Portland State University

ECE 510: Advanced Embedded Robotics
Spring 2012

Final Project Report

GuideBot

Navigation & Localization

Dung Le
David Glover
Tochukwu Nwaoduh

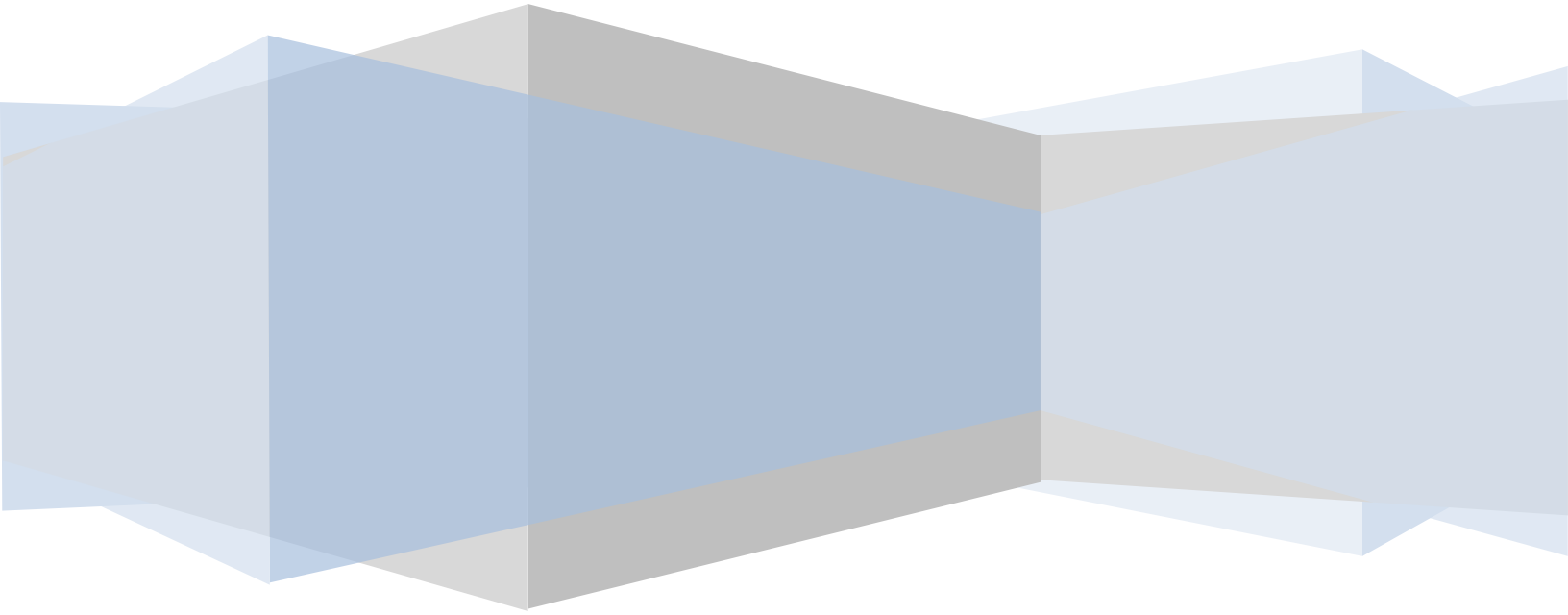


Table of contents

Contents

Introduction

The goal of this project was to allow autonomous navigation through the Fourth Avenue Building and Engineering Building by the GuideBot robot. Autonomous robot navigation is the ability of a robot to navigate an unstructured environment without continuous human supervision. With recent successes in planetary exploration rovers and ground robots, interest has increased in autonomous navigation. The ultimate goal of this project is to create a robot guide, GuideBot, which will take visitors on a guided tour of the Engineering Building.

The Mobile Robot Programming Toolkit (MRPT) library was used extensively for this project. MRPT provides an extensive, portable and well-tested set of C++ libraries and applications which cover the most common data structures and algorithms employed in a number of mobile robotics research areas such as Simultaneous Localization and Mapping (SLAM), Localization, Path Planning and Computer Vision.

The floor plan of the Fourth Avenue Building was saved as a bitmap file and stored as an object of the COccupancyGridMap2D class provided by the MRPT library. The COccupancyGridMap2D stored the map in the form of a probabilistic occupancy grid map. This grip map was then used for Path Planning and Navigation.

This project employed the Kinect, a motion sensing input device developed by Microsoft, used as a range finder. Results from the Kinect were used for obstacle avoidance, and localization.

Previous Work

The previous team mostly worked on getting readings from the sensors and the robot base as well as implementing basic functionality for driving, route finding, and obstacle avoidance.

The work used CActivMediaRobotBase object to implement robot navigation. The *robot.setVelocities(double linear_velocity,double angular_velocity)* CActivRobotBase method was employed for navigation, where **linear_velocity** is the velocity in meters per second (positive for forward or negative for backwards) and **angular_velocity** is in radians per second. Two navigation functions, *moveForward(double distance)* and *turn(double angle),* for turning and forward movement were implemented for

navigation - where **distance** is in meters and **angle** is in radians to turn. These functions are used in traveling from one node to another.

For route finding, a graph structure was used to represent the map of the Fourth Avenue Building. Each location in the graph is a node that contains the name of the location as well as the index. Edges were added to the nodes that are directly connected to each other. A method produced instructions on how to navigate from one node to another and back. If the robot cannot get to the destination from traveling any of its edges directly, a Graph class method returns a stack with the nodes that the robot must travel to get to its destination. The method uses the Depth First Search algorithm to build the stack, which it then pops off into a location array that can be traversed to get to the final destination.

The previous team encountered problems integrating the Kinect to their project. Their idea for a Kinect integration was to have images stored for each location, so that when the robot reaches its destination, or at intervals/checkpoints along the way, it would correct its location.

These works were documented in this Wiki site:

https://projects.cecs.pdx.edu/projects/ras-peoplebot

Our Tasks

- Get robot up and running
- Robot Navigation: drive robot from initial location to a given target assuming a map is provided.
- Robot Localization : determine actual location of robot during navigation using odometry and sensor reading

Implementation

Below is the high level system block diagram for the project. The User Interface allows a user to input commands to the Drive Controller, which in turn outputs results and a navigation map to the user. We also provide a 3D visualization of robot during runtime and simulation. The Path Planner, Localization and Collision Detection are described in more detail in this report.

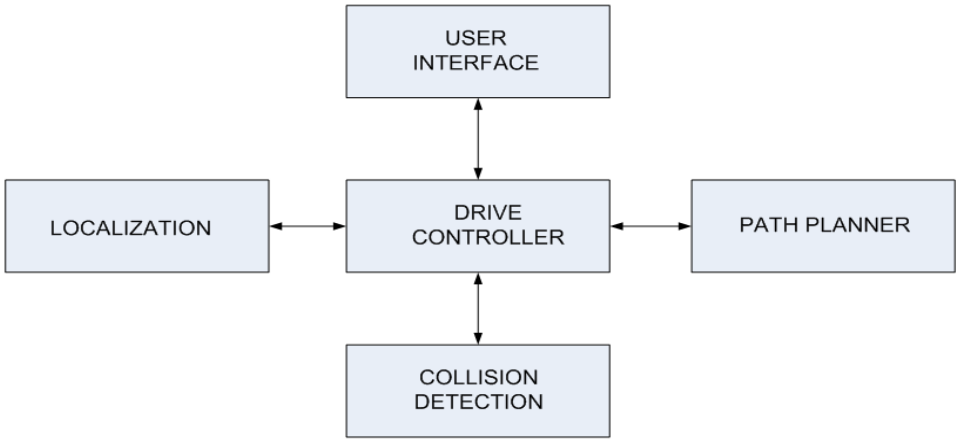


Fig 1: High level system block diagram

MRPT Library Overview

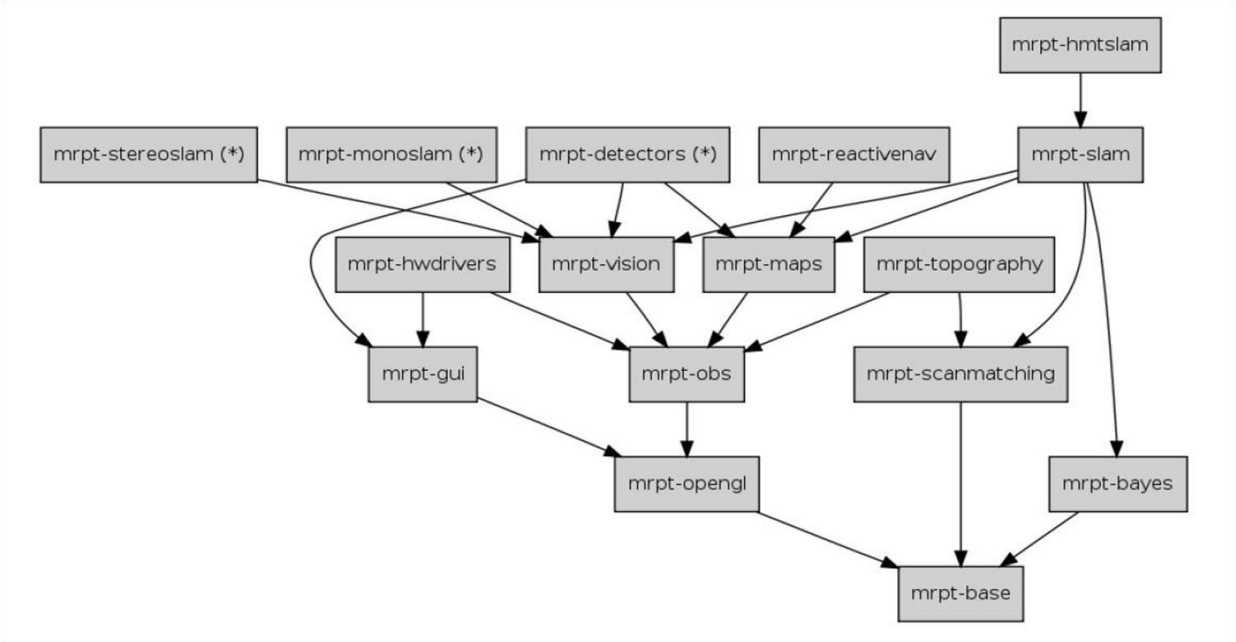


Fig 2: The MRPT library tree

Relevant MRPT Libraries

mrpt::hwdrivers::CActivMediaRobotBase: Our robot base driver for accessing robot base information and velocity control. Note that this driver is designed to work with robot base firmware via serial protocol. This class uses the Aria library.

mrpt::slam::CMonteCarloLocalization2D: Represents a Probability Density Function (PDF) over a 2D pose (x,y,phi), using a set of weighted samples.

mrpt::bayes::CParticleFilter: Our particle filter, executed on the probability density function. After the filtering process, the probability density function can be used for localization purpose.

mrpt::slam::COccupancyGridMap2D : Object to store the robot map in the form of a probabilistic occupancy grid map, it is one of the inputs of the probability density function.

Install MRPT and Compile Source

Instructions for installing MRPT (and cmake) can be found here

- http://www.mrpt.org/MRPT_in_GNU/Linux_repositories (recommended)
- http://www.mrpt.org/Building_and_Installing_Instructions
- https://projects.cecs.pdx.edu/documents/154 (cmake configuration)

Once mrpt and cmake have installed, under source folder just type *make*. After compiling successfully, to start the program, type

```
./guidebotNavigation
```

or

```
./guidebotNavigation < cmds.txt
```

(Where cmds.txt is a text file containing a list of commands, useful for repeating the same test multiple times without entering the starting and ending coordinates each time.)

Code Structure

Our source files include

- guidebotNavigation.cpp (main)
- CMakeLists.txt, (cmake buildscript)
- guidebotNavConf.ini (configuration file)
- FAB-LL-Central-200px.png (floorplan image)

There is also a command script for run-time tests.

In main(), four threads are spawned:

- thread_kinect
- thread_update_pdf
- thread_display
- thread_wall_detected

This is necessary to maximize CPU usage and ensure a long running job, such as update_pdf(localization), does not affect robot movement. These threads share common resources stored in TThreadRobotParams struct.

After spawning these threads, the main thread brings up the prompt to begin accepting commands from user (or other modules). Please refer to User Interface section for more details of the prompt interface.

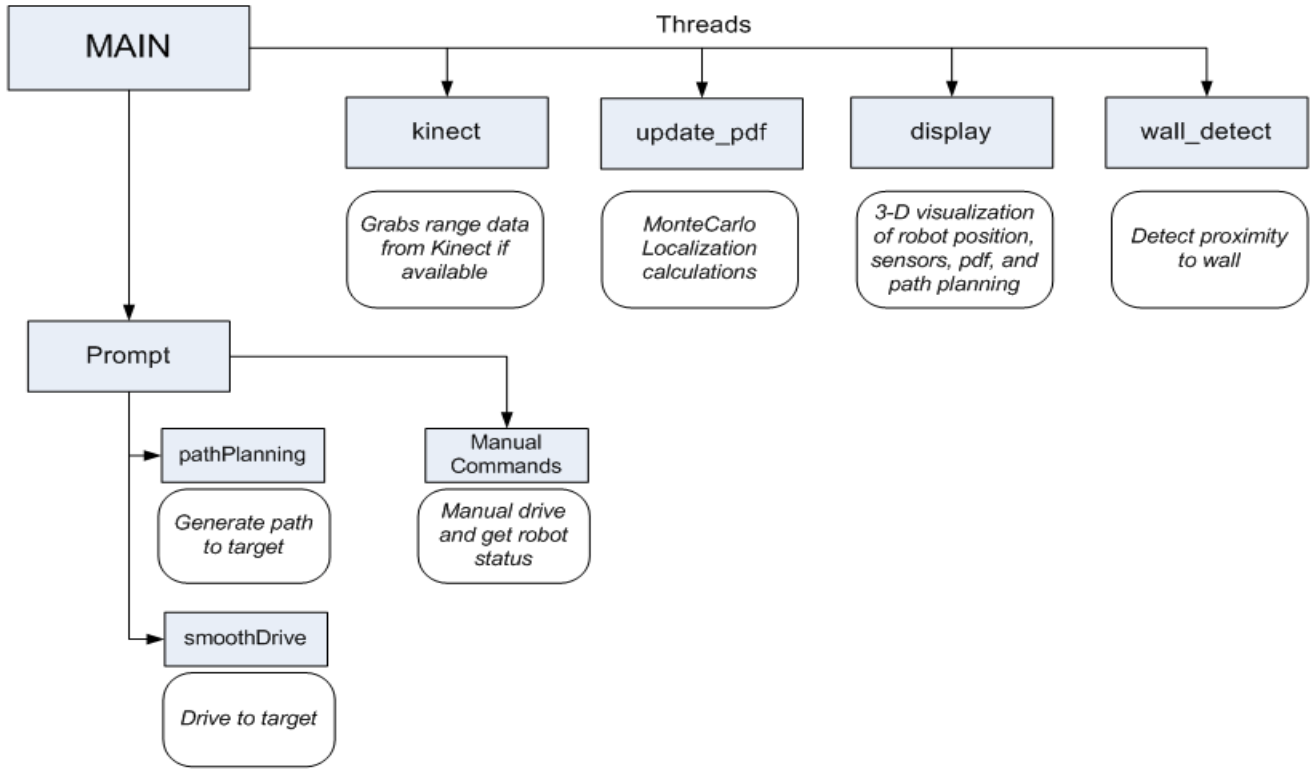


Fig 3: Main source structure

The Configuration File

MRPT provides a useful configuration class for adjusting configuration parameters in the main source code, avoiding modification and recompiling processes. This is very helpful during the testing and tuning process.

We put all necessary parameters in a separate text file named *guidebotNavConf.ini*

The configuration file format is as follows:

```
//comments
[section_name]
parameter_name = value
```

Reading these parameters from the main source code is straightforward, for example:

```
CConfigFile cf(CONFIG_FILE_NAME);
string value =
cf.read_string("section_name","parameter_name","default_val
ue", true);
```

For more information, please visit the MRPT API documentation#.

Interaction with hardware

Block diagram

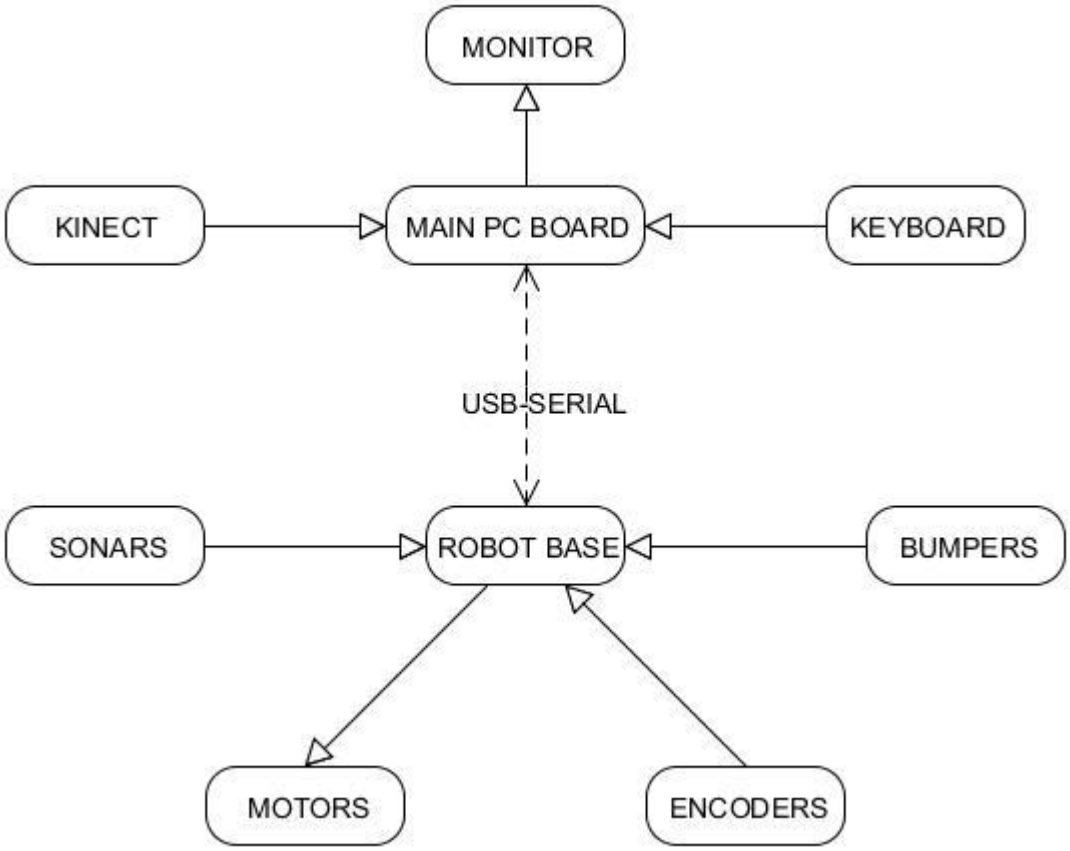


Fig 4: Hardware block diagram

Robot Base

```
#include <mrpt/hwdrivers/CActivMediaRobotBase.h>#
```

This software driver implements the communications for ActivMedia robotic bases for accessing to robot odometry, ticks counts, velocities, battery charge status, and sonar readings, as well as basic velocity control.

Only few setup steps are needed to use this driver:

- Connect robot base to main PCboard (a Linux machine) using the serial-to-USB adapter
- In software:

Declare

```
CActivMediaRobotBase   robot;
```

Configure serial port, note that all configuration are specified in *guidebotNavConf.ini* file

```
robot.setSerialPortConfig( port, port_baud );
```

Initialize

```
robot.initialize();
```

Robot Odometry

The MRPT library provides out-of-the-box kinematic model representing current robot location via state variables <x,y,phi>. The corresponding class is mrpt::poses::CPose2D

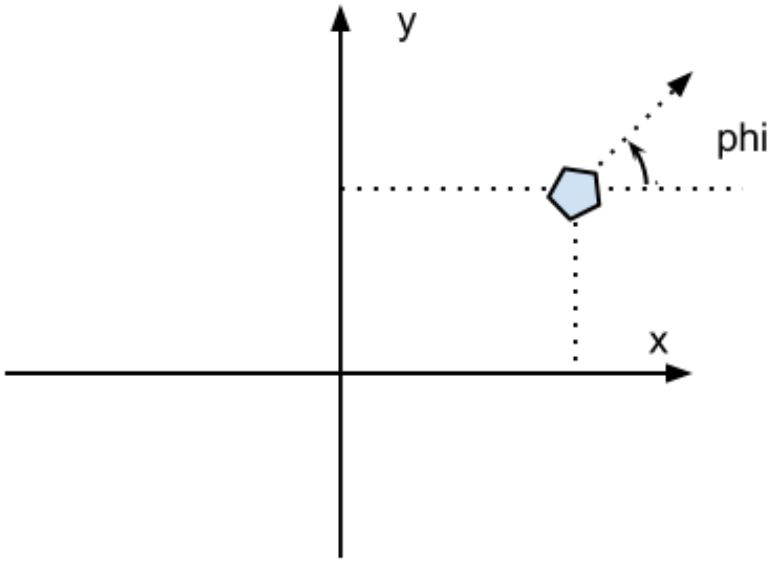


Fig 5. MRPT kinematic model

Methods used in our implementation

- `getOdometry()`
- `changeOdometry()`

We noticed there seems to be an issue with the *changeOdometry*() method. It does update the current pose to a specified pose, but fails to properly incorporate the phi value into this change. For example, if we set the robot pose to a 90 degree from its initial orientation (0 degrees when initialized) then move forward, the odometry will be

reported as if it was moving at 0 degrees. Our initial tests had all started with phi set to 0 degrees, so this wasn't found until later in our project when we started out facing 180 degrees. In those tests the robot would initially turn the correct way and move towards the target, however the display showed the robot moving in the opposite direction. We provide a passable fix for this issue by never setting the phi of the robot. Our implementation is to save the offset of the x, y and phi any time we need to set the robot position. Then we set the robot to x,y = 0,0 and phi = current robot phi (not actual phi). Whenever we get the odometry, we get the current odometry, rotate the x,y values by the offset phi angle and add the stored x,y values. To get the fixed odometry, we implemented the function fixOdometry.

- Setting new odometry:

```
CPose2D startOdo;
robot.getOdometry( startOdo );
startOdo.x(0);
startOdo.y(0);
CPose2D newOffset(newX,newY,newPhi - startOdo.phi());
thrPar.odometryOffset.set(newOffset);
robot.changeOdometry(startOdo);
```

- Getting current odometry:

```
CPose2D odo;
robot.getOdometry(odo);
fixOdometry( odo, thrPar.odometryOffset.get() );
```

Robot Sonar

The *getSonarsReadings()* method will give new readings from the current 16 sonars attached to robot base (the results stored in the CObservationRange object).

There are several problems we encountered with our robot sonar:

- The reading is only accurate if the ping direction is more or less perpendicular to the obstacle surface. (Otherwise we always get about 8 meters reported)
- The sensor poses (location and direction of sonars) are reported incorrectly from the base. To deal with the misconfigured sonars, we have created a function, adjustCObservationRangeSonarPose, which will replace the pose of each sonar returned by the robot base with the proper pose. Another function was written to create a deque of TSegment3D (MRPT line segment class) representing the 16 sonar readings. These line segments can be used to display the sonars in the 3D visualisation.
- Sonar is currently not supported by MRPT localization methods. This will be discussed in below sections.

Kinect

We use Kinect reading for localization (part of action-observation pair). We have run tests on sonar without success and suspected that MRPT library does not (fully) support sonar for localization purpose. The source code for the computeObservationLikelihood function used to determine the possibility of a given range reading coming from a given pose only accepts laser scan objects, the sonar readings are not accepted. Also there seems to be no (MRPT) documentation or previous works on localization using sonar. The common approaches from MRPT site are laser range finder and Kinect because of their superior outputs comparing to Sonars. Their limitation is however narrow scan window (vs 360 degree sonar's array scan). Also laser is an expensive option that we can't afford.

The MRPT provides good and well documented library for the Kinect. All information for setting-up and usages can found here http://www.mrpt.org/Kinect_and_MRPT

We are going to mention here what we have tried to get the Kinect working properly (The tested platform is ubuntu 10.4)

Kinect Setup

- MRPT must be installed first, see MRPT installation section
- install libusb-1.0-0-dev
  - `sudo apt-get install libusb-1.0-0-dev`
- setup for accessing the device without root privileges
  - create a file named 51-kinect.rules in /etc/udev/rules.d/ with the following content

```
# Install this file into /etc/udev/rules.d/ for accessing Kinect
# without root privileges.

#ATTR{product}=="Xbox NUI Motor"
SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02b0", MODE="0666"


#ATTR{product}=="Xbox NUI Audio"
SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02ad", MODE="0666"

# ATTR{product}=="Xbox NUI Camera"
SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02ae", MODE="0666"
```

This file can also be downloaded here http://mrpt.googlecode.com/svn/trunk/scripts/51-kinect.rules

Testing the Kinect

The easy way to test the kinect is to use the kinect mrpt-apps. After previous setup, under terminal use one of the following applications to see if kinect is setup properly:

- kinect-3d-slam
- kinect-stereo-calib
- kinect-3d-view

You can also download and compile the following example

http://www.mrpt.org/Example_Kinect_To_2D_laser_scan

Our Kinect Implementation

Most of the codes are reused from the kinect-to-2d-laser-demo example. We have a background thread that monitors the Kinect and outputs new range data continuously into a global parameter struct (with thread safe access):

```
void thread_kinect(TThreadRobotParam &p);
```

To get the current reading from this parameter struct:

```
static CObservation2DRangeScan* getKinect2DScan(const
TThreadRobotParam & p,  CObservation3DRangeScanPtr &
lastObs);
```

Notice that if there is no new reading, the method returns NULL.

Navigation

Path Planning

For navigation, we use the MRPT CPathPlanningCircularRobot class to generate a path from the current position of the robot to a target point. The usage of this class is demonstrated in the MRPT sample code pathPlanning. The sample code loads a provided map and generates a path between two defined points, then draws in circles to denote the steps on the path.

In order to use this function, we need to import a map into MRPT. We use a map of the building downloaded from the Facilities and Planning website#. The map was first converted to a bitmap image using GNU Image Manipulation Tool (GIMP). This bitmap is then imported into MRPT as a COccupancyGridMap2D object using the loadFromBitmapFile method, providing the resolution of the map in meters per pixel as an argument to the function. The resolution was determined from the pixel width of the map scale divided by the listed distance. Each pixel will be a cell of the map with a probability of being occupied based on the grayscale value of the pixel, where white denotes an empty cell and black denotes an obstacle.

When using the CPathPlanningCircularRobot object, we provide values for the attributes robotRadius and minStepInReturnedPath. The robotRadius value affects how close to the wall the path will be. The minStepInReturnedPath value will change how far apart each step in the path is. Calling the method computePath will give us a deque containing a list of points that will lead to our destination. This function works, and will provide a path to any locations we chose. Optimization could be done by using an altered map for path finding, to alter allowed areas to further control where the robot would travel. Also, this may be more than is needed for the task, and another function could be provided that would give paths using preset points (coming from corners, doorways, or other selected locations. The two could be used in unison if needed, perhaps the robot could use the computePath method when it is asked to go to areas that are not within its database of locations, while all other times it would rely upon the preprogrammed points.

Driving

Our first method used to follow the path was to use iterate through the provided list of points. First we would calculate the angle to turn to in order to face the next point, then drive forwards until the calculated distance from the current odometry to the target point was below a set threshold. This works, but is very unelegant to watch, as the robot spent a lot of time stopping and starting.

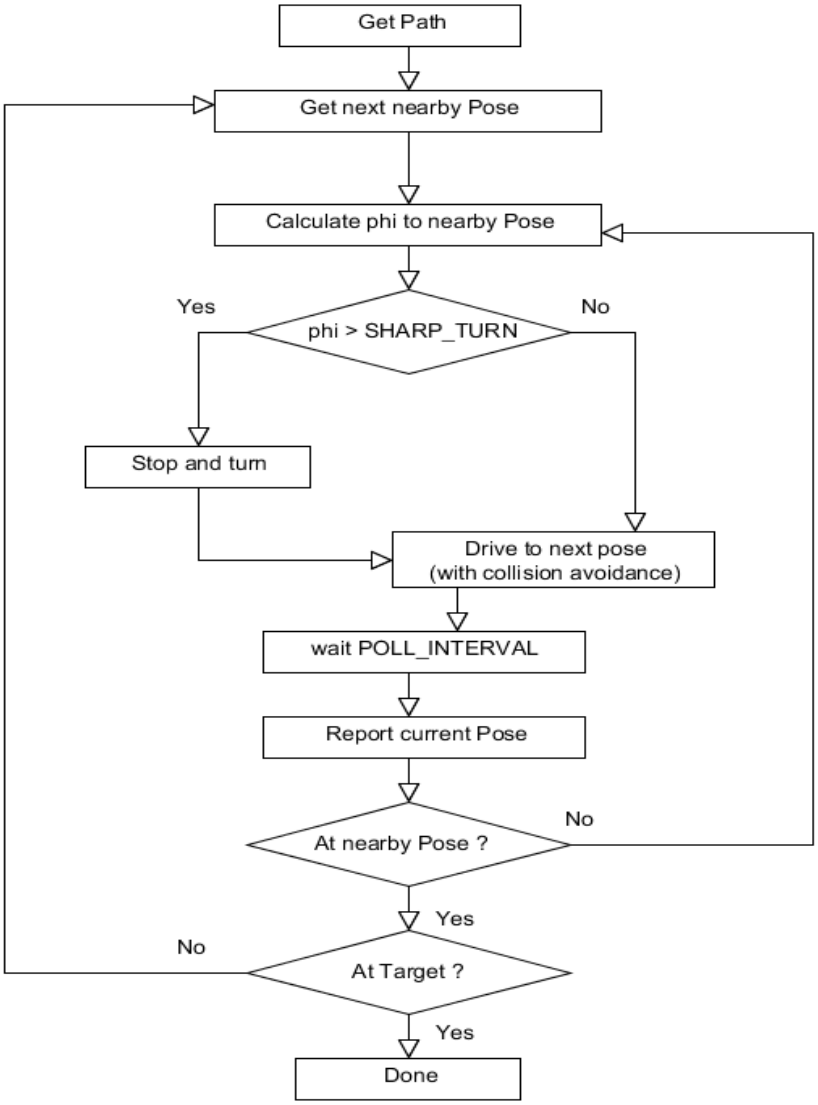


Fig 6: Smooth drive algorithm

We created the smoothDrive function as an improvement. As it is moving towards each target, it continuously checks the angle to the target and will adjust the forward velocity and angular velocity in order to connect the path with smooth arcs. If angle to the target is above a threshold given by SHARP_TURN, then the robot will stop and turn to face the target. If the angle to the target is below the SHARP_TURN threshold, the angular velocity will be set by the angle to the target divided by the ANGULAR_SPEED_DIV setting. As a method of collision avoidance, the smooth drive function also checks for flags set by the wall_detect thread if the kinect range readings indicate that there is an object to close on either side or the center. If one of the side flags is set, the robot will set speed to half and begin turning for a short period (sleep of 1000 miliseconds currently), after this it will resume travel to its current target point.

Localization

Monte Carlo Localization (MCL)

MCL uses a set of N weighted, random particles $S = \{s_i \mid i = 1..N\}$. Particle in MCL are of the type <<x y phi>, p> where x, y, phi denote a robot position, and p is a numerical weighting factor, represents the probability that the robot is currently at that position

The general algorithm is

1. Initialize the set of sample, either with locations evenly distributed and their importance weights are equal, or to a deterministic known location.
2. Repeat with the current set of samples:
    a. Move the robot by a fixed distance
    b. Update the location of *each* of the samples (using movement model).
    c. Take a sensor reading.
    d. Assign the importance weights of *each* sample to the likelihood of that sensor reading (using sensor model).
    e. Create a new collection of samples by re-sampling from the current set of samples based on their importance weights.(filtering)

This is illustrated as below.

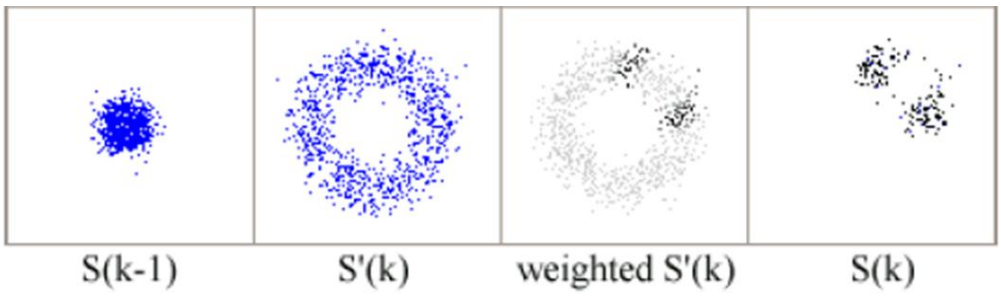


Fig 7: Filtering example

- Assuming initial robot location is at block center but *direction is unknown*, initialize the particle set S(k-1) with equal weights (1)
- Robot moves one step to the right corner (2a)

- Updating current particle location based on motion model gives the ring distribution (2b). Notice that direction unknown means we don't trust phi incremental (this is equivalent to setting minStdPHI = 360 degree, see motion model section below)

- Take sensor reading and re-assign weights based on sensor model (2c,d)

- Filter the current set based on certain threshold to get the new set S(k) (2e)

here is the another example:

http://www.cs.washington.edu/robotics/mcl/animations/an-238-12a.gif

Adaptive Sampling

Adaptive sampling determines the number of samples on-the-fly, is employed to trade-off computation and accuracy. The general idea is sampling stopped whenever the sum of weights p exceeds a threshold. Two common scenarios that change the set size:

- If the position predicted by odometry is well in tune with the sensor reading, each individual p is large and the sample set remains small.
- If the sensor reading carries a lot of surprise, as is typically the case when the robot is globally uncertain or when it lost track of its position, the individual p values are small and the sample set is large.

For more information of MonteCarlo localization, please refer to related papers [S. Thrun, D. Fox, W. Burgard, F. Dellaert]

Localization Implementation and Configuration

MCL algorithm is provided by the MRPT library via two important classes:

- CMonteCarloLocalization2D: represents the probability density function (pdf - sample set)
- CParticleFilter: invokes resampling (filtering) process using executeOn() method on the pdf.

These are used in our thread_update_pdf() thread. This thread is running in background so that it does not affect the driving process. Calculation time of executeOn() could take a couple to hundreds of millisecond based on current robot processing power and the initial number of particles. note that if adaptiveSampleSize

option is set, the number of particles in the current set can be reduced over time (only most likely particle remains, along with some random particles in case robot loses its direction completely)

CMonteCarloLocalization2D

Configuration for the CMonteCarloLocalization2D (pdf) object includes:

- The map, which is the image of our testing floorplan. The COccupancyGridMap2D class provides method to translate this image to an occupancy gridmap that later passed to our pdf. Note that this gridmap can be further configured via TInsertionOptions (for mapping) and TLikelihoodOptions (sensor model). We do not implement full SLAM hence the TInsertionOptions with not be discussed here. For TLikelihoodOptions please see sensor model section below.

  Method:

  ```
  gridmap.loadFromBitmapFile(MAP_FILE,resolution ,xCentralPixel,
  yCentralPixel);

  pdf.options.metricMap = &gridmap;
  ```

  Note: if xCentralPixel,yCentralPixel reset to -1, the central pixel is at center of the map

- TMonteCarloLocalizationParams. We currently use the KLD_options for the adaptive sample size (Refer to paper: D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte Carlo localization: Efficient position estimation for mobile robots,").

  Method:

  ```
  TMonteCarloLocalizationParams  pdfPredictionOptions;

  pdf.options = pdfPredictionOptions;
  ```

  Parameter used:

  ```
  [KLD_options]
  KLD_binSize_PHI_deg=10
  KLD_binSize_XY=0.10
  KLD_delta=0.01
  KLD_epsilon=0.01
  KLD_maxSampleSize=1000
  KLD_minSampleSize=100
  KLD_minSamplesPerBin=0
  ```

We haven't performed many experiments with these parameters, except for KLD_maxSampleSize KLD_minSampleSize, mostly because we spent much of our time getting the integrated system ready for testing. We also didn't spend enough time on reading the paper and couldn't find better documentation from the MRPT site. Further inspections on other parameters are recommended.

CParticleFilter

Configuration for the CParticleFilter (PF) is straightforward from ParticleFilterOptions class:

```
CParticleFilter::TParticleFilterOptions    pfOptions;

PF.m_options = pfOptions;
```

Fortunately, the detail for this option is well explained:

```
[PF_options]
// The Particle Filter algorithm:
//    0: pfStandardProposal
//    1: pfAuxiliaryPFStandard
//    2: pfOptimalProposal
//    3: pfAuxiliaryPFOptimal
//
PF_algorithm=0

// The Particle Filter Resampling method:
//    0: prMultinomial
//    1: prResidual
//    2: prStratified
//    3: prSystematic
resamplingMethod=0

// Set to 1 to enable KLD adaptive sample size:
adaptiveSampleSize=1

// Only for algorithm=3 (pfAuxiliaryPFOptimal)
pfAuxFilterOptimal_MaximumSearchSamples=10

// Resampling threshold
BETA=0.5

// Number of particles
sampleSize=1000
```

Motion and Sensor Model

Motion and Sensor Model are supported in MRPT library via these classes:

- `CActionRobotMovement2D::TMotionModelOptions`
- `COccupancyGridMap2D::TLikelihoodOptions`

For motion model, we used GausianModel, with following "noise" parameters for odometry:

- `minStdXY      = 0.02     // (meters)`
- `minStdPHI     = 1.0      // (degrees)`

These parameters can be adjusted based on correctness of robot encoders (vs real traveled distance) and the sampling time between pdf calculations, which is the POLL_INTERVAL.

Method:

```
CActionRobotMovement2D::TMotionModelOptions odom_params;

odom_params.modelSelection = CActionRobotMovement2D::mmGaussian;

odom_params.gausianModel.minStdXY  = minStdXY

odom_params.gausianModel.minStdPHI = minStdPHI
```

For sensor model, we looked at COccupancyGridMap2D::TLikelihoodOptions that were specified along with our gridmap:

```
COccupancyGridMap2D::TLikelihoodOptions likelihoodOption;

gridmap.likelihoodOptions = likelihoodOption;
```

Defined parameters:

```
[MetricMap_occupancyGrid_00_likelihoodOpts]
likelihoodMethod=4          // 0=MI, 1=Beam Model, 2=RSLC, 3=Cells
Difs, 4=LF_Trun, 5=LF_II
LF_decimation=20
LF_stdHit=0.20
LF_maxCorrsDistance=0.30
LF_zHit=0.95
LF_zRandom=0.05
LF_maxRange=15
LF_alternateAverageMethod=0
MI_exponent=10
MI_skip_rays=10
MI_ratio_max_distance=2
```

```
rayTracing_useDistanceFilter=0
rayTracing_decimation=10
rayTracing_stdHit=0.30
consensus_takeEachRange=30
consensus_pow=1
```

These options are somewhat explained in the MRPT site#. Notice that we took the default model from pf-localization application which probably used a "real" laser range finder, so this should not be taken for granted in further development. Again, this is possibly a flaw in our implementation because we haven't yet figured out:

- The exact meaning of each parameters
- How to match them with the Kinect characteristics itself.

In our implementation, the pair of action and observation is taken periodically (every POLL_INTERAL) then fed to the particle filter's execution method:

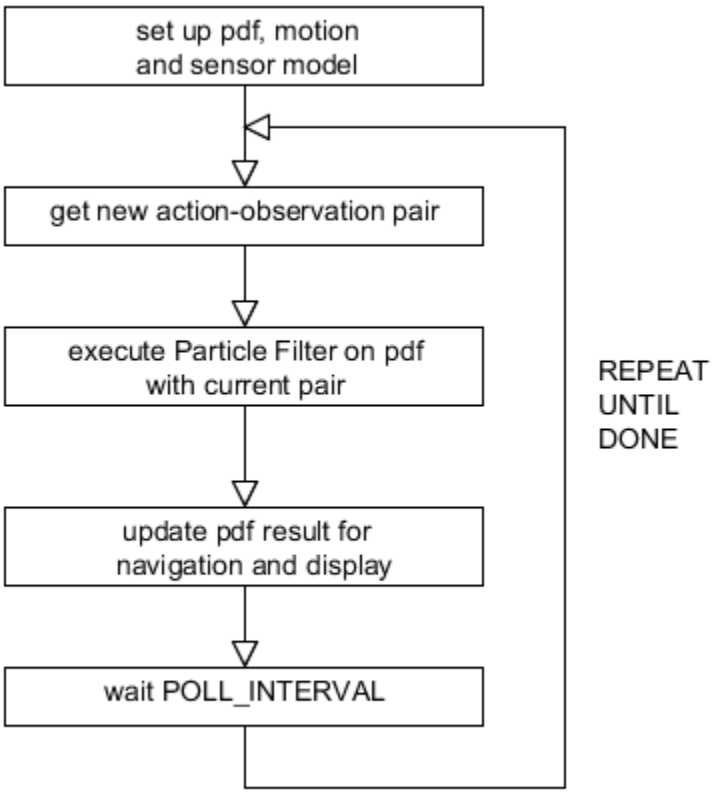


Fig 8: The update pdf thread's flow diagram

Action (odometry):

```
CActionCollectionPtr actions;

odom->computeFromOdometry(currentOdo - previousOdo, odom_params);

actions->insert(*odom);
```

Observation (Kinect):

```
CSensoryFramePtr senFrame;

CObservation2DRangeScan* obs_2d = getKinect2DScan(p, last_obs);

obsPtr.setFromPointerDoNotFreeAtDtor(obs_2d);

senFrame->insert(obsPtr);
```

Execution call:

```
PF.executeOn(
           pdf,
           actions.pointer(),    // Action
           senFrame.pointer(),   // Obs.
           &PF_stats        // Output statistics
);
```

Once the calculation finished, the following methods can be used to extract the MCL prediction:

```
pdf.getMean( pdfEstimationPose );

pdf.getMostLikelyParticle();
```

Current Results on Localization

We first tested MCL with Sonars. However later we found out that the MRPT did not fully support localization using sonar. The result after filtering was a particle distribution with all zero weights.

We noticed that all MRPT examples on localization seem to prefer the laser range finder. Since buying a laser for this project is not a possible option, and there is an alternative method to fake laser scan using the Kinect, with a little change, we replaced sonar observation with Kinect observation.

The pdf calculation provided much better results with the Kinect and showed reasonable particle distribution on the given map, at least for a few-meter-run test. This proved that we on right the track although the accuracy of prediction is questionable.

Localization Summary

- We need a good sensor model for the Kinect.
- More experiment with different parameter sets. We have provided a simulator for this purpose.
- Sonar reading can be used along with Kinect reading in sensory frame, but a filter is required and the library-support issue must be fixed.
- Once localization has provided accurate prediction, we could overwrite current robot odometry with this prediction. This completes the navigation and localization process.

User Interface

Command Prompt

When the software is run, it will display a command menu in the terminal. The menu will be displayed after each command is completed. This structure comes from the pioneerRobotDemo# code from MRPT, with a few additions of our own (changing pose, changing target, path following) and is controlled by a simple loop in main.

```
Press the key for your option:

w/s   : +/- linear speed

a/d   : +/- angular speed

space : stop

o     : Query odometry

n     : Query sonars

b     : Query battery level

p     : Query bumpers

P     : Follow Path

e     : Enter new current pose

t     : Enter new target for path Planning:

x     : Quit
```

Changing the linear or angular velocities will add a set amount to the current velocities, and allow manual driving of the robot using the keyboard (a wireless keyboard helps). <Space> will reset the current linear and angular velocities. One note on these commands is that the odometry is not retrieved and so the 3D display will not show robot movement unless the ‘o’ command is given to query odometry, which will display the current x,y location, phi - direction the robot is facing, current linear and angular velocities in meters / second and degrees / second, and current left and right encoder ticks. Query sonars will give you the current range values on the sonars and also show them on the 3D display. Query battery level will give the current voltage of the battery,

12.5 is a fully charged battery. Query bumpers did not give us valid readings, but we did not do any hardware troubleshooting to find the cause.

The 'P' command will enter the navigation state, and starts with finding a path from the current pose (set by the Enter new current pose command) to the target pose (set by the Enter new target for path Planning command), after which it will use the smoothDrive function to follow the path. Manual commands are not accepted while in the path following state, space bar in the 3D display window will return to the manual mode.

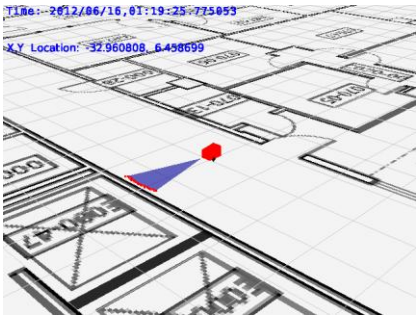
This prompt is also useful if our navigation module need to communicate with other modules to control robot behavior (system integration).

To save time during testing, we redirect commands from a text file, for example:

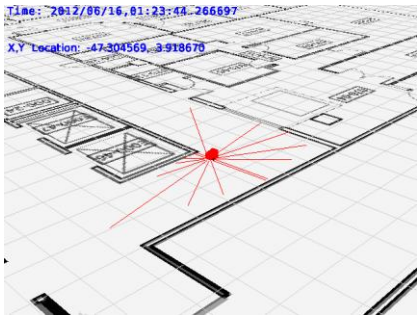
```
./guidebotNavigation < cmds.txt
```

3D Display

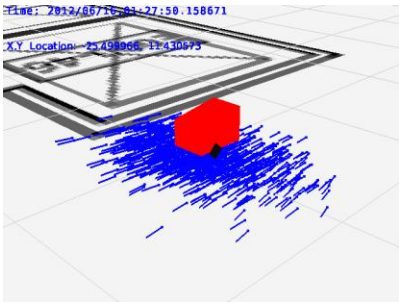
The 3D display was built on the base of the display3D# sample code from MRPT. It was moved into a separate thread so that it can update the display in the background. You can zoom in and out using the scroll wheel on a mouse, and click and drag the window to rotate the display. The map coordinates of the mouse position are displayed on the second line of text, as well as the current status of the thread_wall_detect variables if there is range values less than 1 meter on the kinect.



Kinect range display



Sonar range display



PDF display

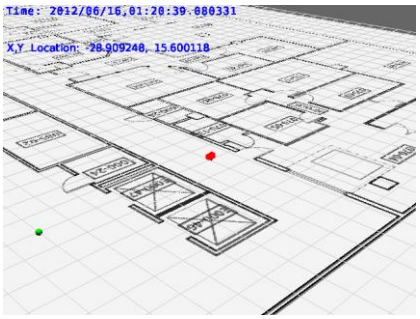
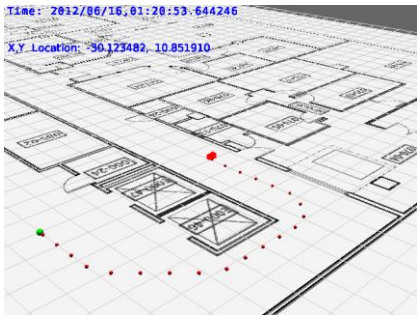
		
Target point shown as green sphere.	Path to target shown as a series of smaller red spheres.	

Fig 9: 3D displays

Samples of the 3D display are shown above. The data shown in the window is all updated through the TThreadRobotParam struct. A subset of the struct is listed below that are used for the display.

- Current robot pose and current target pose, these are set by the main thread. The currentOdo value is not updated automatically in the manual state, use the 'o' show odometry command in the terminal to update the display. The current odometry is updated automatically in the smoothDrive function that follows the path to the target.

```
mrpt::synch::CThreadSafeVariable<CPose2D>                    currentOdo;

mrpt::synch::CThreadSafeVariable<CPose2D>                    targetOdo;
```

- The PDF, display is changed only when displayNewPdf is set to true, if displayClearOldPdf is set to false, it will not delete the previous PDF (it will show the current and all previous PDF objects). The pdf is updated in the thread_update_pdf thread.

```
mrpt::synch::CThreadSafeVariable<CObjectPtr>                 pdf;

mrpt::synch::CThreadSafeVariable<bool>                     displayNewPdf;

mrpt::synch::CThreadSafeVariable<bool>                     displayClearOldPdf;
```

- The path to be followed, displayed only when displayNewPath is set to true, if displayClearOldPath is set to false, it will not delete the previous path (it will show the current and all previous paths). The path is created when the 'P' Follow Path command is given in the terminal. To change the displayClearOldPath

setting, in the 3D window press the lowercase ‘p’ to set it to true and the uppercase ‘P’ to set it to false.

```
mrpt::synch::CThreadSafeVariable<std::deque<poses::TPoint2D> >     thePath;

mrpt::synch::CThreadSafeVariable<bool>                displayNewPath;

mrpt::synch::CThreadSafeVariable<bool>                displayClearOldPath;
```

- The sonar range display, updated when displayNewSonars is set to true, if displayClearOldSonar is set to false, it will not delete the previous sonar display (it will show the current and all previous sonars). The deque of line segments needs to be created by the displaySonars function because there is no MRT method of displaying sonars as a 3D object. This is set in the main thread, either manually with the terminal command ‘n’ Query sonars, or automatically during the smoothDrive function. To change the displayClearOldSonar setting, in the 3D window press the lowercase ‘s’ to set it to true and the uppercase ‘S’ to set it to false.

```
mrpt::synch::CThreadSafeVariable<deque<TSegment3D> >        sonars;

mrpt::synch::CThreadSafeVariable<bool>                displayNewSonars;

mrpt::synch::CThreadSafeVariable<bool>                displayClearOldSonar;
```

- Obstacle display, to show when kinect ranges less than 1 meter are seen to the left, right or center. These values are set from the thread thread_wall_detect.

```
mrpt::synch::CThreadSafeVariable<bool>                leftObstacle;

mrpt::synch::CThreadSafeVariable<bool>                rightObstacle;

mrpt::synch::CThreadSafeVariable<bool>                centerObstacle;
```

Commands are also accepted in the 3D window, as seen above. These commands are different than the commands given in the terminal menu, and it would be a good next step to integrate the two interfaces. <Space> in the 3D window will stop the main thread if it is running the ‘P’ Follow Path command. <ESC> will trigger the the 3D display thread and the other threads to quit by setting the quit variable in the TThreadRobotParam struct.

Conclusion

Contributions

In this project, we have implemented robot Navigation, Localization and provided a friendly User Interface, as contribution to the PSU Robotics Peoplebot-Guidebot ‘s project.

Navigation is conducted on a given map and computed path, along with simple collision avoidance. Localization uses robot odometry and Kinect observation to execute Particle Filter algorithm on MonteCarlo ’s probability density function, giving possible locations of robot during navigation and their corresponding weights. This in turn can be used as feedback to correct robot odometry errors. Our User Interface gives a 3D view of current robot location and movement as well as sensor readings and particle distribution. This interface can also accept character commands from outside modules for system integration purposes.

Remaining Issues

We rely on MRPT library throughout the whole project since it has been initiated from previous teams. It is also open-source and fairly documented so that the implementing process was relatively straightforward. However we have observed a few bugs or unsupported features as mentioned from this library which we have to work around before we can get some reasonable results.

Currently the Navigation function does not support wall following and completed obstacle avoidance, since this was not our main focus in this project. As a result, robot movement is not as robust as expected. However, with the existed framework, we believe adding these abilities should be easy to do.

The Localization function works on principle but has not been well-tuned. The prediction results are somewhat acceptable but are not reliable enough to correct robot odometry.

Future Works

As mentioned, Localization was not fully implemented in this project. The user interface allows for simulation of particle distribution along the robot's path, but this is currently not employed for navigation error correction. Future works should be focusing on tuning the particle filter using different algorithm and modelling parameters to obtain better reliability, before updating robot odometry based on pdf calculations.

Sonar should also be used for wall and obstacle avoidance. Obstacle avoidance implemented using the Kinect is currently not robust enough and the Kinect fails to provide valid range data when the robot is too close to an obstacle. Sonars can be employed here to make up for the Kinect's limitation.

We envision two options:

1. The Kinect and Sonars can be used concurrently for obstacle avoidance, where the Kinect detects obstacles at a high elevation (such as the upper end of a table), while Sonars detects obstacles at lower elevation (such as the legs of the table). Sonar ranges can be made available to the wall detection thread so that it will have more complete information.

2. The Kinect is used for obstacle avoidance by default, and the Sonars are only activated when the Kinect detects obstacles whose distance from the robot is below a certain threshold, or Kinect reading are found to be invalid due to its proximity to an obstacle.

Troubleshooting

1. The robot reset button is your friend.

2. Cool startup: no beeping sound from robot base (although there may be ticking sounds from sonar). Robot may be left uncharged while not in used and will not working properly if battery level is below 12V. Usually we left robot charged overnight before testing. If robot batteries are in good condition and the power LED is on, try to press and hold the reset key to get robot to the testing sequence. If power LED is not on, check the automotive fuse on the robot main power supply board. (Note that from the command prompt, type "b" to get current battery status.)

3. Bad reading/checksum: serial connector. This indicates the serial adapter might not work properly. Also make sure that the connection is secured. The only working Baudrate we could use so far is 9600.

4. Power for kinect: which one is which? Kinect required additional power supply beside USB power. We have wired robot base power supply to the *kinect-specific usb-liked* connector(female). Do NOT remove this unless you need to replace it, in which case, be careful about the wire colors (of the connector):

   BROWN = VCC

   WHITE = GND

   Otherwise, plug this connector onto its corresponding male connector to power the kinect up. Finally plug the kinect's standard USB connector to the PC board.

5. Program crashed: segmentation fault. Yeah, it's probably a leftover bug in our program. Are we freeing a referencing object? Notice that we have several threads sharing the common resources in TThreadRobotPaRam struct. Although it utilizes CThreadSafeVariable (get/set interface) for synchronized accesses, objects in this struct must have explicit COPY (=) operator defined, otherwise the get/set is operated on a shallow copy.

6. Robot reports wrong odometry: If the robot initially turns the correct way to follow the path, but shows on the display that it is moving in a different direction it is likely that the changeOdometry function was used to change the robot's phi. This does not seem to have been implemented properly in the MRPT library, see our temporary fix in previous sections.

References

1. Mobile Robot Programming Toolkit, mrpt.org
2. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. Dieter Fox, Wolfram Burgard, Frank Dellaert, Sebastian Thrun.
3. Corridor navigation and wall-following stable control for sonar-based mobile robots. Ricardo Carellia, Eduardo Oliveira Freire.
4. Pdx-Ras peoplebot wiki site: https://projects.cecs.pdx.edu/projects/ras-peoplebot