be given to the mbed, which controls the dc motors and servo motors of the robot car. The sensors used are a Laser for scanning the environment and a Wii-remote that we hacked into and used it as an Inertial Measurement unit. The software implementation is done using Robot Operating System (ROS), which is used over Ubuntu. The software implementation is based on the obtained data from sensors along with the algorithm for autonomous navigation computes the best path to reach the destination. The detailed description of each of the system module is described ahead.

3. Hardware Interfacing

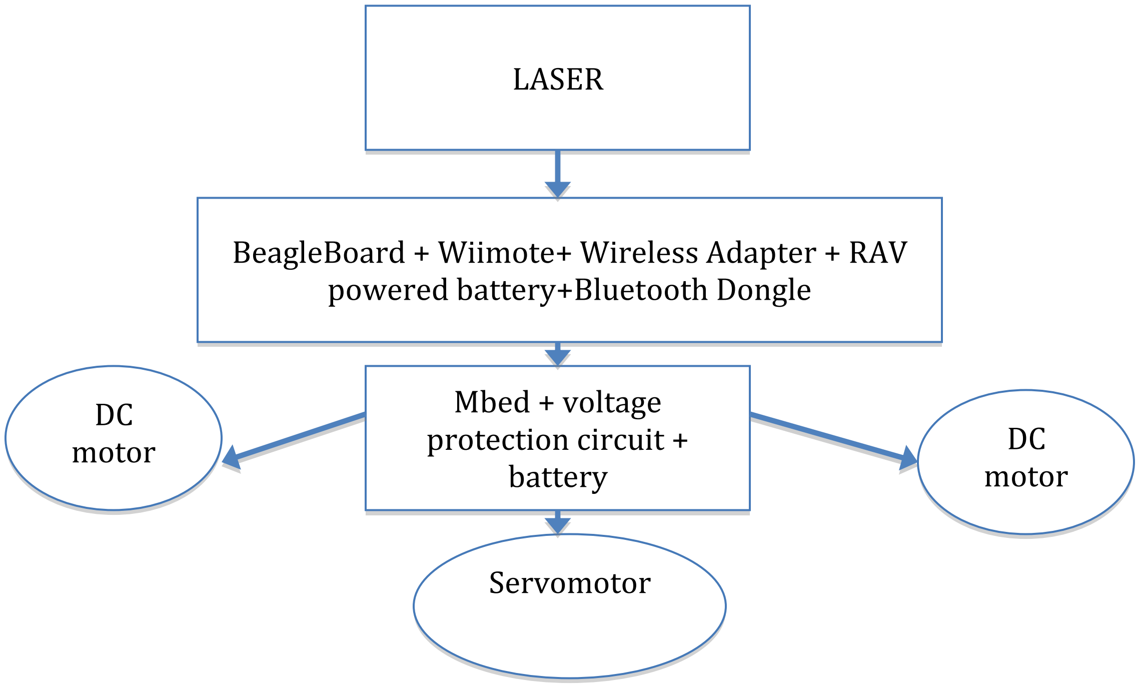
3.1 The Beagleboard along with the Laser, Wiimote and the batteries are mounted on a robot car. The robot car has three tiers, each have been described below:

1) Lowermost Tier:
   Components: DC motor, H-Bridge, Servomotor, Mbed, Flywheel diodes, Voltage Regulators and Batteries.
   Description: MBED microcontroller is programmed to receive commands from beagle board and send PWM wave to H-bridge in order to control the velocity of the DC motors and turning angles of servo motor(HS81). In order to reduce the power drain by mbed from the heavy loaded beagleboard, we have powered the mbed with power from the four AA sized battery by regulating the voltage to 5V with the 7805 transistor. To protect the mbed from blowing off by the back emf or surge in voltage, the Flywheel diodes are connected to the PWM output pins(21 and 26) of mbed.

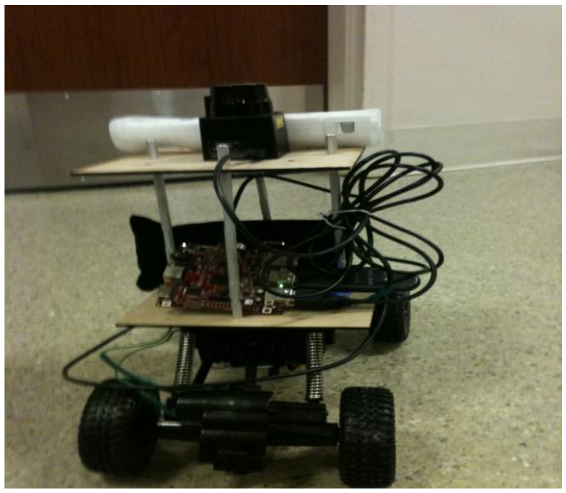
2) Middle Tier:
   Components: Beagleboard, Wiimote, Battery for beagleboard, Wireless Adapter Bluetooth dongle
   Description: Beagleboard runs the ROS drivers for Hokuyo laser node and Wiimote. Wiimote is used as an IMU to get the orientation of the robot car. Wireless adapter is used to connect beagleboad to wifi. Because of the high current requirement of beagleboard connected to laser, mbed, wireless adapter, and Bluetooth dongle, we have finally adopted the RAV powered rechargeable battery(10000mAh) so that it can supply 2A current to the components interfaced to the Beagleboard for 2 hours.

3) Upper Tier:
   Components: Hokuyo URG-O4LX- UG01 Laser
   Description: It is light in weight and has low power consumption (2.5W for longer working hours). The operating voltage is 5V. We have placed the Laser at a height such that it can scan the environment correctly. . The range is 20mm to 5.6m and 240° with an accuracy of 30mm. The rate of scan is 100msec/scan.

3.2 Diagram of the Robot Car:


LASER
BeagleBoard + Wiimote+ Wireless Adapter + RAV powered battery+Bluetooth Dongle
DC motor
Mbed + voltage protection circuit + battery
DC motor
Servomotor

Picture of the Robot Car:



3.3 Interfacing Details:

BeagleBoard interfacing with remote Computer

We installed Oneric(11.10) server version of Ubuntu on beagleboard. Because of its limitation in display and computing power, we have to use a remote computer to do this project in this way: beagle board responsible for collecting the laser scan, yaw data and send motion commands to mbed, while remote computer execute the SLAM algorithm and Navigation path planning. Beagleboard and remote computer communicate via the inherent ROS TCP/IP protocol. We have set static IP address to both of them so that we don't have to set them up each time.

BeagleBoard Interfacing to Wiimote:

Wiimote communicate with beagle board with Bluetooth. With the wiiuse library, we have written a ROS package to obtain the yaw data and integrate over time to get the orientation in real time.

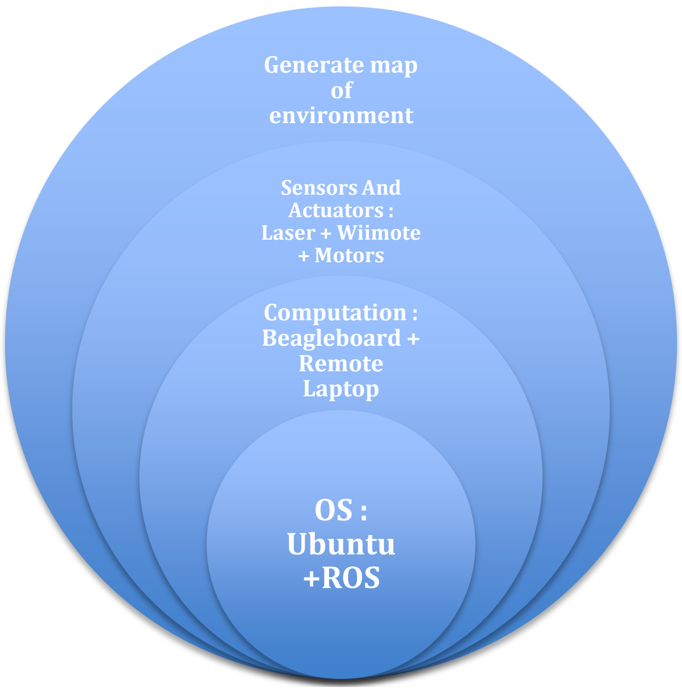
Beagleboard Interfacing to Laser:

We adopted the hokuyo_node package, which uses serial communication to obtain the real time scan data.

4. Software Implementation:

4.1 System Diagram:

Below is the overall system diagram including the software and hardware components.


Generate map of environment
Sensors And Actuators : Laser + Wiimote + Motors
Computation : Beagleboard + Remote Laptop
OS : Ubuntu +ROS

OS: The core of the system is Ubuntu + ROS.

Computation: The extraction of data from the sensors and algorithm computation is balanced between the Beagle board and the Remote Laptop.

Sensors and Actuators: Laser, wiimote, DC motors and servo motors.

Map: 2D map of the environment using SLAM.

4.2 Background Information about ROS:

In case you are interested in ROS and does not have the time to explore it for yourself, here is some brief explanation of ROS so that the discussion of this project makes sense to you. If you are already familiar with ROS, please skip this section.

ROS

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

- Packages: Packages are the lowest level of ROS software organization. They can contain anything: libraries, tools, executables, etc.

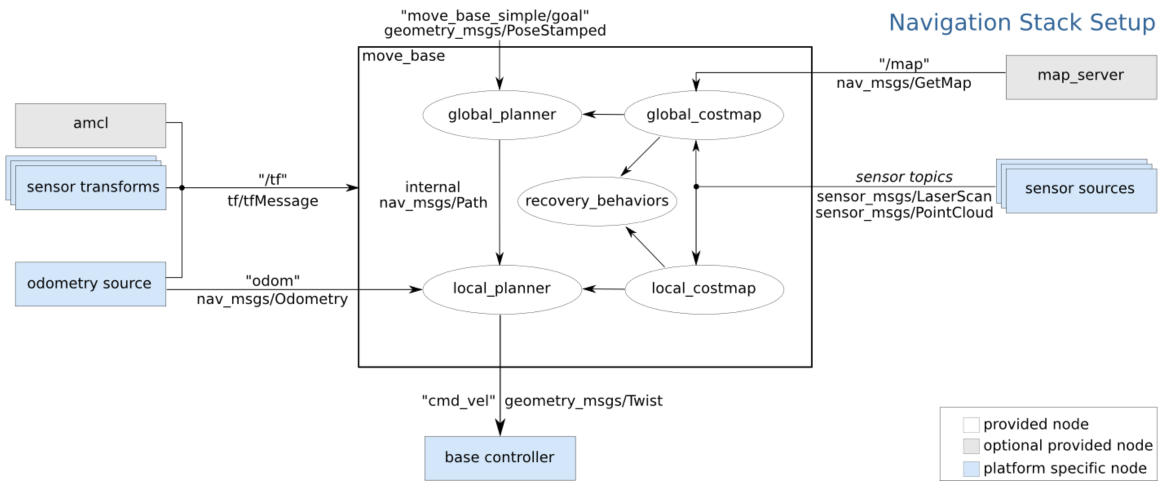
- Manifest: A manifest is a description of a *package*. Its most important role is to define dependencies between *packages*.
- Stacks: Stacks are collections of *package*s that form a higher-level library.

ROS Node:
A *node* is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on. All running nodes have a graph resource name that uniquely identifies them to the rest of the system. For example, /hokuyo_node could be the name of a Hokuyo driver broadcasting laser scans. Nodes also have a *node type*, that simplifies the process of referring to a node executable on the fileystem. These node types are package resource names with the name of the node's package and the name of the node executable file. In order to resolve a node type, ROS searches for all executables in the package with the specified name and chooses the first that it finds.

ROS Topic:
Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data *subscribe* to the relevant topic; nodes that generate data *publish* to the relevant topic. There can be multiple publishers and subscribers to a topic. Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead.

4.3 System Stack:

A 2D navigation stack that takes in information from odometry source, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base.



4.3 Package Summary
amcl :

amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. Amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates. On startup, amcl initializes its particle filter according to the parameters provided. Note that, because of the defaults, if no parameters are set, the initial filter state will be a moderately sized particle cloud centered about (0,0,0).
Subscribed Topics:
Scan, Tf, Initial Pose, map
Published Topics:
amcl_pose , tf.

Base controller:
Package : sub_vel
The navigation stack assumes that it can send velocity commands using a geometry_msgs/Twist message assumed to be in the base coordinate frame of the robot on the "cmd_vel" topic. This means there must be a node subscribing to the "cmd_vel" topic that is capable of taking (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) velocities and converting them into motor commands to send to a mobile base.
Subscribed Topics:
cmd_vel
Published Topics:
cmd_vel

move_base:
given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the nav_core::BaseGlobalPlanner interface specified in the nav_core package and any local planner adhering to the nav_core::BaseLocalPlanner interface specified in the nav_core package. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner (see the costmap_2d package) that are used to accomplish navigation tasks.
Subscribed Topics:
move_base_simple/goal
Published Topics :
cmd_vel

map_server :
map_server provides the map_server ROS Node, which offers map data as a ROS Service. It also provides the map_saver command-line utility, which allows dynamically generated maps to be saved to file. When comparing to the threshold parameters, the occupancy probability of an image pixel is computed as follows: occ = (255 - color_avg) / 255.0, where color_avg is the 8-bit value that results from averaging over all channels, e.g. if the image is 24-bit color, a pixel with the color 0x0a0a0a has a

probability of 0.96, which is very occupied. The color 0xeeeeee yields 0.07, which is very unoccupied.
Published Topics:
map_metadata , map

sensor sources- hokuyo node :
hokuyo_node is a driver for SCIP 2.0 compliant Hokuyo laser range-finders. This driver was designed primarily for the Hokuyo UTM-30LX, also known as the Hokuyo Top-URG. The driver has been extended to support some SCIP1.0 compliant range-finders such as the URG-04LX. Hokuyo scans are taken in a counter-clockwise direction. Angles are measured counter clockwise with 0 pointing directly forward.
Published Topics:
scan

sensor transform – transform:
tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.
Subscribed Topics:
tf_old
Published Topics:
tf

odometry source- wiimote:
The navigation stack uses tf to determine the robot's location in the world and relate sensor data to a static map. However, tf does not provide any information about the velocity of the robot. Because of this, the navigation stack requires that any odometry source publish both a transform and a nav_msgs/Odometry message over ROS that contains velocity information.
Subscribed Topics:
Published Topics:

Global Planner:
In order to score trajectories efficiently, the Map Grid is used. For each control cycle, a grid is created around the robot (the size of the local costmap), and the global path is mapped onto this area. This means certain of the grid cells will be marked with distance 0 to a path point, and distance 0 to the goal. A propagation algorithm then efficiently marks all other cells with their manhattan distance to the closest of the points marked with zero. This map grid is then used in the scoring of trajectories.The goal of the global path may often lie outside the small area covered by the map_grid, when scoring trajectories for proximity to goal, what is considered is the "local goal", meaning the first path point which is inside the area having a consecutive point outside the area. The size of the area is determined by move_base.
Subscribed Topics:
move_base_simple/goal

Published Topics:
nav_msgs/Path

Local Planner
In order to score trajectories efficiently, the Map Grid is used. For each control cycle, a grid is created around the robot (the size of the local costmap), and the global path is mapped onto this area. This means certain of the grid cells will be marked with distance 0 to a path point, and distance 0 to the goal. A propagation algorithm then efficiently marks all other cells with their manhattan distance to the closest of the points marked with zero. This map grid is then used in the scoring of trajectories.The goal of the global path may often lie outside the small area covered by the map_grid, when scoring trajectories for proximity to goal, what is considered is the "local goal", meaning the first path point which is inside the area having a consecutive point outside the area. The size of the area is determined by move_base.
Subscribed Topic:
nav_msgs/path

costmap_2d
The costmap_2d package provides a configurable structure that uses sensor data to store and update information about obstacles in the world through the costmap_2d::Costmap2DROS object. The costmap_2d::Costmap2DROS object provides a purely two dimensional interface to its users, meaning that queries about obstacles can only be made in columns. For example, a table and a shoe in the same position in the XY plane, but with different Z positions would result in the corresponding cell in the costmap_2d::Costmap2DROS object's costmap having an identical cost value. However, the underlying structure that the costmap_2d::Costmap2DROS object uses to store information about the world is, by default, three dimensional(see voxel_grid). This allows the costmap_2d::Costmap2DROS object to be principled in the way that it clears out space in the world because it performs clearing and marking operations in this three dimensional grid.

Subscribed Topics:
scan , map
Published Topics:
Obstacles, inflated obstacles, voxel grid, unknown space.

rotate_recovery
The rotate_recovery::RotateRecovery is a simple recovery behavior that attempts to clear out space in the navigation stack's costmaps by rotating the robot 360 degrees if local obstacles allow. It adheres to the nav_core::RecoveryBehavior interface found in the nav_core package and can be used as a recovery behavior plugin for the move_base node.

Packages Implemented by Ourselves:

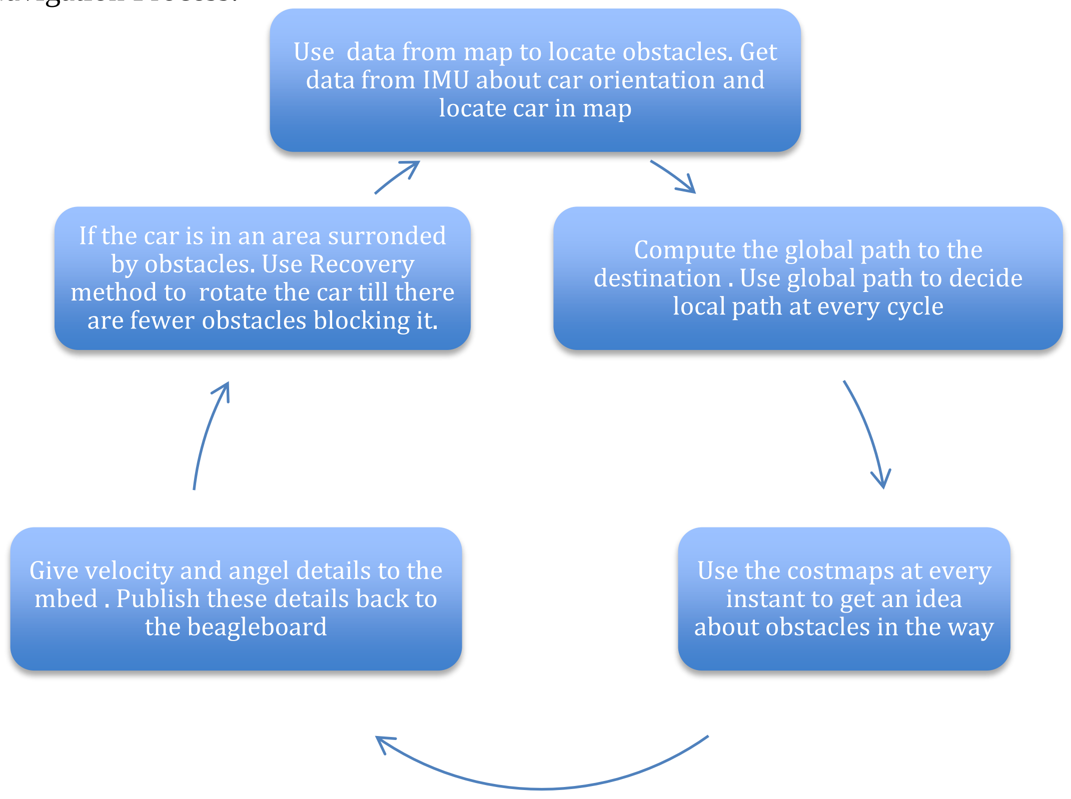
1. Base controller package. We have implemented the base controller package. This package runs on beagle board.The functionality of this package:
   A. Subscribe to the cmd_vel topic and retrieve the velocity commands in this topic.

    B. According to the velocity commands, send corresponding serial commands to mbed.
    C. Publish the current velocity at a rate of 18Hz.

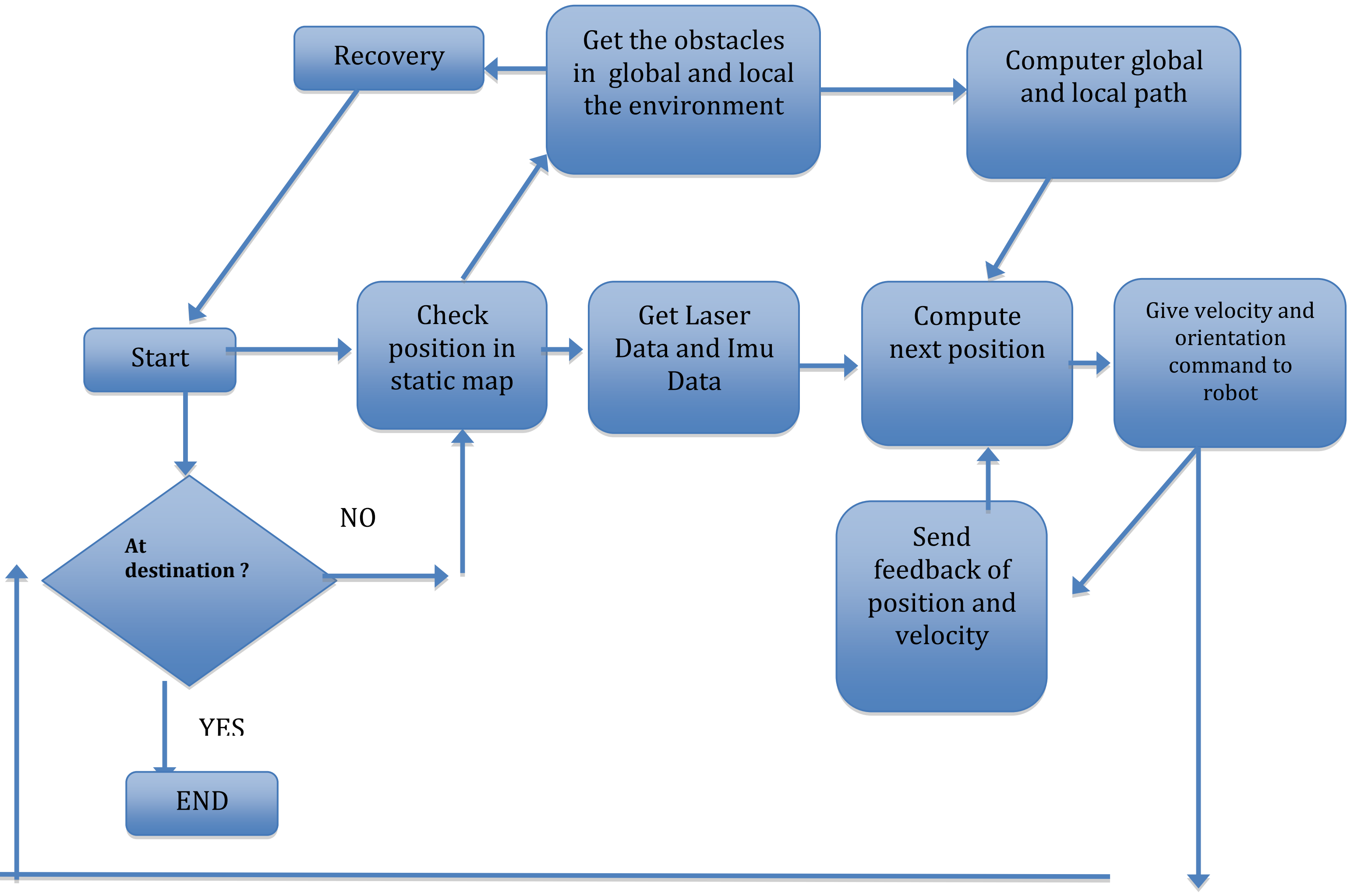
2. Wiimote driver package. We have implemented the wiimote driver package. This package runs on beagle board. This wiimote driver could achieve very high angle accuracy. This package is responsible for:
    A. Invoke the wiiuse[1] library.
    B. Connects with and read data from Wiimote
    C. Compute the real time orientation and angle data and publish them over Master.
3. TF publisher. This publisher is used to publish static transformation over frames (map vs odom) and (base_link vs base_laser).
4. Odom broadcaster. This package is used to broadcast the odometery transformation between base_link and odom frames. Here are the calculation algorithm we have used:
    1) Subscribe to the velocity topic "act_vel" and retrieve the real time velocity of the car.
    2) Subscribe to the orientation topic "quaternion" and "angle" and retrieve the real time angle of the car.
    3) Every time a velocity topic is received, it will compute the real time position of the car by integration discretely of the last position and average velocity of the past period.
5. Simple goal. This package is used to publish the destination of this car.

[1] Wiiuse: https://github.com/rpavlik/wiiuse

Navigation Process:


Use data from map to locate obstacles. Get data from IMU about car orientation and locate car in map
Compute the global path to the destination . Use global path to decide local path at every cycle
Use the costmaps at every instant to get an idea about obstacles in the way
Give velocity and angel details to the mbed . Publish these details back to the beagleboard
If the car is in an area surronded by obstacles. Use Recovery method to rotate the car till there are fewer obstacles blocking it.

The algorithm flowchart:


Recovery
Get the obstacles in global and local the environment
Computer global and local path
Start
Check position in static map
Get Laser Data and Imu Data
Compute next position
Give velocity and orientation command to robot
At destination ?
NO
Send feedback of position and velocity
YES
END

Limitations:

1. As you could tell, we have no IMU to tell the actual position of our car. We intended to use wiimote as IMU in the first place. However, we found it to have very poor accuracy regarding the acceleration data. Then we dig into the reason and find out that because of the principle of accelerometers, they don't have very precise measuring or even close to precise measuring regarding 2D space. And we don't have enough time at the end to buy and use an encoder to do the localization. Therefore, we limit the velocity range and use constant velocity. By this way, the odometer data could be calculated precisely. We have done extensive checks on the odometer output and they are very precise. However, this implementation have give us several drawbacks:
    a. When the car got stuck at obstacles, it would not give the actual velocity of the car. Instead, the robot would think the car to be still moving. Therefore, if this car ever got stuck, the odometer data given is not right any more. This drawback could be prevented if we employ the encoder and play with it.
    b. For the obstacle avoidance functionality provided by the navigation software stack, if the car is limited to constant velocity, it would increase the hardship and the resulting configuring effort would increase tremendously.

2. The mobility of the car. For the navigation stack to work well under a lot of unknown spaces, the car should have the following capabilities are as following:
    1. Rotate itself without moving in order to do its self recovery.
    2. Move in any direction and could change speed.

    Our car does not have the above functionality so that the controlling of the car in the speed and angle does not achieve very good performance.

Future Improvements:

We will build a new robot and add the encoder component to the implementation so that the obstacle avoidance functionality could be achieved.

Instruction for beginners:

You could follow the tutorial of the ROS navigation stack. They have very detailed explanation. Because we do not suggest you to use this car for future project, so that the website tutorial is a good start. This navigation stack is a very complicated software which only works with extensive tuning of the configurations, which could take at least two weeks amount of time, which we did not anticipate in the first place. The car is very critical for this project. Please find yourself a very handy and easily controlled one. The car have to have the fully capability mentioned above, otherwise, the tweaking of the parameters would not work. In addition, before you start on any part of this project, make sure you understand from the start to the final point of the project, otherwise, you may find at the end that what you have chosen for the earlier part does not work well for your final goal.