Implementation of Gray-Scale Imaging for Non-Uniform Coverage Control

Matthew Wen
University of California, San Diego
Winter 2015

Summary

This report documents my work in adding non-uniform coverage control capability using gray-scale imaging to represent a non-uniform importance function. Background research involving coverage control and gray-scale imaging was conducted. My efforts culminated in an addition to the simple_deployment.cpp code in which the code generates a gray scale image of a Gaussian distribution imposed upon the image generated by the overhead camera, which represents the area that the Turtlebots inhabit. Due to the structure of the deployment code, the Gaussian gray-scale image was able to be scaled to multiple robots, such that multiple robots can distribute themselves according to the same importance function simultaneously.

Big Picture

The purpose of the ROS Turtlebot project is to develop control algorithms to direct the behavior of robots within a multi-robot network. A core aspect of the ROS Turtlebot project is the focus on a multi-robot network. The use of more than one robot, each equipped with its own set of sensors and capabilities, would indicate a scaled increase in overall performance or capabilities as the number of robots, deployed for a single task, increase. However, the integration of the capabilities of each robot into one coordinated effort is a complex and difficult task. Current studies involve coordinated mapping of an area with the Kinect sensors on the Turtlebots using SLAM algorithms. Another study involves the deployment of Turtlebots into certain arrangements or configurations based upon communication between the Turtlebots.

My Contributions

During the Fall Quarter I conducted background research on coverage control via deployment. Based upon the existing deployment code developed by Rokus Ottervanger, I developed a methodology for using gray scale imaging to represent a non-uniform importance function that the Turtlebots must evenly distribute themselves across. During Winter Quarter, I implemented this methodology into the simple_deployment.cpp code and developed an algorithm that represents a Gaussian distribution using gray-scale imaging, which is representative of a non-uniform importance function. The algorithm can be scaled to multiple Turtlebots, such that multiple robots actively communicate with each other (through a master computer) to evenly space themselves based on the importance function.

I have also done some preliminary work this quarter to investigate methods to increase the speed of the deployment code using calculations in MATLAB. The specific area of the deployment code I have been investigating is how to increase the calculation speed of the centroids of Voronoi Cells by using different estimation techniques that sacrifice accuracy for speed. My investigation looked into the amount of accuracy lost versus the time saved every iteration of the deployment code. However, my investigation is very preliminary, and I have not determined a viable solution yet.

Preliminaries

Uniform coverage control occurs in four major steps. First, the Turtlebots determine their position in the area covered by the overhead camera using localization algorithms. Second, Voronoi diagrams are calculated which divide up the area into cells based on the perpendicular bisectors between neighboring Turtlebots. To put it simply, each Voronoi cell generated contains a Turtlebot's position and all the points in the area closest to the Turtlebot that occupies that cell. Third, the centroid of each Voronoi cell is calculated, and the centroid of the Voronoi cell is set as the goal position that the Turtlebot is commanded to go to. Fourth, the Turtlebot moves toward the goal point until the deployment code iterates again, going through all four steps repeatedly. The repeated iteration of the deployment code continues, resulting in the Turtlebots eventually reaching an equilibrium state, where they are evenly distributed across the area.

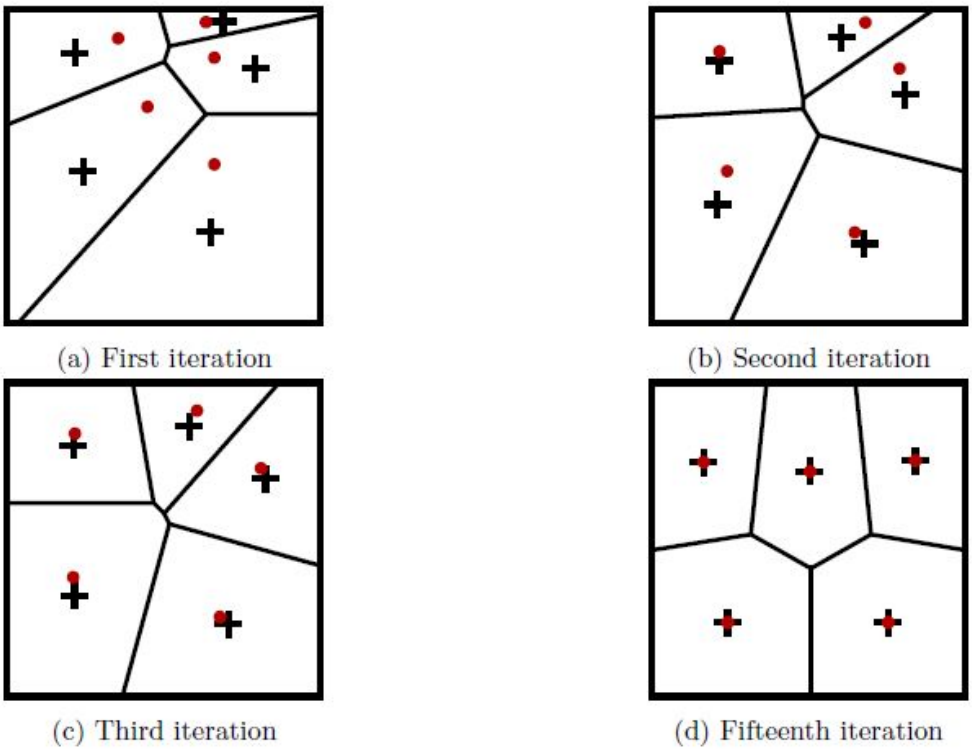

Figure 1 Successive iterations of deployment involving Voronoi cell diagrams. [1]

Methodology

The implementation of non-uniform coverage control required alternation of the third step in the deployment code, in which the centroid is calculated. The uniform coverage control code uses black and white pixels to calculate the Voronoi cells and split them up into separate images, which means the image of each Voronoi cells is represented as an array of zeros and ones. Ones represent pixels that are part of the Voronoi cell, and zeros represent pixels that are not, as demonstrated in Figure 2.

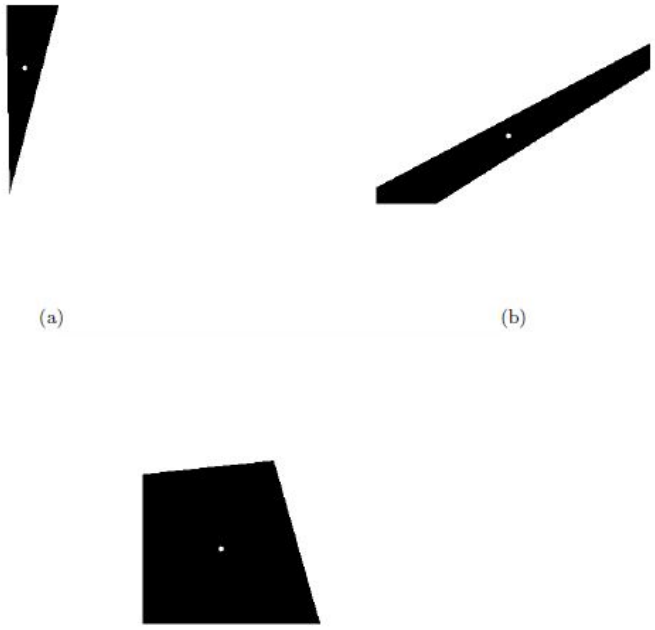

Figure 2 Voronoi cell diagrams images represented by black and white pixels. [2]

However, to implement a non-uniform importance function, gray scale imaging provides the capability to represent pixels as values ranging from 0 to 255. As a result, gradients can be established across the entire area the Turtlebots inhabit instead of just an area with uniform weight. A MATLAB code was developed to generate a Gaussian distribution importance function across the area of the Turtlebots, with the values of the Gaussian being within the range of 0 to 255 for a 640x480 image as shown in Figure 3.

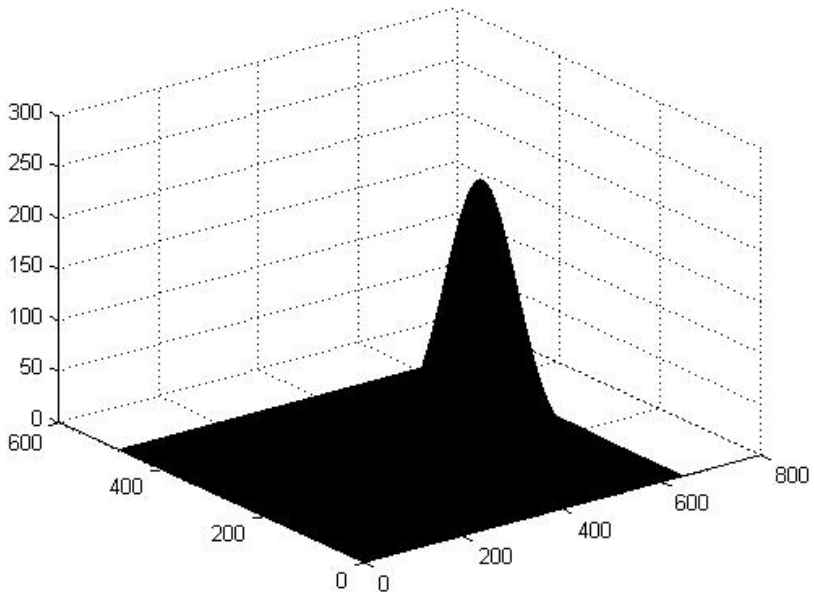

Figure 3 Gaussian importance function generated in MATLAB for a 640x480 size image.

With the gray-scale image of the Gaussian importance function defined, this gray scale image is then multiplied with every black and white Voronoi cell image, such that each pixel in the Voronoi cell is properly weighted with respect to its position in the defined Gaussian function. Consequently, when the centroid of each Voronoi cell is generated, the weight of the non-uniform importance function is taken into account.

Figure 4 shows the gray-scale image of the Gaussian importance function and the corresponding goal position of the Turtlebot. In the example depicted in Figure 4, there is only one Turtlebot present in the area, so the goal position is close to the peak of the Gaussian distribution. However, tests were conducted with multiple Turtlebots, and they performed as expected with each Voronoi cell centroid calculated based on the Gaussian importance function.

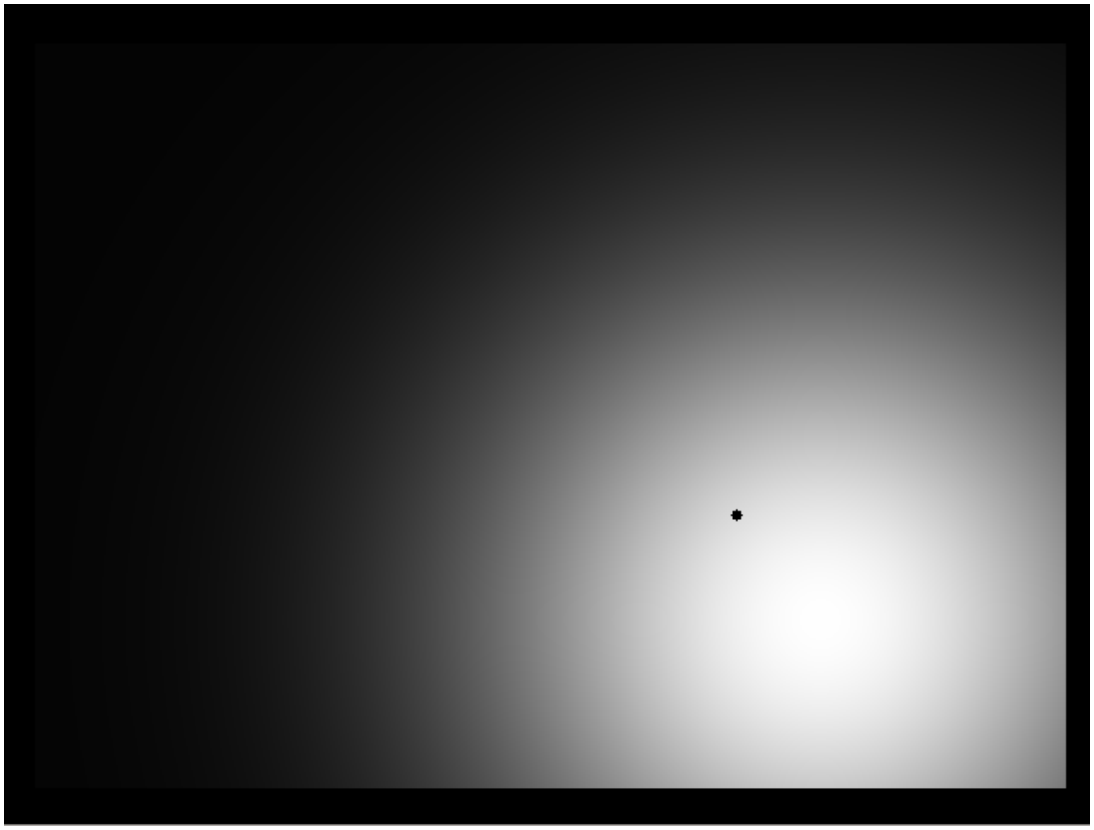

Figure 4 Voronoi cell of the gray-scale Gaussian importance function with goal position of a Turtlebot depicted as the black dot in the center-right of the image.

There are some notable features of the non-uniform importance function code that can be exploited for increased capabilities. The C++ code is shown below and there are a number of values that can be altered to change the Gaussian importance function. The location of the peak of the Gaussian as well as the x and y variance of the Gaussian can be modified. Since the non-uniform importance function is recalculated for every iteration of the deployment code, it allows for the importance function to be updated in real-time if a ROS node is configured to publish values such as the x and y location of the peak and the x and y variance. Dynamic, real-time update of the non-uniform importance function is not set up yet, but can be done in the near future, given the structure of the code.

```
//Multiply the voronoi cell image with a non-uniform importance function

	cv::Mat impfunc = cv::Mat::zeros(480, 640, CV_8UC1);
	cv::Mat newcelImg = cv::Mat::zeros(480, 640, CV_8UC1);
	float peakx = 0.75; //choose peakx to be a value between 0 and 1
	float peaky = 0.75; //choose peaky to be a value between 0 and 1
	float peakxdummy;
	float peakydummy;
	float ampl = 251;
	float sigmadist;
	float sigmax = 100;
	float sigmay = 100;
	int i;
	int j;

	for(i = 0; i < impfunc.rows; i++){

		for(j = 0; j < impfunc.cols; j++){
```

```
                impfunc.at<uchar>(i,j) = 253 - ampl * exp( -( pow (j - (640*peakx),2.0)
/ pow(2*sigmax, 2.0) + pow( i - (480*peaky) , 2.0 ) / pow( 2*sigmay , 2.0 ) ) );

              }

       }

       for(i = 0; i < impfunc.rows; i++){

              for(j = 0; j < impfunc.cols; j++){

                newcelImg.at<uchar>(i,j) = impfunc.at<uchar>(i,j) *
celImg.at<uchar>(i,j);

              }

       }

       celImg = newcelImg;

        // Compute the center of mass of the Voronoi cell
        cv::Moments m = moments(celImg, false);
        cv::Point centroid(m.m10/m.m00, m.m01/m.m00);
```

One important note about the algorithm is that the Gaussian distribution is defined from 2 to 253 and not the full range of 0 to 255. The reason for this is because there are rounding errors as the code converts float values into integers between 0 and 255. As a result, when the range is defined from 0 to 255, entire regions of the Gaussian function are given the incorrect weighted value as shown in Figure 5. In addition, it is important to note that the Gaussian distribution calculation must be subtracted from the maximum value (in the case of the current code, the maximum value is 253) thereby inverting the image, so that the peak of the Gaussian is displayed in white, thereby meaning the peak corresponds to a higher weight.

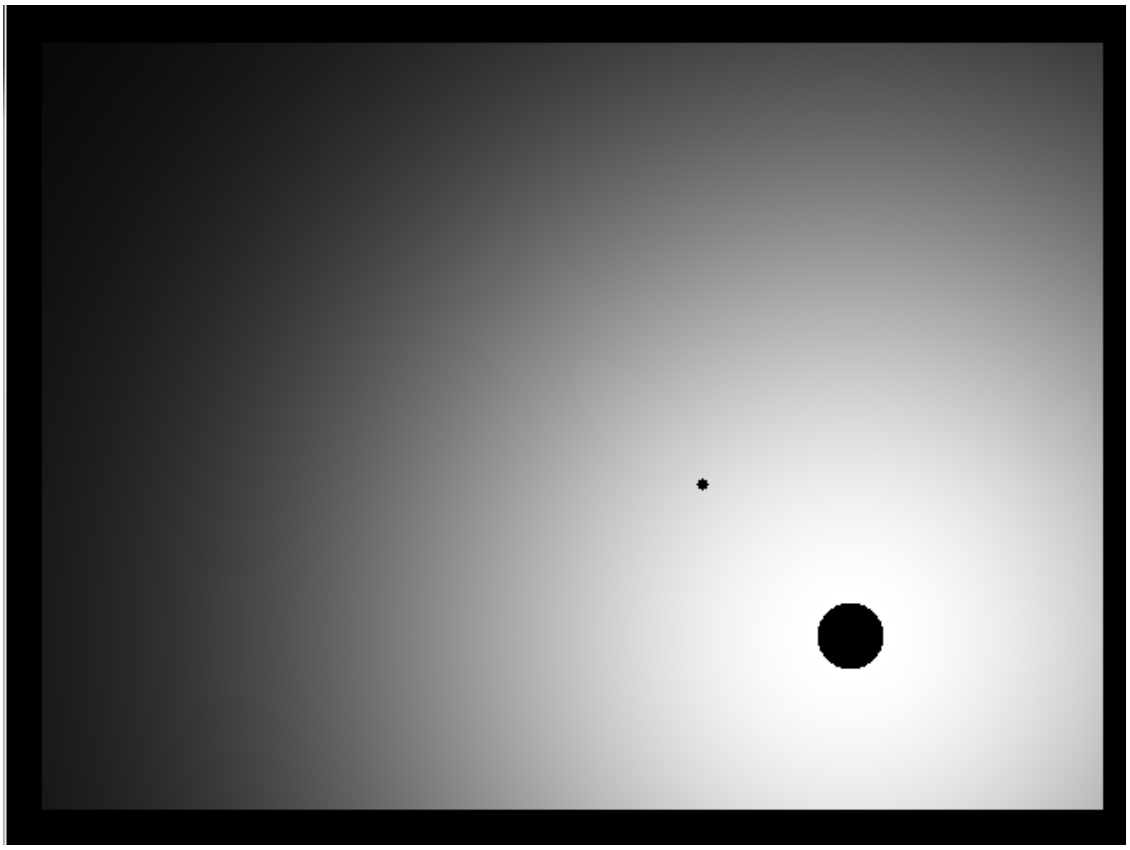

Figure 5 Voronoi cell of the gray-scale Gaussian importance function with goal position of a Turtlebot depicted as the black dot in the center-right of the image. The large black dot in the lower right is due to a rounding error with the gray-scale imaging is defined from 0 to 255.

Tips

There are a few troubleshooting tips I learned throughout Winter Quarter. The first one being that the deployment code depends on the camera reading and identifying the Aruco markers on the Turtlebots in order to run. As a result, it is important that the Aruco markers be tweaked and angled to ensure that the camera is reading the marker on each Turtlebot. Unfortunately, sometimes the camera does not read the Aruco markers very well once the Turtlebots reach certain positions or are at certain angles. This behavior may suggest the need to reprint the Aruco markers or re-calibrate the camera.

There were a few times where the code I ran one day would not run the next day under the same conditions. When this happens there are a few areas that I discovered were issues that I suggest to investigate if one encounters this situation. First, make sure that the .bashrc file on both the Turtlebots being used and the master computer are unaltered. An alteration in the .bashrc file, depending on the change, can inhibit certain nodes from publishing or subscribing when deployment is running. Another issue I discovered was there was an accidental modification of the GitHub branch I was working on. As a result, when a code is not working that should work, another tip I learned was to check the update history of the branch using GitHub to ensure that the branch was not modified in a way that could change the performance of the code.

The last major tip is that when a computer is switched to a new branch in GitHub, that branch must be compiled on the computer in use before running the code of that branch. Overall, the process for switching branches in GitHub is to: 1) search for and pull the desired branch from the GitHub repository 2) update the desired branch from the GitHub repository 3) re-compile the code of the desired branch. After all three steps have been completed the code from the branch should run as expected.

Conclusions and Future Work

This quarter I developed an algorithm to distribute the Turtlebots according to a Gaussian importance function using gray-scale imaging. The algorithm I developed can be scaled to multiple Turtlebots. There are a few areas of the code that need to be refined before placing it into the master code. I plan on implementing a variable in the code, such that when the variable is set to 'true' in the command line then the non-uniform importance function will be used to distribute the Turtlebots. In addition, I would like to further develop the dynamic, real-time updating of the importance function by creating a node that publishes the x and y coordinates of the Gaussian peak. The x and y coordinates would be controlled and altered using the arrow keys on the keyboard similar to the teleop node used to manually drive the Turtlebots.

Another area to be improved in deployment is the speed of the code. As stated in the beginning of the report, I have conducted some preliminary simulations of different methods to calculate the centroid of the Voronoi cells more quickly. I am also planning on implementing timers into the deployment code to determine areas that are the greatest cause for delays in each iteration of the code.

References

[1] Dominik Moritz. Lloyd's algorithm. http://en.wikipedia.org/wiki/Lloyd's algorithm, 2013. Accessed: September 30, 2014.

[2] Ottervanger, Rokus. "Implementation of a Distributed Algorithm for Coverage Control." 2014.

Referenced Code

Simple_deployment.cpp

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

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <geometry_msgs/PoseStamped.h>
#include <nav_msgs/OccupancyGrid.h>
//#include <move_base_msgs/MoveBaseAction.h>
//#include <actionlib/client/simple_action_client.h>
#include <tf/tf.h>

#include <turtlebot_deployment/PoseWithName.h>

#include "agent.h"
#include "VoronoiDiagramGenerator.h"

//typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;
const cv::Scalar WHITE = cv::Scalar(255);
```

```
const cv::Scalar GRAY = cv::Scalar(120);
const cv::Scalar BLACK = cv::Scalar(0);

/// Functor definitions
struct MatchString
{
    MatchString(const std::string& s) : s_(s) {}
    bool operator()(const Agent& obj) const
    {
        return obj.getName() == s_;
    }
private:
    const std::string& s_;
};

struct SortAgentsOnDistance
{
    bool operator()( const Agent& agent1, const Agent& agent2 ) const {
        return agent1.getDistance() < agent2.getDistance();
    }
};

/// Class definition
class SimpleDeployment
{
public:
    // Constructor and destructor
    SimpleDeployment();

private:

    // Methods
    void positionsCallback(const turtlebot_deployment::PoseWithName::ConstPtr& pose);
    void mapCallback(const nav_msgs::OccupancyGrid::ConstPtr& map);
    void publish();
    cv::Mat drawMap();
    double distance(geometry_msgs::Pose,geometry_msgs::Pose);
    void removeOldAgents();

    // ROS stuff
    ros::NodeHandle ph_, nh_;
    ros::Subscriber pos_sub_;
    ros::Subscriber map_sub_;
    ros::Publisher goal_pub_;
    ros::Timer timer_;
//    MoveBaseClient ac_;

    // Message variables
    nav_msgs::OccupancyGrid map_;
    bool got_map_;
    bool got_me_;
    geometry_msgs::PoseStamped goal_;

    // Other member variables
    double hold_time_;
    Agent this_agent_;
    double resolution_;
    double period_;
    bool show_cells_;

    std::vector<Agent> agent_catalog_;
};
```

```
/// Method bodies
SimpleDeployment::SimpleDeployment():
    ph_("~"),
    this_agent_(),
    got_map_(false),
    got_me_(false),
    hold_time_(0.0),
    resolution_(0.0),
    period_(2.0),
    show_cells_(false)
{
    ROS_DEBUG("SimpleDeployment: Constructing SimpleDeployment object");
    std::string temp_name;
    ph_.param("robot_name", temp_name, this_agent_.getName() );     // Name of this
agent
    ph_.param("hold_time", hold_time_, hold_time_ );                // Time to hold
peer agents in agent catalog
    ph_.param("resolution", resolution_, resolution_ );             // Resolution of
the voronoi image (in m/px)
    ph_.param("period", period_, period_);                          // Period of
execution for the publish method
    ph_.param("show_cells_", show_cells_, show_cells_);                // Determines
whether agent will attempt to port visualization
    this_agent_.setName(temp_name);

    if (this_agent_.getName() == "not defined") {
        ROS_ERROR("SimpleDeployment: Robot name not set");
    }
    else if (hold_time_ <= 0.0) {
        ROS_ERROR("SimpleDeployment: Hold time invalid or not set");
    }
    else if (resolution_ <= 0.0) {
        ROS_ERROR("SimpleDeployment: Resolution invalid or not set");
    }
    else if (period_ <= 0.0) {
        ROS_ERROR("SimpleDeployment: Period not valid");
    }
    else {

        pos_sub_ = nh_.subscribe<turtlebot_deployment::PoseWithName>("/all_positions",
1, &SimpleDeployment::positionsCallback, this);
        map_sub_ = nh_.subscribe<nav_msgs::OccupancyGrid>("/map",1,
&SimpleDeployment::mapCallback, this);
        goal_pub_ = nh_.advertise<geometry_msgs::PoseStamped>("move_base_simple/goal",
1, true);

        timer_ = nh_.createTimer(ros::Duration(period_),
boost::bind(&SimpleDeployment::publish, this));
       if (show_cells_) {
              cv::namedWindow( "Voronoi Cells", cv::WINDOW_AUTOSIZE );

              // Display importance function

              cv::namedWindow( "Importance Function", cv::WINDOW_AUTOSIZE );

       }
    }
    ROS_DEBUG("SimpleDeployment: SimpleDeployment object constructed");
}

void SimpleDeployment::positionsCallback(const
turtlebot_deployment::PoseWithName::ConstPtr& posePtr)
{
```

```
    ROS_DEBUG("SimpleDeployment: Positions received, finding robot in database");
    // Search for agent in the catalog using functor that compares the name to an
input string
    std::vector<Agent>::iterator it = std::find_if(agent_catalog_.begin(),
agent_catalog_.end(), MatchString(posePtr->name) );

    // If the agent is already in the catalog, update the position and recalculate the
distance.
    if ( it != agent_catalog_.end() ) {
        ROS_DEBUG("SimpleDeployment: Robot found, updating pose and distance");

        it->setPose(posePtr->pose);
        it->setDistance(distance(this_agent_.getPose(),posePtr->pose));
    }

    // else (the agent is not yet in the catalog), create an Agent object and push it
into the catalog vector
    else {
        ROS_DEBUG("SimpleDeployment: Robot not found in database");

        if ( posePtr->name != this_agent_.getName() ) {
            ROS_DEBUG("SimpleDeployment: Robot is not me. Adding it to database");

            // This initializes an object called "agent" with id = 1, the pose of the
incoming message, and the distance from this agent to the agent that published the
message
            Agent agent( 1, *posePtr, distance(this_agent_.getPose(), posePtr->pose)
);
            agent_catalog_.push_back( agent );
        }
        else {
            ROS_ERROR("SimpleDeployment: Robot is me! Updating position and adding to
database");
            this_agent_.setPose(posePtr->pose);
            got_me_ = true;

            // This initializes an object called "agent" with id = 0, the pose of the
incoming message, and a distance of 0.0;
            Agent agent( 0, *posePtr, 0.0 );
            agent_catalog_.push_back( agent );
        }
    }

    // Sort agent list on distance (using functor)
    std::sort( agent_catalog_.begin(), agent_catalog_.end(), SortAgentsOnDistance() );
    ROS_DEBUG("SimpleDeployment: Positions processed");
}

// Function to retreive and store map information.
void SimpleDeployment::mapCallback(const nav_msgs::OccupancyGrid::ConstPtr& mapPtr)
{
    map_ = *mapPtr;
    got_map_ = true;
}

// In this function, the Voronoi cell is calculated, integrated and the new goal point
is calculated and published.
void SimpleDeployment::publish()
{
    if ( got_map_ && got_me_ ) {
        double factor = map_.info.resolution / resolution_; // zoom factor for openCV
visualization
```

```
        ROS_DEBUG("SimpleDeployment: Map received, determining Voronoi cells and
publishing goal");

        removeOldAgents();

        // Create variables for x-values, y-values and the maximum and minimum of
these, needed for VoronoiDiagramGenerator
        float xvalues[agent_catalog_.size()];
        float yvalues[agent_catalog_.size()];
        double xmin = 0.0, xmax = 0.0, ymin = 0.0, ymax = 0.0;
        cv::Point seedPt = cv::Point(1,1);

        // Create empty image with the size of the map to draw points and voronoi
diagram in
        cv::Mat vorImg =
cv::Mat::zeros(map_.info.height*factor,map_.info.width*factor,CV_8UC1);

        for ( uint i = 0; i < agent_catalog_.size(); i++ ) {
            geometry_msgs::Pose pose = agent_catalog_[i].getPose();

            // Keep track of x and y values
            xvalues[i] = pose.position.x;
            yvalues[i] = pose.position.y;

            // Keep track of min and max x
            if ( pose.position.x < xmin ) {
                xmin = pose.position.x;
            } else if ( pose.position.x > xmax ) {
                xmax = pose.position.x;
            }

            // Keep track of min and max y
            if ( pose.position.y < ymin ) {
                ymin = pose.position.y;
            } else if ( pose.position.y > ymax ) {
                ymax = pose.position.y;
            }

            // Store point as seed point if it represents this agent
            if ( agent_catalog_[i].getName() == this_agent_.getName() ){
                // Scale positions in metric system column and row numbers in image
                int c = ( pose.position.x - map_.info.origin.position.x ) * factor /
map_.info.resolution;
                int r = map_.info.height * factor - ( pose.position.y -
map_.info.origin.position.y ) * factor / map_.info.resolution;
                cv::Point pt =  cv::Point(c,r);

                seedPt = pt;
            }
            // Draw point on image
//            cv::circle( vorImg, pt, 3, WHITE, -1, 8);
        }

        ROS_DEBUG("SimpleDeployment: creating a VDG object and generating Voronoi
diagram");
        // Construct a VoronoiDiagramGenerator (VoronoiDiagramGenerator.h)
        VoronoiDiagramGenerator VDG;

        xmin = map_.info.origin.position.x; xmax = map_.info.width  *
map_.info.resolution + map_.info.origin.position.x;
        ymin = map_.info.origin.position.y; ymax = map_.info.height *
map_.info.resolution + map_.info.origin.position.y;
```

```
        // Generate the Voronoi diagram using the collected x and y values, the number
of points, and the min and max x and y values

VDG.generateVoronoi(xvalues,yvalues,agent_catalog_.size(),float(xmin),float(xmax),floa
t(ymin),float(ymax));

        float x1,y1,x2,y2;

        ROS_DEBUG("SimpleDeployment: collecting line segments from the VDG object");
        // Collect the generated line segments from the VDG object
        while(VDG.getNext(x1,y1,x2,y2))
        {
            // Scale the line segment end-point coordinates to column and row numbers
in image
            int c1 = ( x1 - map_.info.origin.position.x ) * factor /
map_.info.resolution;
            int r1 = vorImg.rows - ( y1 - map_.info.origin.position.y ) * factor /
map_.info.resolution;
            int c2 = ( x2 - map_.info.origin.position.x ) * factor /
map_.info.resolution;
            int r2 = vorImg.rows - ( y2 - map_.info.origin.position.y ) * factor /
map_.info.resolution;

            // Draw line segment
            cv::Point pt1 = cv::Point(c1,r1),
                      pt2 = cv::Point(c2,r2);
            cv::line(vorImg,pt1,pt2,WHITE);
        }

        ROS_DEBUG("SimpleDeployment: drawing map occupancygrid and resizing to voronoi
image size");
        // Create cv image from map data and resize it to the same size as voronoi
image
        cv::Mat mapImg = drawMap();
        cv::Mat viewImg(vorImg.size(),CV_8UC1);
        cv::resize(mapImg, viewImg, vorImg.size(), 0.0, 0.0, cv::INTER_NEAREST );

        // Add images together to make the total image
        cv::Mat totalImg(vorImg.size(),CV_8UC1);
        cv::bitwise_or(viewImg,vorImg,totalImg);

        cv::Mat celImg = cv::Mat::zeros(vorImg.rows+2, vorImg.cols+2, CV_8UC1);
        cv::Scalar newVal = cv::Scalar(1), upDiff = cv::Scalar(100), loDiff =
cv::Scalar(256);
        cv::Rect rect;

        cv::floodFill(totalImg,celImg,seedPt,newVal,&rect,loDiff,upDiff,4 + (255 << 8)
+ cv::FLOODFILL_MASK_ONLY);

//Multiply the voronoi cell image with a non-uniform importance function

       cv::Mat impfunc = cv::Mat::zeros(480, 640, CV_8UC1);
       cv::Mat newcelImg = cv::Mat::zeros(480, 640, CV_8UC1);
       float peakx = 0.75; //choose peakx to be a value between 0 and 1
       float peaky = 0.75; //choose peaky to be a value between 0 and 1
       float peakxdummy;
       float peakydummy;
       float ampl = 251;
       float sigmadist;
       float sigmax = 100;
       float sigmay = 100;
       int i;
       int j;
```

```
	/*if (peakx < 0.5){
		peakxdummy = 1.0-peakx;
	}

	if (peakx > 0.5){
		peakxdummy = peakx;
	}

	if (peaky < 0.5){
		peakydummy = 1.0-peaky;
	}

	if (peaky > 0.5){
		peakydummy = peaky;
	}

	sigmadist = sqrt ( pow( 480 * peakxdummy, 2.0 ) + pow( 640 * peakydummy, 2.0 )
);

	sigmax = sigmadist/4.0;
	sigmay = sigmadist/4.0;*/

	for(i = 0; i < impfunc.rows; i++){

		for(j = 0; j < impfunc.cols; j++){

		  impfunc.at<uchar>(i,j) = 253 - ampl * exp( -( pow (j - (640*peakx),2.0)
/ pow(2*sigmax, 2.0) + pow( i - (480*peaky) , 2.0 ) / pow( 2*sigmay , 2.0 ) ) );

		/*if(impfunc.at<uchar>(i,j)>254){

		impfunc.at<uchar>(i,j) = 254;

		}

		if(impfunc.at<uchar>(i,j)<1){

		impfunc.at<uchar>(i,j) = 1;

		}*/

		}

	}

	//bitwise_not ( src, dst );

	//cv::subtract(cv::Scalar:all(255),impfunc,impfunc);

	for(i = 0; i < impfunc.rows; i++){

		for(j = 0; j < impfunc.cols; j++){

		  newcelImg.at<uchar>(i,j) = impfunc.at<uchar>(i,j) *
celImg.at<uchar>(i,j);

		}

	}

	//newcelImg = impfunc.mul(celImg);
```

```
       celImg = newcelImg;

        // Compute the center of mass of the Voronoi cell
        cv::Moments m = moments(celImg, false);
        cv::Point centroid(m.m10/m.m00, m.m01/m.m00);

        cv::circle( celImg, centroid, 3, BLACK, -1, 8);

        // Convert seed point to celImg coordinate system (totalImg(x,y) =
celImg(x+1,y+1)
        cv::Point onePt(1,1);
        centroid = centroid - onePt;

        for ( uint i = 0; i < agent_catalog_.size(); i++ ){

            int c = ( xvalues[i] - map_.info.origin.position.x ) * factor /
map_.info.resolution;
            int r = map_.info.height * factor - ( yvalues[i] -
map_.info.origin.position.y ) * factor / map_.info.resolution;
            cv::Point pt =  cv::Point(c,r);
            cv::circle( totalImg, pt, 3, GRAY, -1, 8);
        }

        cv::circle( totalImg, seedPt, 3, WHITE, -1, 8);
        cv::circle( totalImg, centroid, 2, WHITE, -1, 8);                //where
centroid is the goal position

       // Due to bandwidth issues, only display this image if requested
       if (show_cells_) {
              cv::imshow( "Voronoi Cells", totalImg );

             // Display Importance Function

             cv::imshow( "Importance Function", newcelImg );

       }
        cv::waitKey(3);

        // Scale goal position in map back to true goal position
        geometry_msgs::Pose goalPose;
//        goalPose.position.x = centroid.x * map_.info.resolution / factor +
map_.info.origin.position.x;
        goalPose.position.x = centroid.x / factor + map_.info.origin.position.x;
//        goalPose.position.y = (map_.info.height - centroid.y / factor) *
map_.info.resolution + map_.info.origin.position.y;
        goalPose.position.y = (map_.info.height - centroid.y / factor) +
map_.info.origin.position.y;
        double phi = atan2( seedPt.y - centroid.y, centroid.x - seedPt.x );
        goalPose.orientation = tf::createQuaternionMsgFromYaw(phi);

        goal_.pose = goalPose;
        goal_.header.frame_id = "map";
        goal_.header.stamp = ros::Time::now();

        goal_pub_.publish(goal_);

//        move_base_msgs::MoveBaseGoal goal;

//        goal.target_pose.header.frame_id = "map";
//        goal.target_pose.header.stamp = ros::Time::now();

//        goal.target_pose.pose = goalPose;
```

```
//        ac_.sendGoal(goal);
    }
    else {
        ROS_DEBUG("SimpleDeployment: No map received");
    }

}


/// Function definitions

cv::Mat SimpleDeployment::drawMap()
{
    ROS_DEBUG("Drawing map");
    cv::Mat src = cv::Mat::zeros(map_.info.height,map_.info.width,CV_8UC1);

    // Copy map values to image
    for ( int i = 0; i < map_.info.height; i++ ) {
        for ( uint j = 0; j < map_.info.width; j++ ) {
            signed char* pxPtr = &map_.data[i*map_.info.width + j];
            if ( *pxPtr == -1 ) {
                src.at<uchar>(i,j) = 0;
            }
            else {
                src.at<uchar>(i,j) = round(map_.data[i*map_.info.width + j]*2.55);
            }
        }
    }
    return src;
}

double SimpleDeployment::distance(geometry_msgs::Pose pose2, geometry_msgs::Pose
pose1)
{
    double dx = pose2.position.x - pose1.position.x;
    dx = dx*dx;

    double dy = pose2.position.y - pose1.position.y;
    dy = dy*dy;

    double dz = pose2.position.z - pose1.position.z;
    dz = dz*dz;

    // Using squared Euclidean distance
    return /*sqrt(dx+dy+dz);*/ dx+dy+dz;
}

void SimpleDeployment::removeOldAgents(){
    for( std::vector<Agent>::iterator it = agent_catalog_.begin(); it !=
agent_catalog_.end(); ++it) {
        if ( it->getAge().toSec() > hold_time_ ) {
            agent_catalog_.erase( it );
            it--;
       /*[leonardo/deployment-9]
processROS_MASTER_URI=http://calipso.dynamic.ucsd.edu:11311/

 has died [pid 3427, exit code -6, cmd
/home/turtlebot/git_catkin_ws/devel/lib/turtlebot_deployment/simple_deployment
__name:=deployment __log:=/home/turtlebot/.ros/log/6c4f8f40-b182-11e4-89f6-
386077cd78ee/leonardo-deployment-9.log].
log file: /home/turtlebot/.ros/log/6c4f8f40-b182-11e4-89f6-386077cd78ee/leonardo-
deployment-9*.log*/
```

```
        }
        else {
        }
    }
}

/// Main
int main(int argc, char** argv)
{
    ros::init(argc, argv, "simple_deployment");
    SimpleDeployment simple_deployment;

    ros::spin();
}
```