Warning Concerning Copyright Restrictions

The Copyright Law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted materials.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be used for any purpose other than private study, scholarship, or research. If electronic transmission of reserve material is used for purposes in excess of what constitutes "fair use," that user may be liable for copyright infringement.
Dynamic Obstacle Avoidance on a Self-balancing Robot Platform

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts in Computer Science and the Honors Program

by

Zakary Littlefield

Dr. Kostas Bekris, Thesis Advisor

May, 2011
We recommend that the thesis prepared under our supervision by

ZAKARY LITTLEFIELD

entitled

Dynamic Obstacle Avoidance on a Self-Balancing Robot Platform

be accepted in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE, COMPUTER SCIENCE

Kostas Bekris, Ph.D., Thesis Advisor

Tamara Valentine, Ph. D., Director, Honors Program

May, 2011
Abstract

Autonomous tasks are increasingly becoming part of our everyday life, whether in a factory floor where robotic manipulators manufacture goods, or when cars acquire the capability of parking themselves. These tasks often involve a human operator that has some high level control over the system. This can lead to situations where the human operator believes that the system is safe, when this might not be the case. This is why robust control algorithms need to be implemented that provide safety guarantees when a mechanical device is teleoperated by a human user. These algorithms can intervene when an unsafe choice is made to protect the operator and the system itself. The challenge that this work specifically focuses upon is dynamic obstacle avoidance by a robotic unit that is guided by a human user. Dynamic obstacles are objects that move in the environment independently from the robot and can potentially raise safety concerns for the robotic platform. In order to detect these obstacles in the environment, sensor data along with some probabilities need to be utilized to infer the magnitude and direction of an object’s speed. This project is primarily concerned with this estimation and methods to do this accurately.
Acknowledgements

To my project group members,

Winway Pang, Nathan Silva, and David Leblanc,

To my good friend for all the hardware help,

Brian Hamilton,

To my family,

Donna, Gene, Seth, and Spencer Littlefield,

Thank you.
Table of Contents

Abstract........................................................................................................................... i
Acknowledgment...................................................................................................... ii
Table of Contents............................................................................................................ iii
Introduction................................................................................................................ 1
Dynamic Obstacle Avoidance........................................................................................ 2
Main Software Systems............................................................................................... 2
Hardware Implementation............................................................................................ 3
Appendix Descriptions.............................................................................................. 4
Conclusion....................................................................................................................... 6
GUI.cpp......................................................................................................................... 7
ObstacleDetect_node.cpp............................................................................................. 13
SegwayJoy.cpp............................................................................................................ 29
SensorViz.cpp.............................................................................................................. 31
VFH.cpp....................................................................................................................... 34
**Introduction**

Safety is a primary concern when working with machinery and other such technology. Being able to accurately determine when these machines will become unsafe is of upmost importance. This project explores issues that arise when implementing safety algorithms to run on a real system.

For most algorithms in the motion safety domain, simulations are conducted to test the safety guarantees. Then, determined by available time and publication timelines, a test on a physical system may be conducted. This may lead to algorithms that are shown to work in simulation, but may or may not work when implemented on a real system. This can be due to coding errors or an inaccurate model of how a real system would move. Either way, it seems to be necessary to implement the approach on a real system in order to fully test the guarantees that an algorithm can provide.
Dynamic Obstacle Avoidance

An algorithm that provides dynamic obstacle avoidance allows for a moving body to be able to avoid obstacles that are moving in the environment. This is a major extension from the simpler case of static obstacle avoidance because a time element now has to be considered. Objects can be in different places at different times, making it more difficult to determine when or if the system will collide with obstacles.

There are two major families of algorithms in this domain. There are the algorithms that create a plan for the entire trajectory that the system will execute and then, when a collision is detected in the near future, the system will replan into the future so as to avoid that obstacle. Then there are reactive techniques, ones that don’t compute a full path, but instead head in a desired direction and always focus on avoiding obstacles. Both have advantages, but we chose to implement a reactive technique due to simplicity of the algorithms.

Main Software Systems

To make our code as modular, or replaceable, as possible, we decided to implement using the ROS software system. ROS (Robot Operating System) is a software framework that allows for inter-process communication and data sharing. This also allows any future research groups to be able to replace any of the “nodes” in the system to fit their needs.

For development ease, we use Ubuntu, a distribution of the Linux operating system. This allows ease of use because we can control many different features of the system fairly easily using configuration files and other tools. To edit our source code, we use a
modified version of gedit, the main text editor in Linux. These modifications allow for just as much control over the code as an elaborate editor like Visual Studio or Netbeans.

**Hardware Implementation**

We were supplied with ample hardware to make the project a reality. Our main test hardware was the Segway Robotic Mobility Platform (RMP) 200. This platform is a differential drive, self-balancing system. A differential drive system is one that can control the two wheels on the robot independently from one another.

To sense the environment, we use a SICK LMS 100 laser range scanner. This sensor uses infrared beams to measure the distance to the nearest obstacle within a 20-meter radius. These beams also can be sent in a 180-degree arc in front of the robot.

To provide control to the user, we use an Xbox 360 controller and piece of software in ROS to provide inputs to the Segway platform.
Appendix Descriptions

In the appendices of this documentation, a set of source files can be found. These source files are the main contribution made during the course of this project. They provide the functionality to implement obstacle avoidance algorithms on the Segway Platform.

GUI.cpp

This class was created to be an easy to use interface between a user and the ROS-nodes. This class’s GUI contains a separate button for every node in our framework, which can be toggled on and off separately while the system is running. This class was built on the GTK library.

ObstacleDetect_node.cpp

This source file is the foundation for implementing dynamic obstacle avoidance algorithms. Using Bayesian filters, this code utilizes previous and current laser sensor data to estimate the location and velocity of obstacles. This class also includes a GUI in the form a window that displays a grid space with blocks of black and red. The black blocks represent detected obstacles, which are estimated to be static, while the red blocks indicate obstacles that are moving. On the moving obstacles, green lines indicate the direction of travel.

SegwayJoy.cpp

This class was created to be the interface between the joystick driver and the Segway driver allowing for users to directly control the Segway without any intervening software.
This class subscribes to the joystick publisher and publishes twist messages, which contain the velocity information needed by the Segway driver.

SensorViz.cpp
This class was created to allow for a debugger or user to easily interpret the laser sensor data. This class uses the OpenGL library to create a window containing lines distributed around the bottom center representing every individual laser measurement while red rings are drawn around the same center to indicate every meter of distance. This class does not publish any messages and only subscribes to the laser publisher.

VFH.cpp
This source file is responsible for static obstacle avoidance. It was written as a baseline to be replaced by other obstacle avoidance algorithms in the future. When running, this class takes the place of the SegwayJoy class. This class subscribes to the joystick and laser publishers and publishes Twist messages, velocity and angular velocity, for the Segway driver to execute. This class implements the Vector Field Histogram avoidance algorithm, which determines feasibility of a desired control. It does this by directly checking the laser data for valleys, or contiguous laser measurements that exceed a threshold value of one meter. The algorithm checks the user’s desired control against the measurement data to see if it lies within a valley or not. If not, the algorithm searches for the valley that is closest to the user’s desired control and provides a control through the center of that valley.
Conclusion

The main goal of this project was to create a framework to implement obstacle avoidance algorithms on the Segway RMP 200 platform. My group members and I have created the basis that future work can use to easily continue this line of work. We have provided a way to detect moving obstacles in an environment, along with basic driver software to get the system to run smoothly.
GUI.cpp

#include <gtk/gtk.h>
#include <stdio.h>
#include <stdlib.h>

//booleans to keep the state of the buttons
bool segway_node;
bool segway_control_node;
bool joystick_node;
bool sensor_node;
bool sensorviz_node;
bool obstacles_node;
bool avoidance_node;

G_MODULE_EXPORT void on_window_destroy(GtkObject *object, gpointer user_data)
{
    //kill all ros nodes
    system("rosnode kill --all &");
    if(segway_node)
    {
        segway_node=false;
    }
    if(segway_control_node)
    {
        segway_control_node=false;
    }
    if(joystick_node)
    {
        joystick_node=false;
    }
    if(sensor_node)
    {
        sensor_node=false;
    }
    if(sensorviz_node)
    {
        sensorviz_node=false;
    }
    if(obstacles_node)
    {
        obstacles_node=false;
    }
}
if(avoidance_node)
{
    avoidance_node=false;
}

//kill the roscore process
system("killall python &");

//quit the gtk window
gtk_main_quit();

///Function callbacks for each button
G_MODULE_EXPORT void on_button1_clicked(GtkObject *object, gpointer user_data)
{
    if(segway_node)
    {
        system("rosnode kill test_node &");
        segway_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
    {
        system("rosrun segwaytry test_node &");
        segway_node=true;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"On");
    }
}

G_MODULE_EXPORT void on_button2_clicked(GtkObject *object, gpointer user_data)
{
    if(segway_control_node)
    {
        system("rosnode kill segwayjoy &");
        segway_control_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
{    system("rosrun unrr segway segwayjoy_node ");
    segway_control_node=true;
    GtkWidget* b=(GtkWidget*)(object);
    gtk_button_set_label(b,"On");
}

G_MODULE_EXPORT void on_button3_clicked(GtkObject *object, gpointer user_data)
{
    if(joystick_node)
    {
        system("rosnode kill joy_node ");
        joystick_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
    {
        system("roslaunch unrr segway joy.launch ");
        joystick_node=true;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"On");
    }
}

G_MODULE_EXPORT void on_button4_clicked(GtkObject *object, gpointer user_data)
{
    if(sensor_node)
    {
        system("rosnode kill lms1xx ");
        sensor_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
    {
        system("rosrun LMS1xx LMS100 ");
        sensor_node=true;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"On");
    }
}

G_MODULE_EXPORT void on_button5_clicked(GtkObject *object, gpointer user_data)
{
    if(sensorviz_node)
    {
        system("rosnode kill sensor ");
        sensorviz_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
    {
        system("rosrun unr_segway sensorviz ");
        sensorviz_node=true;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"On");
    }
}
G_MODULE_EXPORT void on_button6_clicked(GtkObject *object, gpointer user_data)
{
    if(obstacles_node)
    {
        system("rosnode kill obstacle_detector ");
        obstacles_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else
    {
        system("rosrun unr_segway obstacle_reader ");
        obstacles_node=true;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"On");
    }
}
G_MODULE_EXPORT void on_button7_clicked(GtkObject *object, gpointer user_data)
{
    if(avoidance_node)
    {
        system("rosnode kill vfh ");
        avoidance_node=false;
        GtkWidget* b=(GtkWidget*)(object);
        gtk_button_set_label(b,"Off");
    }
    else

void system("rosrun unr_segway vfh ");
avoidance_node=true;
GtkButton* b=(GtkButton*)(object);
gtk_button_set_label(b,"On");

int main(int argc, char *argv[])
{
    //initialize all the booleans to false
    segway_node=false;
    segway_control_node=false;
    joystick_node=false;
    sensor_node=false;
    sensorviz_node=false;
    obstacles_node=false;
    avoidance_node=false;

    //holder objects to read from the .glade file
    GtkBuilder *builder;
    GtkWidget *window;
    GtkWidget *segway;
    GtkWidget *seg_driver;
    GtkWidget *joystick;
    GtkWidget *LMS;
    GtkWidget *sensorviz;
    GtkWidget *obstacles;
    GtkWidget *avoidance;

    //initialize the window
    gtk_init(&argc, &argv);

    //build the gui interface from the glade file
    builder = gtk_builder_new();
    int val=gtk_builder_add_from_file(builder,"/home/rmp/Desktop/unr_segway/src/GUI.glade" ,NULL);

    //get each element from the glade file that was built
    window = GTK_WIDGET(gtk_builder_get_object (builder, "main_window");
    segway = GTK_WIDGET(gtk_builder_get_object (builder, "segway_driver");
seg_driver = GTK_WIDGET(gtk_builder_get_object(builder, "segway_control"));
joystick = GTK_WIDGET(gtk_builder_get_object(builder, "joystick"));
LMS = GTK_WIDGET(gtk_builder_get_object(builder, "sensor"));
sensorviz = GTK_WIDGET(gtk_builder_get_object(builder, "sensorviz"));
obstacles = GTK_WIDGET(gtk_builder_get_object(builder, "obstacles"));
avoidance = GTK_WIDGET(gtk_builder_get_object(builder, "avoidance"));

//connect all the callback functions
    g_signal_connect (window, "destroy", G_CALLBACK (on_window_destroy), NULL);
    g_signal_connect (segway, "clicked", G_CALLBACK (on_button1_clicked), NULL);
    g_signal_connect (seg_driver, "clicked", G_CALLBACK (on_button2_clicked), NULL);
    g_signal_connect (joystick, "clicked", G_CALLBACK (on_button3_clicked), NULL);
    g_signal_connect (LMS, "clicked", G_CALLBACK (on_button4_clicked), NULL);
    g_signal_connect (sensorviz, "clicked", G_CALLBACK (on_button5_clicked), NULL);
    g_signal_connect (obstacles, "clicked", G_CALLBACK (on_button6_clicked), NULL);
    g_signal_connect (avoidance, "clicked", G_CALLBACK (on_button7_clicked), NULL);

    g_object_unref(G_OBJECT(builder));

    //set the window title
    gtk_window_set_title((GtkWidget*)window,"Segway System");

    //show the window
    gtk_widget_show(window);

    //start roscore if it not already started
    system("roscore &");

    //infinite wait
    gtk_main();

    return 0;
ObstacleDetect_node.cpp

#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>
#include <math.h>
#include <vector>
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include <segwaytry/state.h>
#include "std_msgs/Empty.h"

#define SCREENWIDTH 800
#define SCREENHEIGHT 400
#define XRES 320 //number of cells in the x direction (horizontal)
#define YRES 160 //number of cells in the y direction (vertical)
#define RATE 500 //rate given in milliseconds
#define MAXRANGE 20 //maximum range of sensor
#define MAXOBSTACLES 10 //max number of obstacles to detect at a time
#define THRESHOLD .001 //probability threshold for detecting an obstacle
#define ERROR 5 //if the clusters are within this length, finish
#define MAXITERATIONS 40 //max number of clustering iterations

#define PI 3.14159265

using namespace std;

//struct to hold the current velocity estimate
struct velocity_estimate
{
    int x; // measured in cells
    int y;
};

//The class that provides the functionality of finding a moving object and detecting its velocity
class ObstacleReader
{
    public:
        ObstacleReader();
        ~ObstacleReader();
        void init();
};
void display();
void predictionFunction();
void movementFunction();
void Grouping(int,int,int);
double GaussianRand(double,double);
void cluster();
double distance(int,int,int,int);

double scan[360];
double occDist[XRES][YRES]; //holds the probability of an obstacle being in that cell
double newOccDist[XRES][YRES]; //holds the new probability distribution

bool validRegion[XRES][YRES];

double probDist[XRES][YRES][VELRES][VELRES]; //holds the current velocity estimate

double numSuccessors[XRES][YRES];
double numAntecedents[XRES][YRES];
double newVel[XRES][YRES][VELRES][VELRES];

int grouping[XRES][YRES];

int numGroups; //holds the number of groups to check for

double groupVelocity[MAXOBSTACLES][VELRES][VELRES];

bool beginning; //special case of the first estimates

//clustering data
int difference[XRES][YRES];
int oldCluster[XRES][YRES];
int newCluster[XRES][YRES];

velocity_estimate oldEstimate[MAXOBSTACLES];
velocity_estimate newEstimate[MAXOBSTACLES];

int centroidx[MAXOBSTACLES];
int centroidy[MAXOBSTACLES];
int clusterSize[MAXOBSTACLES];
int clusterCount;
int newClusterCount;
double new_x;
double new_y;

double new_yaw;

ros::Publisher pub;

};
ObstacleReader::ObstacleReader()
{
    init();
}
ObstacleReader::~ObstacleReader()
{
}

void ObstacleReader::init()
{
    srand(time(NULL));
    beginning =true;
    //initialize all the variables
    double init = 1.0 / (VELRES*VELRES*1.0);

    new_x=0;
    new_y=0;
    new_yaw=0;

    for(int i=0;i<XRES;i++)
    {
        for(int j=0;j<YRES;j++)
        {
            occDist[i][j]=1/40.0; // completely arbitrary
            newOccDist[i][j]=0;
            validRegion[i][j]=false;
            oldCluster[i][j]=-1;
            for(int k=0;k<VELRES;k++)
            {
                for(int l=0;l<VELRES;l++)
                {
                    probDist[i][j][k][l]=init;
                }
            }
        }
    }
}
for(int i=0;i<MAXOBSTACLES;i++)
{
    for(int k=0;k<VELRES;k++)
    {
        for(int l=0;l<VELRES;l++)
        {
            groupVelocity[i][k][l]=init;
        }
    }
    oldEstimate[i].x=0;
    oldEstimate[i].y=0;
    newEstimate[i].x=0;
    newEstimate[i].y=0;
}
}
void ObstacleReader::display()
{

double value;
    double max;
    int x,y;
    int middle=(VELRES-1)/2;
    //how to draw the obstacles
    for(int i=0;i<XRES;i++)
    {
        for(int j=0;j<YRES;j++)
        {
            value=occDist[i][j];
            if(occDist[i][j]>THRESHOLD)
            {
                glBegin(GL_QUADS);
                    //glColor3d(1.0-value,1.0-value,1.0-value);
                    //glColor3d(j%2,j%2,j%2);
                    glColor3d(0,0,0);
                    glVertex2d(i,j);
                    glVertex2d(i+1,j);
                    //printf("occ: %f\n",SCREENWIDTH*1.0/XRES);
                    glVertex2d(i+1,j+1);
                    glVertex2d(i,j+1);
                    glEnd();

                if(oldCluster[i][j]>0)
                {

glBegin(GL_QUADS);
    //glColor3d(1.0-value,1.0-value,1.0-value);
    //glColor3d(j%2,j%2,j%2);
    glColor3d(1.0,0.0,0.0);
    glVertex2d(i,j);
    glVertex2d(i+1,j);
    //printf("occ: %f\n",SCREENWIDTH*1.0/XRES);
    glVertex2d(i+1,j+1);
    glVertex2d(i,j+1);
    glEnd();

};
}
//drawing the estimated velocity
    glColor3d(0,1,0);
    for(int i=0;i<XRES;i++)
    {
        for(int j=0;j<YRES;j++)
        {
            max=0;
            x=0;
            y=0;
            if(oldCluster[i][j]>0)
            {
                glBegin(GL_LINES);
                    glVertex2d(i+.5,j+.5);
                    glVertex2d(i+.5+oldEstimate[oldCluster[i][j]-1].x,j+.5+oldEstimate[oldCluster[i][j]-1].y);
                glEnd();
            }
        }
    }
}

void ObstacleReader::cluster()
{
    //uses K-means clustering to determine where an obstacle is
    int count=0;

    bool start=true;
    int tempCentroidx[MAXOBSTACLES];
    int tempCentroidy[MAXOBSTACLES];
int numCells[MAXOBSTACLES];
int sumx[MAXOBSTACLES];
int sumy[MAXOBSTACLES];

int resx=XRES/6;
int resy=YRES/5;
for(int i=0;i<MAXOBSTACLES;i++)
{
    if(MAXOBSTACLES<=(XRES/4))
    {
        centroidx[i]=i*2+XRES/2-MAXOBSTACLES;
        centroidy[i]=0;
    }
    else
    {
        centroidx[i]=i%XRES;
        centroidy[i]=i/XRES;
    }
    numCells[i]=0;
    sumx[i]=0;
    sumy[i]=0;
}

int min;
int temp;
int num;
int difference;
do
{
    for(int i=0;i<XRES;i++)
    {
        for(int j=0;j<YRES;j++)
        {
            min=10000;
            if(newCluster[i][j]>0)
            {
                if(!start)
                {
                    for(int k=0;k<MAXOBSTACLES;k++)
                    {
                        temp=distance(centroidx[k],centroidy[k],i,j);
                        if(temp<min)
                        {
                            min=temp;
                        }```
if(temp<min)
{
    min=temp;
    num=k+1;
}
}
newCluster[i][j]=num;
else
{
    num=newCluster[i][j];
}
numCells[num-1]++;
sumx[num-1]+=i;
sumy[num-1]+=j;

for(int i=0;i<MAXOBSTACLES;i++)
{
    if(numCells[i]!=0)
    {
        tempCentroidx[i]=sumx[i]/numCells[i];
        tempCentroidy[i]=sumy[i]/numCells[i];

        difference+=distance(centroidx[i],centroidy[i],tempCentroidx[i],tempCentroidy[i]);
    }
    centroidx[i]=tempCentroidx[i];
    centroidy[i]=tempCentroidy[i];
}
difference/=MAXOBSTACLES;
count++;
start=false;
}
while(difference>ERROR && count<MAXITERATIONS);

for(int i=0;i<MAXOBSTACLES;i++)
{
    clusterSize[i]=numCells[i];
}
return;

double ObstacleReader::distance(int x1,int y1,int x2,int y2)
void ObstacleReader::predictionFunction()
{
    int sensorIndex;
    double radians, degrees, x, y, distance, sensorDistance;
    int middle = (VELRES - 1) / 2;
    int X, Y;
    double sum, sum2;
    double tempVel[VELRES][VELRES];

double resolution = 1.0 * MAXRANGE/YRES;
int indexX=0;
int indexY=0;
sum=0;
sum2=0;

for(degrees = 0; degrees <= 180 ; degrees+=.5)
{
    sensorIndex = (int) (degrees*2);
    for(double i=resolution;i<=MAXRANGE-1; i+=resolution)
    {
        x = i * cos(degrees * PI / 180.0);
        y = i * sin(degrees * PI / 180.0);
        indexX = (int)(x/(resolution)) + (XRES/2);
        indexY = (int)(y/(resolution));
        if(indexX>=0 && indexX<XRES && indexY>=0 && indexY<YRES)
        {
            sensorDistance = scan[sensorIndex];
            if(sensorDistance+resolution<i)
            {
                //assume nothing about what's behind the obstacle
                newOccDist[indexX][indexY] = occDist[indexX][indexY];
                validRegion[indexX][indexY] = false;
            }
            else if(sensorDistance - resolution > i)
            {
                //this means we should be in the free space
                //assign 1 and normalize later
                newOccDist[indexX][indexY]=1;
validRegion[indexX][indexY] = true;
}

else
{
  //this means we should be in obstacle space
  //assign a gaussian number here
  newOccDist[indexX][indexY] = GaussianRand(1000,fabs((sensorDistance -
  i)) * 10);

  validRegion[indexX][indexY] = true;
}
}
}
}

sum=0;
sum2=0;
newOccDist[XRES/2][0]=0;
newOccDist[XRES-1][0]=0;
newOccDist[XRES/2+1][0]=0;
for(int i=0;i<XRES;i++)
{
  for(int j=0;j<YRES;j++)
  {
    if(validRegion[i][j])
      sum+=newOccDist[i][j];
    else
      sum2+=newOccDist[i][j];
  }
}
for(int i=0;i<XRES;i++)
{
  for(int j=0;j<YRES;j++)
  {
    if(validRegion[i][j])
      newOccDist[i][j]=sum;
    else
      newOccDist[i][j]=sum2;
  }
}
cluster();
int numCheckedCells = (int) (1.0/(resolution)); // The number of cells to check around for 2 meters.

velocity_estimate calculated[MAXOBSTACLES];

int count[MAXOBSTACLES];

for(int i=0;i<MAXOBSTACLES;i++)
{
    newEstimate[i].x=0;
    newEstimate[i].y=0;

    calculated[i].x=0;
    calculated[i].y=0;
    count[i]=0;
}

int counter=0;
for(int i=0;i<XRES;i++)
{
    for(int j=0;j<YRES;j++)
    {
        if(newCluster[i][j]>0)
        {
            if(clusterSize[newCluster[i][j]]>=((1.0/resolution)/2))
            {
                for(int innerx=-1*numCheckedCells;innerx<numCheckedCells;innerx++)
                {
                    for(int innery=-1*numCheckedCells;innery<numCheckedCells;innery++)
                    {
                        X=i+innerx;
                        Y=j+innery;
                        if(X >= 0 && X < XRES && Y >= 0 && Y < YRES)
                        {
                            if(oldCluster[X][Y]>0)
                            {
                                calculated[newCluster[i][j]-1].x=(-1*innerx);
                                calculated[newCluster[i][j]-1].y=(-1*innery);
                                count[newCluster[i][j]-1]++;
                            }
                        }
                    }
                }
            }
        }
    }
}

22
newEstimate[newCluster[i][j]-1].x+=oldEstimate[oldCluster[X][Y]-1].x;
newEstimate[newCluster[i][j]-1].y+=oldEstimate[oldCluster[X][Y]-1].y;
}
}
}
}
}
}
}
}
}
}
}
for(int i=0;i<MAXOBSTACLES;i++)
{
    newEstimate[i].x = (calculated[i].x*.8 + oldEstimate[i].x*.2);//*.9+newEstimate[i].x*.1);
    newEstimate[i].y = (calculated[i].y*.8 + oldEstimate[i].y*.2);//*.9+newEstimate[i].y*.1);
}

// transfer the occupancy distribution
for(int i=0;i<MAXOBSTACLES;i++)
{
    for(int j=0;j<YRES;j++)
    {
        occDist[i][j]=newOccDist[i][j];
        oldCluster[i][j]=newCluster[i][j];
    }
}
for(int i=0;i<MAXOBSTACLES;i++)
{
    oldEstimate[i].x=newEstimate[i].x;
    oldEstimate[i].y=newEstimate[i].y;
}
if(beginning)
    beginning=false;
}
void ObstacleReader::movementFunction()
{
    double diff_x=new_x;
    double diff_y=new_y;
double diff_yaw = new_yaw;
  diff_yaw = -1;
if(fabs(diff_x) <= .05) // for stabilization
  {
    diff_x = 0;
  }
if(fabs(diff_y) <= .05)
  {
    diff_y = 0;
  }
if(fabs(diff_yaw) <= .017)
  {
    diff_yaw = 0;
  }

// translate the clusters and occupancy distribution

double changedDist[XRES][YRES];
int changedCluster[XRES][YRES];

int dx;
int dy;

for(int i = 0; i < XRES; i++)
  {
    for(int j = 0; j < YRES; j++)
      {
        changedDist[i][j] = 0;
        changedCluster[i][j] = 0;
      }
  }

for(int i = 0; i < XRES; i++)
  {
    for(int j = 0; j < YRES; j++)
      {
        dx = i - diff_y;
        dy = j - diff_x;
        if(dx >= 0 && dx < XRES && dy >= 0 && dy < YRES)
          {
            changedDist[dx][dy] = occDist[i][j];
            changedCluster[dx][dy] = oldCluster[i][j];
          }
      }
  }
else
{
    if(dx<0)
        dx+=XRES;
    else if(dx>XRES)
        dx-=XRES;

    if(dy<0)
        dy+=YRES;
    else if(dy>YRES)
        dy-=YRES;

    changedDist[dx][dy]=0.00001;
    changedCluster[dx][dy]=0;
}

//rotation correction
for(int i=0;i<XRES;i++)
{
    for(int j=0;j<YRES;j++)
    {
        dx=(int)((i-XRES/2)*cos(diff_yaw)-j*sin(diff_yaw))+XRES/2;
        dy=(int)((i-XRES/2)*sin(diff_yaw)+j*cos(diff_yaw));

        if(dx>=0 && dx<XRES && dy>=0 && dy<YRES)
        {
            changedDist[dx][dy]=occDist[i][j];
            changedCluster[dx][dy]=oldCluster[i][j];
        }
        else if(dy<0)
        {
            dy*=-1;
            dx=XRES-dx;

            changedDist[dx][dy]=0.0001;
            changedCluster[dx][dy]=0;
        }
    }
}

for(int i=0;i<MAXOBSTACLES;i++)
{
    dx=oldEstimate[i].x;
dy=oldEstimate[i].y;
oldEstimate[i].x=cos(diff_yaw)*dx-sin(diff_yaw)*dy;
oldEstimate[i].y=sin(diff_yaw)*dx+cos(diff_yaw)*dy;

for(int i=0;i<XRES;i++)
{
    for(int j=0;j<YRES;j++)
    {
        occDist[i][j]=changedDist[i][j];
        oldCluster[i][j]=changedCluster[i][j];
    }
}
new_x=0;
new_y=0;
new_yaw=0;

double ObstacleReader::GaussianRand(double mean, double variance)
{
    //produces a random number from a Gaussian distribution
    double r1,r2,rSquared;
    double midway;
    do
    {
        r1 = (rand()%10000)/10000.0;
        r2 = (rand()%10000)/10000.0;
        rSquared = r1 * r1 + r2 * r2;
    } while(rSquared >= 1.0 || rSquared == 0.0);
    midway = sqrt(-2.0*log(rSquared)/rSquared);
    return (r1 * midway) * sqrt(variance) + mean;
}

void init()
{
    //initialize the opengl window
    glColor3f(1.0 , 1.0 , 1.0 , 0.0);
    glViewport(0,0,SCREENWIDTH,SCREENHEIGHT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0,XRES,0,YRES,-1,1);
void idleFunction()
{
    // when sitting, redraw
    // glutPostRedisplay();
}

ObstacleReader reader;

// function to call during every time step
void timerFunction(GLint value)
{
    reader.movementFunction();
    std_msgs::Empty empty;
    reader.pub.publish(empty);
    reader.predictionFunction();
    glutPostRedisplay();
    glutTimerFunc(value, timerFunction, value);
}

// segway callback to read odometry
void segwayCallback(const segwaytry::state msg)
{
    reader.new_x = msg.stateInfo[0];
    reader.new_y = msg.stateInfo[1];
    reader.new_yaw = msg.stateInfo[2];
}

void scanCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
    // read the sensor data from the message and store it for later
    for (int i = 0; i < 360; i++)
    {
        // put into the class version of this
        reader.scan[i] = msg->ranges[i + 90];
    }
}

void display()
{
    ros::spinOnce();
    glClear(GL_COLOR_BUFFER_BIT);
reader.display();

    glutSwapBuffers();
}
int main(int argc, char ** argv)
{
    //initialize ROS
    ros::init(argc,argv,"obstacle_detector");

    ros::NodeHandle n;
    ros::Subscriber sub;
    ros::Subscriber sub2;

    sub = n.subscribe("scan",1,scanCallback);
    sub2 = n.subscribe("segway_info",1,segwayCallback);

    reader.pub = n.advertise<std_msgs::Empty>("resetOdom",1);

    //initialize OpenGL
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(SCREENWIDTH,SCREENHEIGHT);
    glutInitWindowPosition(0,400);
    glutCreateWindow("Obstacles");

    init();

    glutDisplayFunc(display);
    glutTimerFunc(RATE,timerFunction,RATE);
    glutMainLoop();

    return 0;
}
SegwayJoy.cpp

#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include <joy/Joy.h>

class SegwayJoy
{
    public:
        SegwayJoy();

    private:
        void joyCallback(const joy::Joy::ConstPtr& joy);

        ros::NodeHandle nh_;
        int linear_,angular_;
        double l_scale_, a_scale_; 
        ros::Publisher vel_pub_; 
        ros::Subscriber joy_sub_; 
};

//this is called whenever we want to publish a command to the segway
void SegwayJoy::joyCallback(const joy::Joy::ConstPtr& joy)
{
    geometry_msgs::Twist vel;
    vel.angular.z = a_scale_ * joy->axes[angular_];
    vel.linear.x = l_scale_ * joy->axes[linear_];
    vel_pub_.publish(vel);
}

SegwayJoy::SegwayJoy():linear_(1),angular_(2)
{
    linear_ = 1;
    angular_ = 3;
    l_scale_ = -16;
    a_scale_ = 1;

    //subscribe to the joystick callback
    joy_sub_ = nh_.subscribe<joy::Joy>("joy",1,&SegwayJoy::joyCallback,this);
    //advertise to the twist message which the segway driver reads
    vel_pub_ = nh_.advertise<geometry_msgs::Twist>("geometry_msgs/Twist",1);
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "segwayjoy");
    SegwayJoy segway;

    ros::spin();
}

SensorViz.cpp

#include <stdlib.h>
#include </usr/include/GL/gl.h>
#include </usr/include/GL/glu.h>
#include </usr/include/GL/glut.h>
#include <math.h>
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"

#define SCREENWIDTH 800
#define SCREENHEIGHT 400
#define PI 3.14159265
#define DISPLAY_RANGE 20

using namespace std;

double scan[360];

void init()
{
    //initialize the opengl window
    glClearColor(0.0 , 0.0 , 0.0 , 0.0);
    glViewport(0,0,SCREENWIDTH,SCREENHEIGHT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1*SCREENWIDTH/2,SCREENWIDTH/2,0,SCREENHEIGHT,-1,1);

    //init the scan data
    for(int i=0;i<360;i++)
    {
        scan[i]=20;
    }
}

void showSensor()
{
    double factorx=(SCREENWIDTH/2)/DISPLAY_RANGE;
    double factory=(SCREENHEIGHT)/DISPLAY_RANGE;
    glBegin(GL_LINES);
    glColor3d(1.0,1.0,1.0);
    //draw the sensor data
    for(int i=0;i<360;i++)
    {
        //...
glVertex2d(0,0);
glVertex2d(cos(i*PI/360.0)*factorx*scan[i],sin(i*PI/360.0)*factory*scan[i]);
}
glEnd();

for(int i = 1; i <= 20; i++)
{
  glColor3d(1.0,0.0,0.0);
  glBegin(GL_LINE_STRIP);
  for(int s = 0; s <= 180; s+=2)
  {
    glVertex2d(cos(s*PI/180.0)*factorx*i,sin(s*PI/180.0)*factory*i);
  }
  glEnd();
}

void display()
{
  ros::spinOnce();
  glClear(GL_COLOR_BUFFER_BIT);
  showSensor();
  glutSwapBuffers();
}

void idleFunction()
{
  //when sitting, redraw
  glutPostRedisplay();
}

void timerFunction(GLint value)
{
  glutPostRedisplay();
  glutTimerFunc(value,timerFunction,value);
}

void scanCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  //read the sensor data from the message and store it for later
  for(int i=0;i<360;i++)
  {
    scan[i]=msg->ranges[i+90];
  }
}

int main(int argc, char ** argv)
{
// initialize ROS
ros::init(argc, argv, "sensor");

ros::NodeHandle n;
ros::Subscriber sub;

sub = n.subscribe("scan", 1, scanCallback);

// initialize OpenGL
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize(SCREENWIDTH, SCREENHEIGHT);

init();

init();

glutDisplayFunc(display);

return 0;
}
VFH.cpp

#include <list>
#include <vector>
#include <math.h>
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include <joy/Joy.h>
#include "sensor_msgs/LaserScan.h"

#define PI 3.1415926535897932384626433832795
#define MINIMUM DISTANCE 1
#define MAX_STEERING ANGLE 57.0

using namespace std;

class VFH
{
public:
    VFH();
    geometry_msgs::Twist getDesiredSteeringAngle(double*, double, double);
    double beamAngles[361];

private:
    int linear, angular;
    double linearScale, angularScale;
    double scanInfo[361];
    ros::NodeHandle nh;
    ros::Publisher segwayControls;
    ros::Subscriber joystick;
    ros::Subscriber laserSensor;
    void joyCallback(const joy::Joy::ConstPtr& joy);
    void scanCallback(const sensor_msgs::LaserScan::ConstPtr& msg);
};

VFH::VFH()
{
    // Sets up the beam angles between -90 and 90 at half degree increments
    // This is used to store the actual angle of the beam
    for(int i = -180; i <= 180; i++)
    {
        beamAngles[180 + i] = i/2.0;
        ROS_INFO("%f\n",beamAngles[180+i]);
    }
}
//Setup subscriber to read joystick commands
joystick = nh.subscribe<joy::Joy>("joy",10,&VFH::joyCallback,this);

//Setup subscriber to read laser sensor data
laserSensor = nh.subscribe("scan",1,&VFH::scanCallback, this);

//Setup advertisor to broadcast segway commands
segwayControls =
nh.advertise<geometry_msgs::Twist>("geometry_msgs/Twist",10);

//Sets parameters needed for interpreting joystick commands
linear = 1;
angular = 3;
linearScale = -4;
angularScale = 1;

//Init the array that holds the measurement data from the laser sensor
//Unites in meters with a max of 20
for(int i = 0; i <= 360; i++)
{
    scanInfo[i] = 20;
}

void VFH::scanCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
    //Reads the sensor data from the message and store it
    //This only reads in data between -90 and 90 degrees at half degree increments
    for(int i = 0; i <= 360; i++)
    {
        scanInfo[i] = msg->ranges[i+90];
    }
}
void VFH::joyCallback(const joy::Joy::ConstPtr& joy)
{
    //Generates a twist message which is the command message for the robot
    geometry_msgs::Twist vel;

    //If the system is set to move forward, enable the collision avoidance
    if(joy->axes[linear] > 0)
    {
        vel = getDesiredSteeringAngle(scanInfo, linearScale*joy->axes[linear],
                                        angularScale * joy->axes[angular]);
    }
else
{
    // Else just execute the command as there are no sensors facing backwards
    vel.linear.x = linearScale * joy->axes[linear];
    vel.angular.z = angularScale * joy->axes[angular];
}

// Broadcast the command
segwayControls.publish(vel);

geometry_msgs::Twist VFH::getDesiredSteeringAngle(double* sensorData, double xVel, double yaw)
{
    geometry_msgs::Twist returnVelocity;

    // Stores the beginning and the end of a range of values
    double beginning = -1, end;

    // Stores the desired orientation
    double desiredAngle;

    // Used to store the closest valley
    // A valley is an area of clear measurements, clear means that an
    // obstacle was not detected within a certain range
    double bestValleyDifference;
    double tempDifference;

    // Index of the best candidate valley
    int bestValley;

    // Holds all the valleys
    vector<vector<double> > myValleys;

    // Temp vector
    vector<double> myTemp;

    // Finds the beginning and ends of all valleys
    for(int i = 0; i <= 360; i++)
    {
        // If the sensor beam is clear for at least 11 units (1.1 meters (11)
        // this is the beginning of a valley
        if(sensorData[i] > MINIMUM_DISTANCE)
        {
            // Mark the beginning
            beginn
beginning = beamAngles[i];

//Continue checking measurements to find the end of the valley
for(int others = i; others <= 360; others++)
{
    //Found the first beam that has detected an obstacle that is
    //clear
    //than .5 meters
    if(sensorData[others] <= MINIMUM_DISTANCE)
    {
        //Marks the end of the valley
        end = beamAngles[others - 1];

        //Store the valley information in a temporary vector
        myTemp.push_back(beginning);
        myTemp.push_back(end);

        //Stores a copy of the vector into the list of valleys
        myValleys.push_back(myTemp);
        myTemp.clear();
    }
    else if(others == 360)
    {
        //Hit the end of the measurements
        //Mark the end
        end = beamAngles[others];

        //Stores the valley and exits the loop
        myTemp.push_back(beginning);
        myTemp.push_back(end);
        myValleys.push_back(myTemp);
        myTemp.clear();
        i = others + 1;
    }
}
}

//Safety check, make sure there is at least one valley
/If there is no valleys, do not allow the system to move forward, only able to turn in place
    if (beginning == -1)
    {
        returnVelocity.angular.z = yaw;
        returnVelocity.linear.x = 0;
        return returnVelocity; //Change this to a message with no forward velocity at all.
    }

    // Gets the direction in radians
    desiredAngle = yaw * MAX_STEERING_ANGLE;

    // Checks if the valley is the entire sensor field
    if (myValleys[0][0] == beamAngles[0] && myValleys[0][1] == beamAngles[360])
    {
        // The valley consists of the entire sensor field meaning that there is no immediate danger of running into an obstacle so do not adjust user controls at all
        returnVelocity.angular.z = yaw;
        returnVelocity.linear.x = xVel;
        return returnVelocity; // Change this to return a message
    }
    else
    {
        // Valley is not the whole field so the best valley must be chosen
        // Gets the difference in steering between the first valley and the desired angle
        bestValleyDifference = desiredAngle - (myValleys[0][0] + myValleys[0][1]) / 2.0;
        bestValley = 0;

        ROS_INFO("Desired Angle: %f\t%f + %f / 2 = %f\n", desiredAngle, myValleys[0].at(0), myValleys[0].at(1), bestValleyDifference);

        // Choose the best valley out of all of them, the best valley is one who's center is closest to the user's desired steering angle
        for (int i = 0; i < myValleys.size(); i++)
        {
            // Calculates the difference between the midpoint of the current valley and what the desired steering angle is
tempDifference = desiredAngle - (myValleys[i][0] + myValleys[i][1]) / 2.0;
ROS_INFO("Desired Angle: %f \pm \frac{%f}{2} = %fn", desiredAngle, myValleys[i].at(0), myValleys[i].at(1), tempDifference);

// Finds out if the new difference is smaller than the previous
if(fabs(tempDifference) < fabs(bestValleyDifference))
{
    // new one is better
    // Store the information
    bestValley = i;
    bestValleyDifference = tempDifference;
}

// Best valley has been found, return the result
returnVelocity.angular.z = ((myValleys[bestValley][0] + myValleys[bestValley][1]) / 2.0) / MAX_STEERING_ANGLE;
returnVelocity.linear.x = xVel;
ROS_INFO("212 YB: %f YA: %fn", yaw, returnVelocity.angular.z);
return returnVelocity;

int main(int argc, char** argv)
{
    // Init the ros backend for this process
    ros::init(argc, argv, "vfh");

    // Creates an instance of itself
    VFH obstacleAvoider;

    // ENGAGE
    ros::spin();
}