Grid Based Indoor Localization System (GrBILS)

Michael Sanzari [1], Aneesh Abhyankar [2]

WINLAB

Rutgers University,

New Jersey, USA

[1] michael.sanzari@rutgers.edu, [2] aneesh.abhyankar@rutgers.edu

Abstract – To solve the problem of accurate indoor positioning at low-cost and using light algorithms, a system utilizing fiducial markers in the form of Chilitags, and an upward facing web-camera was developed. Holonomic Drive was used for the robot with omni-wheels. Robot Operating System (ROS) was used as the backbone software and the networking solution. The application considered here is the case of the Mobile-node to facilitate wireless research in the ORBIT-Testbed at Rutgers Wireless Information Network Laboratory (WINLAB). Benefits of the system include low upfront cost with need for minimal and one time infrastructure, and the integration of new platforms at the cost of an inexpensive camera.

Keywords – Indoor Positioning, Fiducial Markers, Robot Operating System, Holonomic Drive

Introduction

In order to complement the capabilities of WINLAB's Orbit testbed, a Mobile-node is needed to allow researchers to carry out experiments with more flexibility. There are two basic underlying problems herein. One is how to localize the robot in a known environment and the other is how to navigate it thereafter. We looked at solutions such as having a LIDAR, or a laser scanners. However, we found that since our room had little features, in terms of walls and corners, the laser scanner data would start drifting as the robot drifted. Point to be noted here is that we do not have encoders on the robot wheels, and thus cannot measure and eliminate drift mathematically. The solution that we came up was to use an up-facing camera and have chilitags (they are similar to QR codes, but limited to only 1024 different combinations) up on the ceiling. The camera detects a chilitag that it can see and returns the relative position of itself with respect to that tag. It gives several such measurements that can be used to triangulate the position of the robot, thus implicitly

eliminating the drift. We found that a differential two-wheel robot caused problems while accurately achieving the given target. If for example, it overshot its path, it had to carry out a string of complicated movements in order to recover. An omni-directional robot eliminated all these problems, as it can recover easily because of its ability to move in any direction.

System Architecture

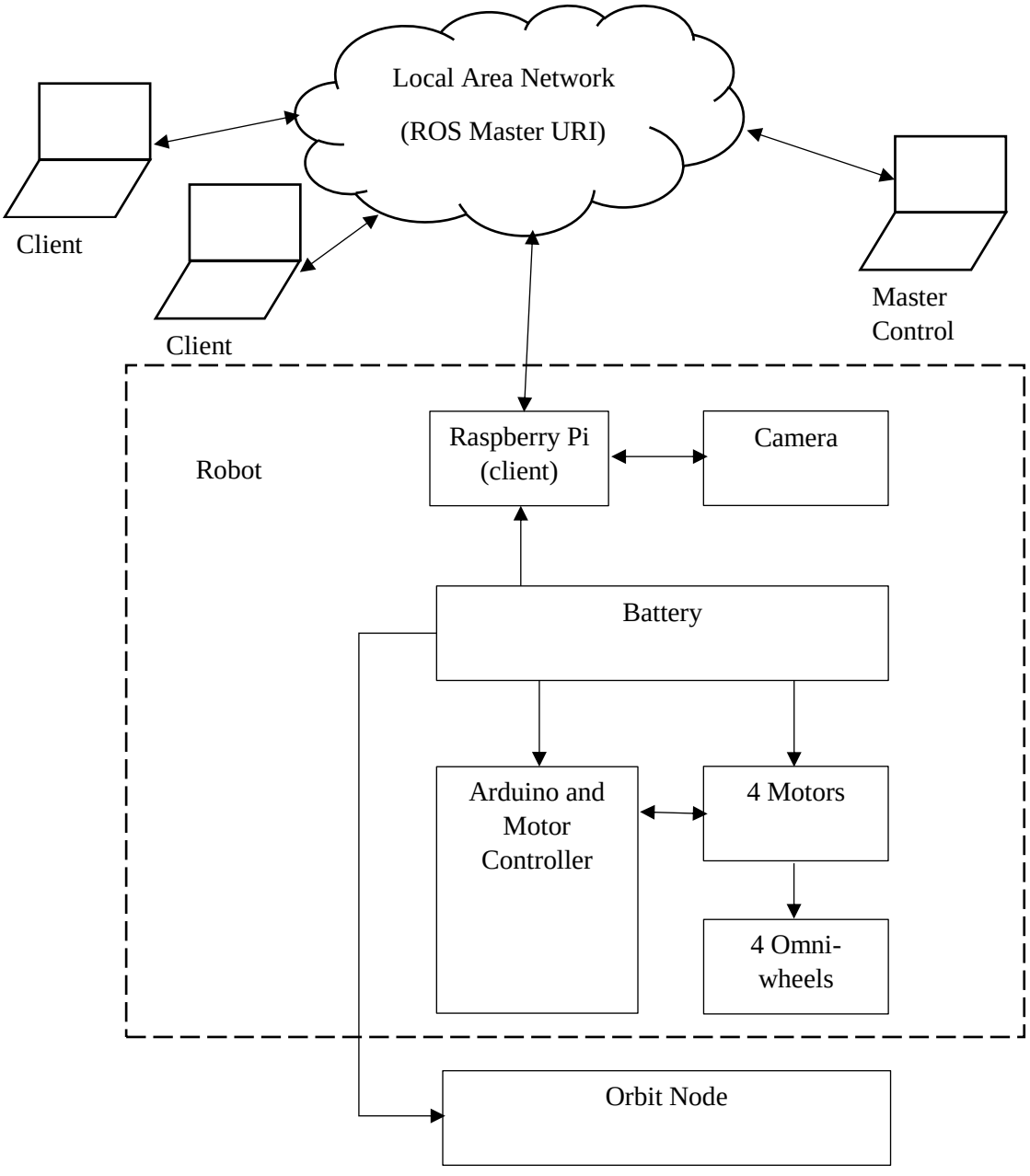

Figure 1 – Architecture diagram of the system

According to the system architecture diagram given in figure 1, the entire control is done by the system called Robot Operating System (ROS) [1]. It is a platform used to develop software in C++ or Python for robotics, machine vision, localization, navigation, etc. It uses several topics as interfaces to exchange information/data between different nodes in the network. Nodes are basically collections of subscribers, that listen to data in the topics and publishers, that write data to the topics. It uses the Local Area Network to create a network of a master and several clients (which may include computers, microcontrollers, IoT devices, etc.) [2]. This network helps seamless interconnection between the master and clients. All the subscribers and publishers contact the master, which forwards the request to clients. The writing of data to topics is broadcast communication, so that any information written to a topic is available to all nodes. Listening to a topic is a unicast communication, so that data is sent only to the node which requests it. One of the clients is the Raspberry Pi that is onboard the mobile node. It figures out robot commands in order to achieve the target based on the relative position of the target with respect to itself. An upward facing USB web cam is plugged into it which looks for chilitags. There is also an Arduino and a motor controller on board in order to translate robot commands to robot motion. The orbit node is to be mounted on the robot so that the users can give commands to move the node under the grid.

Chilitags

Chilitags is a software library designed to detect and identify fiducial markers or tags. It was developed for projects of CHILI lab [3]. They are called tags and look like QR codes, but limited to 1024 different combinations. Following is the image of a chilitag.

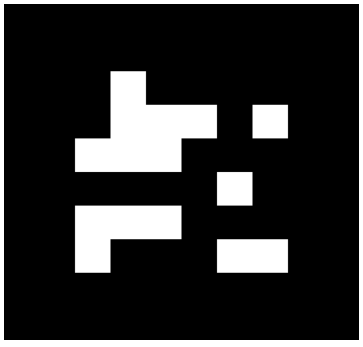

Figure 2 – Chilitag (Fiducial Marker)

We used a ROS wrapper for chilitags [4] that facilitated publishing of the relative position and orientation of the detected tags with respect to the camera in terms of ROS topics. It published the data as TF transform frames, which is basically the position (x, y, z) and orientation (roll, pitch, yaw). At any particular time, the camera sees around 6 to 7 tags. The positional data coming from all of these tags, is used to triangulate the exact position and orientation of the robot.

Map

We used a configuration file to describe the entire map of the grid in a markup language called YAML or Yet Another Markup Language. This configuration file basically contains the known positions of all the tags that are put up on the ceiling in the grid. In our case, there are 462 tags in a 20x20 grid that has 400 nodes. The entire collection of tags arranged in a grid fashion is defined as ‘ceiling_grid’. Thus, whenever the mobile node sees a particular number of tags in the grid, it localizes itself inside the ‘ceiling_grid’. The figure is the map of tag-locations among orbit nodes.

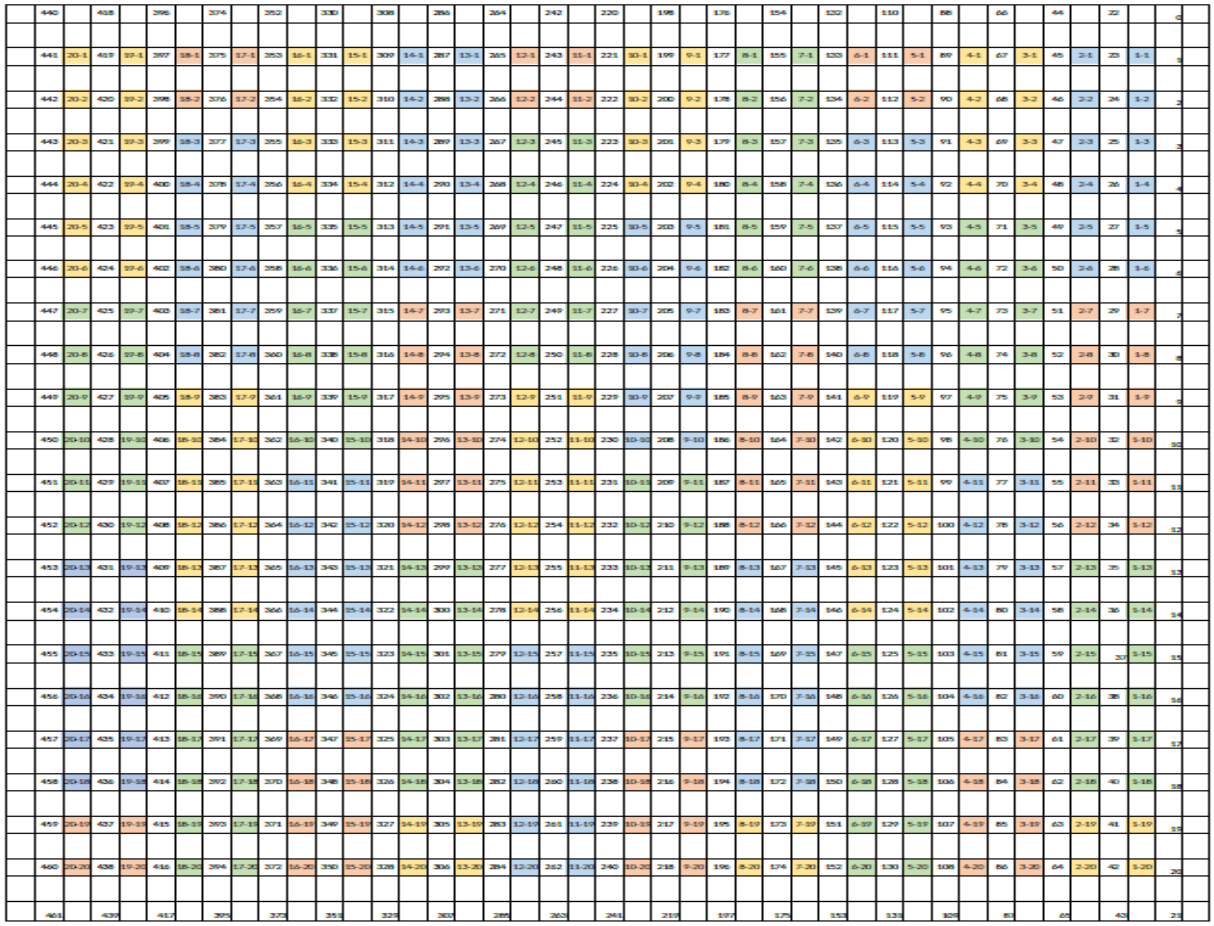

Figure 3 – Map of Chilitags in the Orbit Grid

The photographs below show the chilitag placement inside the orbit grid. The tags are placed at the center of two nodes in rows and we have one row of tags on each of the outer sides of the grid.

Figure 4 – Placement of Chilitags in the Orbit Grid

That is the reason why we have 462 tags in a grid of 400 nodes. The outer layer of tags help the robot to be inside the grid at all times and ensures that the robot never goes into a non-deterministic environment.

The Mobile Node

We started off by working with a more sophisticated differential robot called Pioneer 3DX [5]. We ran into problems right at the beginning where the robot failed to correct itself when it ran over the given target. It had to turn all the way back and move and repeat itself many times before achieving the target and that too with bad accuracy. We changed to a holonomic robot [1] that has wheels that move forwards as well as laterally, making it possible for the robot to move in any direction at any given time. In this way, even if the robot ran over its target by a little bit, it simply moves backwards in order to correct itself. The following figure 5 shows the build of the holonomic robot and the figure 6 shows the omni wheel.

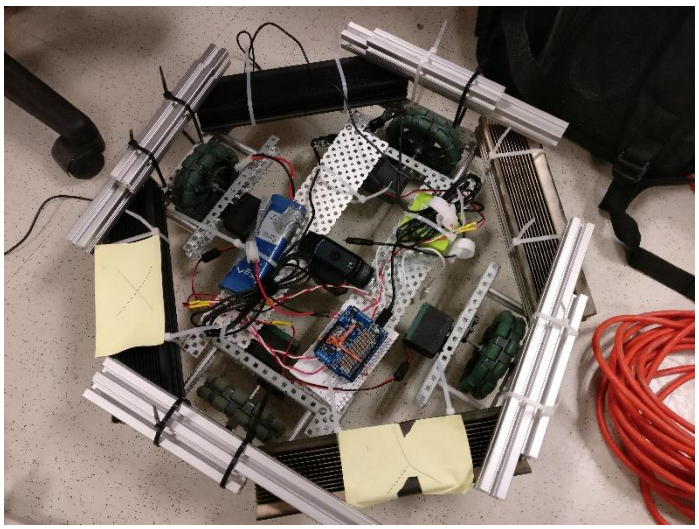

Figure 5 – Holonomic Robot with Camera

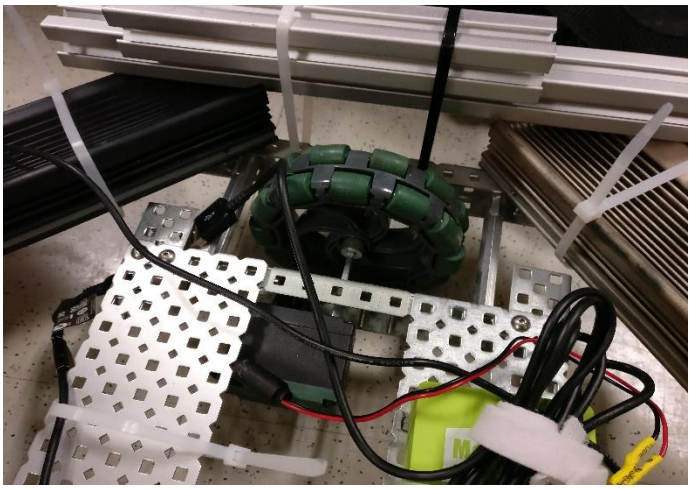

Figure 6 – Omni Wheel (wheel that also moves laterally)

We are right now in a stage of making another, more stable and more powerful mobile-node with better parts, motors and batteries. We have ported the entire robot software onto a Raspberry pi instead of having an Arduino in addition to a Raspberry pi.

Algorithm

In order to move from one location to other, we need to assign appropriate velocities to each of the four wheels of the robot. To solve this problem, we need to first see the geometry of the robot and make certain suitable assumptions [1].

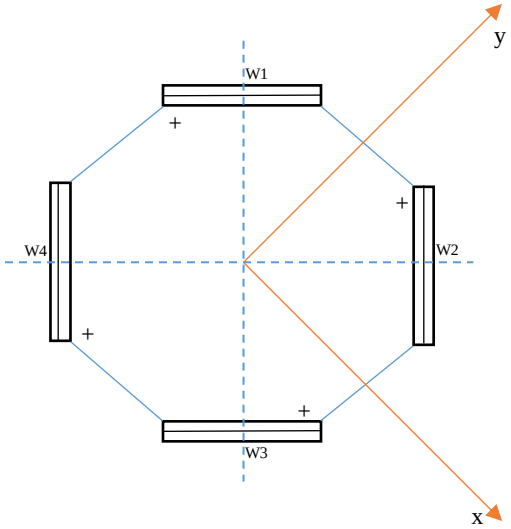

Figure 7 – Geometry of the Robot

The following assumptions need to be made in order to lay out the mathematics of the motion of the robot.

1. w1 to w4 – The four wheels are named as w1 to w4
2. '+' – The direction motion of wheels is said to be positive if it rotates away from '+' and negative if it rotates towards '+' sign
3. Vmax - Maximum velocity of each wheel (we set it to be 255 as it depends on the motor controller)
4. If robot motion along Y-axis is maximum and that along X-axis is 0, the direction of wheel motions is +ve, -ve, +ve and -ve respectively. i.e. Vy = [+1, -1, -1, +1].
5. If robot motion along X-axis is maximum and that along Y-axis is 0, the direction of wheel motions is Vx = [+1, +1, -1, -1].
6. If robot motion is a combination of all these motions, the wheel velocities (with sign) for all these pure motions are calculated and then added together to get the final velocities.

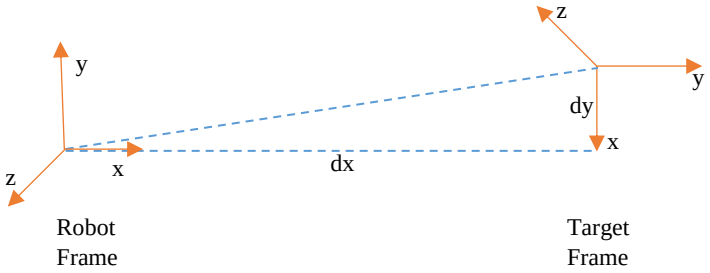

Figure 8 – Relative position of Robot and Target

As seen in the figure 8, the Robot frame is the position of the robot and the Target frame is the location to which the robot needs to move. We measure the relative position of these two frames with respect to each other in terms of Translational and Rotational components. We denote Translational component as dx, dy and dz and Rotational component as roll, pitch and yaw. These translational and rotational components can be positive or negative depending on which direction the robot needs to move and how much the robot needs to rotate in order to achieve the target frame position. The two motions of the robot, i.e. translational and rotational add up to give wheel velocities. We have limited it to 255 as mentioned above. Thus, Vmax is comprised of Vmax_trans and Vmax_ang. We divide Vmax as follows:

$$Vmax\_ang = (115/180) * yaw$$

$$Vmax\_trans = Vmax - Vmax\_ang$$

This ration, i.e 155/180 can be tweaked to suit specific velocity requirements. We found it by testing the robot in order to decide the distribution of Vmax into linear velocity and rotational velocity.Thus, angular component is more if yaw is more, i.e. if the robot has to rotate more to achieve the target orientation. The distance between robot and target frame is given as follows:

$$distance = \sqrt{dx^2 + dy^2}$$

We limit the distance to a threshold value, so as to limit velocity as an error protection measure. Here, threshold = 1.25 meters

Raw wheel velocities are calculated as follows:

wheels[i] = Vx[i]*dx + Vy[i]*dy

where, i from 0 to 3

We calculate the speed factor and scale factor in order to find wheel commands as follows:

scale_factor = max(abs(wheels[i]))

where, i from 0 to 3

speed_factor = interpolate distance from [0 to threshold] to [25 to Vmax_lin]

(‘25’ because it is the minimum velocity command that makes robot start moving)

Raw wheel velocities are translated to wheel commands using speed_factor and ascale_factor as follows:

commands[i] = ((wheels[i]/scale_factor)*speed_factor) + Vmax_ang

where, i from 0 to 3

In this way, we calculate the commands that are to be given to each of the motor, so that the robot moves in a particular direction at a particular velocity. We can combine these commands in order to achieve the given target to the robot. The user gives the target to the robot as three arguments, viz. x- coordinate, y-coordinate and orientation in degrees. The user gives several such targets in a list in a text file, which the software reads and conquers one by one. This use case can be tweaked and changed as per the user experience.

Future Work

The main thing to do in very near future is to build another, more stable robot with a greater battery that can power the node as well as robot. Another thing to do is to make collision avoidance system using sensors or more chilitags mounted on static obstacles like the poles. We also have to make a charging station that is placed at a known location, so that whenever the battery runs down a certain level, the robot will autonomously go in the charging station and start charging. The way to monitor battery level would not be difficult as it can be monitored on an input pin of the Raspberry pi and published as a ROS topic and monitored via ROS. The next step would be to get the mobile node in the grid and to make a detailed technical manual, so that the mobile node can be replicated easily.

References

1. Helder P. Oliveira, Armando J. Sousa, A. Paulo Moreira and Paulo J. Costa, Precise Modeling of a Four Wheeled Omni-directional Robot – *Proc. Robotica 2008 978-972-96895-3-6*
2. wiki.ros.org/ROS/Introduction
3. wiki.ros.org/ROS/NetworkSetup
4. github.com/chili-epfl/chilitags
5. github.com/chili-epfl/ros_markers
6. wiki.ros.org/ROSARIA