May 2006

# Probabilistic Robot Localization using Visual Landmarks

Peter E. Anderson-Sprecher
*Macalester College*, psprecher@gmail.com

# Probabilistic Robot Localization Using Visual Landmarks

**Peter Anderson-Sprecher**

Senior Honors Thesis

Macalester College
St. Paul, Minnesota
Department of Computer Science

Advisor — Prof. S. Fox
Second Reader — Prof. G. Schneider

May 1, 2006

## Abstract

Effective robot navigation and route planning is impossible unless the position of the robot within its environment is known. Motion sensors that track the relative movement of a robot are inherently unreliable, so it is necessary to use cues from the external environment to periodically localize the robot. There are many methods for accomplishing this, most of which either probabilistically estimate the robot's movement based on range sensors, or require having enough unique visual landmarks present to geometrically calculate the robot's position at any time.

In this project I examined the feasibility of using the probabilistic Monte Carlo localization algorithm to estimate a robot's location based off of occasional visual landmark cues. Using visual landmarks has several advantages over using range sensor data in that landmark readings are less affected by unexpected objects and can be used for fast global localization.

To test this system I designed a robot capable of navigating Olin-Rice by observing pieces of colored paper placed at regular intervals along the halls as an extension of my summer 2005 research on RUPART. The localization system could not localize the robot in many situations due to the sparse nature of the landmarks, but results suggest that with minor modifications the system could become a reliable localization scheme.

# Contents

# 1 Introduction

A primary task that one faces in any attempt to build a working robot is localization, the act of determining a robot's location within its environment. If a robot's location within some sort of map can be known at all times then performing such things as route navigation and goal planning are quite simple. Localization involves comparing data from an external sensor with information about the robot's environment in order to determine the robot's most probable location.

It is not sufficient to simply estimate the robot's position through "dead reckoning," or using motion sensors to measure movement from a known start position. Due to the inherently unreliability of motion sensors, if the robot is run for a long distance without performing any kind of localization it will be impossible to have any meaningful data regarding the robot's position. The extent of motion sensor error is very dependent on the type of robot used, but a small wheeled robot such as the Pioneer 2 robot used in this study will have its assumed position be off by as much as 5 meters within a minute of running. The task of compensating for these motion sensor errors during the robot's movement is called "Tracking" and is an essential capability of any localization system.

Furthermore, it is often necessary to determine where a robot is in an environment without any knowledge of the robot's starting position. This task, called "Global Localization," is often described using a kidnapping metaphor: you pick up the robot, blindfold it, and put it anywhere in the environment and it must be able to determine its location. This is in general much more difficult than simply maintaining the robot's location. (Russel 908)

In this study I examine the feasibility of using distributed visual landmarks to accomplish localization in a probabilistic fashion. Robots in large building-type environments usually are designed to rely on range sensor (i.e. sonar or laser) data for localization, simply because vision sensors usually require extensive placement of landmarks prior to running the robot. However, there are also several distinct advantages of using visual landmarks instead.

## 2 Sensor input

There are two decisions that must be made when developing a localization system. First, one must determine the nature of the sensor data that will be used to compare against some known environment as well as the map or other method for storing environmental data. Second, one must choose a method for extrapolating from observed sensor data the most likely location of the robot at a given time.

The two most often-used types of sensor data are visual, usually from some sort of camera, and range, usually from sonar or laser sensors. There are, however, many other potential sources for localization sensing. One notable example is the Global Positioning System, a localization system commonly used in automobiles and aircraft, which relies on timing signals from satellites in geostationary orbit.

## 2.1 Range sensors

Range sensor data are probably the most widely commonly used sensors for localization. Using data from range sensors has several advantages. First, it requires virtually no processing before being used for localization, since range sensors give an absolute distance to the nearest obstacle which can be easily compared with a 2-dimensional map. Second, robots are almost universally equipped with either sonar or laser range sensors, so a localization scheme using range sensors can be easily adapted to work on just about any robot.

However, there are also disadvantages to using range sensors. Range sensor based localization can be thrown off by unexpected and/or moving objects, as well as objects that are too small to be taken into account in a map but show up on range sensors. A classic example of this involves a robot entering a ballroom full of people and trying to localize based on sonar sensors emanating from the robot at leg level. In this case, the presence of people will make the room look more like a small room or closet than an open ballroom because most sonar readings will reflect off of people's legs instead of walls. If localization is attempted in this situation the robot will believe that it is in another room entirely.

Using range sensor-based localization also makes global localization quite difficult. A given set of sensor data may not be unique to one location within the environment; many locations may yield similar sensor readings. For example, if a robot is traversing the hallways of a standard office building, it would be impossible to quickly determine which hallway the robot is in. The robot would have to traverse several hallways and junctions before gathering enough data to determine its position.

## 2.2 Vision sensors

Vision sensors, usually taking the form of a camera mounted on the robot, are an alternative to range sensors that can help solve some of these issues. Direct camera snapshots of the environment are not useful, however, since the camera only gives a bitmap of what is seen at any given time, and there is no easy way to map this bitmap onto knowledge about the surrounding environment. To circumvent this problem, one must apply some sort of preprocessing to the bitmap in order to extract useful data from the camera before data obtained from vision sensors can be applied to localization.

### 2.2.1 Landmark-based localization

The easiest way to localize based on visual data is to find certain easily identifiable objects in the environment and pay attention only when one or more of those are in the camera's field of view. When any of these landmarks are seen by the robot, the perceived locations of the landmarks can be compared to their known locations within the environment and the robot's position can be deduced.

Landmarks used for localization can be either artificial or natural. An artificial landmark is one that would not exist normally in the environment but has been placed to aid localization. Natural landmarks, on the other hand, are objects that are coincidental to the environment but still stand out enough to be easily recognizable. (Murphy 326) For example, a robot wandering through an office building might want to use a houseplant and a poster as natural landmarks, since they are unique and stand out from the background of the building.

Identifying blobs of uniform color is a function supported by most vision processing software, so an easy way to use artificial landmarks is to deliberately place markers of different solid colors around the environment in locations that are easily seen by the robot. Patterned artificial landmarks are also often used because it is easier to make them uniquely identifiable than it is with solid-color landmarks and the processing required to identify them can be relatively minor.

While artificial landmarks have the advantages of being easily identifiable and placed in useful locations, such placement can be troublesome. For comparable accuracy the frequency of placement must be the same regardless of the size of the environment. This means that large environments can be particularly difficult. For example, a field for robotic soccer is quite easy to mark so that several markers are visible at all times, but this would be very difficult in an open building environment such as what I attempted to have RUPART navigate. Fields for robotic soccer typically have over

5 landmarks in a field that is less than 10 square meters in size, so using a comparable marker density in Olin hall (about 400 $m^2$) would require several hundred markers. In section 4, I explore the idea of using artificial landmark-based localization with much more sparsely placed markers, but in general this problem is very difficult. Because of this difficulty, artificial landmarks are mostly used in small, controlled environments such as the fields used in robotic soccer competitions. Localization techniques for robotic soccer are described in more depth in section 3.1.

One way around these problems associated with placing artificial landmarks in a large environment is to try to use only natural landmarks instead. Several studies focus on the feasibility of identifying and tracking natural landmarks for use in robot localization, notably the work done by Isard (1999) on using the Conditional Density Propagation algorithm for natural landmark identification.

However, while attractive in concept natural landmarks have several distinct disadvantages: first, they require more computational power to identify than simple artificial landmarks, and second, it is hard to guarantee that landmarks are globally unique (landmarks may be very similar to each other.) For example, in an office environment there might be a brightly-colored poster on the wall in one room and the same poster in an adjacent room. Unless the robot knows ahead of time that there are identical landmarks in each room, it could easily mistake its position.

Whether natural or artificial, visual landmarks solve many of the problems inherent to range and other similar sensors. While range sensors are vulnerable to the effect of unexpected objects or room "clutter," visual sensors using landmarks are not affected by these kinds of extraneous objects since for successful localization they require only that the landmarks themselves be visible. Also, visual landmarks are often unique, making global localization very easy when compared to systems using sonar sensors.

### 2.2.2 Vision-based localization without landmarks

Another way to use vision sensor data in localization is to avoid searching for landmarks entirely and instead use the entire image as a reference for localization. One way to accomplish this is to construct a Computer Aided Design (CAD) model of the robot's environment and compare images directly to that. This, however, is impractical both because it is impossible to accurately model a real-world environment due to constant fluctuations and object complexities and because it would be computationally expensive to extract position data from comparing 2-D observations to such a model. It is better to reduce the complexity of the data found in the composite 2-D image maps and search for emerging trends that will be observed independent of a particular location for the robot. (Krose 1) One standard method for accomplishing this is that of Principal Component Analysis (PCA). The idea behind PCA is that the vast majority of what the robot sees will not be useful for localization. For example, a robot wandering down an office hallway might see large blocks of unchanging color along the walls and the floor, but it will also see regions of change near specific features on the wall and along the boundary between the wall and the floor. If we can somehow extract only those parts that are interesting, such as these regions of contrast, it will be easier to compare observations and localize the robot.

PCA is a statistical technique that is used to reduce the dimensionality of a dataset while preserving most of the information contained within it. In its pure form, it operates on a high-dimensional dataset in the following manner:

1. Subtract the mean of each of the data dimensions.
2. Calculate the covariance matrix of the data.
3. Order the eigenvectors by their eigenvalues. The eigenvector with the highest eigenvalue is the principal component of the data set, and the kth component corresponds to the vector with the kth highest eigenvalue.

(Smith 18)

Once the principal components have been identified, most of the less important components can be discarded without losing much information. When applied to robot vision, k sample images of d pixels each are collected initially, and then the images are represented as k d-dimensional data vectors. Then, PCA can be applied to the resulting d-dimensional data set. The principal components will also take the form of d-dimensional data vectors, which can also be interpreted as images similar to those collected in real time. While for significant (80%) image reconstruction as many as 90 eigenvectors are needed, robot localization can be accomplished with around 15 eigenvectors. (Krose 9,15)

Before these observations can be used for localization, a number of training samples must be collected from known locations within the environment for comparison with the robot's observations during localization. Because of this, PCA is most useful for robot localization when the environment is small enough to maintain a high density of training samples without making comparison tasks computationally expensive. Office environments several hundred square meters in size can be easily supported, but if the environment becomes much larger then maintaining a sufficient density of training samples would be difficult.

## 3 Localization systems

Regardless of the method used to collect and process the environment data, there are many methods for extracting position data from whatever is learned about the environment. One standard classification for these systems is that of iconic versus feature-based, where the former consists of algorithms that use sensor data with no pre-processing (i.e. raw sonar readings) while the latter performs some kind of feature extraction (i.e. finding landmarks.) (Murphy 377) Here we will use a slightly different classification: instantaneous localization algorithms, which use the robot's current location and sensor data collected at one, or perhaps two, instants in time, versus continuous localization algorithms, which use an aggregate of observations over time.

## 3.1 Instantaneous localization

The most straightforward instantaneous localization method is pure geometric localization, which involves taking the sensor data and geometrically calculating a best estimate of the robot's location based on what is known about the environment.

Regardless of the exact method used, there are several advantages to instantaneous localization:

1. There is no difference between global localization and tracking. The robot's position is calculated based solely on what is seen at one or more locations and not dependent on the robot being near an assumed position, so there is no penalty for not knowing the robot's position at the beginning.

2. Instantaneous localization is typically computationally inexpensive. No information is kept from previous observations, so only a few calculations are needed to obtain the robot's location. Because the robot's position must be triangulated based on its sensor data, instantaneous localization is generally used with sensor data that can be traced to identifiable sources with known locations. Range sensors, for exam-

ple, are a very poor choice since any set of observed distances could correspond to a great number of different locations, while visual landmarks are a very good choice since they can be uniquely identified and correspond to exact points in space.

One very successful instantaneous localization system is the Global Positioning System (GPS), which uses timing signals from 24 satellites in orbit around the globe as sensor input. These satellites each contain an atomic clock that constantly broadcasts the current time, which allows a receiver to determine the distance of each satellite by measuring the observed difference in the propagation delay of the signals. The satellites are positioned such that at least 4 are visible at any time, allowing for localization anywhere in the world at any time.

Instantaneous localization can also be used with visual landmarks. It is possible to triangulate the position of the robot after at least 2 measurements of a single marker (assuming the distance traveled between observations is known,) or after a single observation of at least 3 markers. However, the former is less commonly used because parallax computations are susceptible to minor error in either motion sensors or vision sensors. In section 5 I develop and examine a possible alternative to geometric localization for situations in which only one marker is visible at a time.

One situation in which geometric localization based on multiple markers is used often is in the burgeoning field of robotic soccer competition. A robotic soccer match usually consists of a number of robots in a small rectangular field that attempt to get the ball into their team's goal. The robots are entirely autonomous during the event except for communication with the other robots on their team, so naturally it is imperative that a very robust, fast, and reliable localization scheme be available. (Beetz 1)

In this environment, the task of ensuring that many markers are visible at all times is made easy by the fact that the field is relatively small and can be prepared ahead of time. The field is marked with a minimum of one distinctly colored post at each corner of the field, but usually more posts along each side are used together with markers placed on distant walls which give the robots an idea of absolute direction. Goals and the areas near goals are also marked with special colors so that important goal-oriented tasks can be easily accomplished. (Beetz 2)

Once the environment is prepared in this way, the robot can often deduce its location from the perceived angles between known marker locations with a straightforward geometric calculation. The robot will be able to know its position with no prior information provided that the robot is able to see at least three posts at any one time or two posts plus a marker that gives the robot an idea of its absolute orientation.

However, even with an extensively marked environment it is difficult to build a system capable of robust instantaneous localization in the manner described above, both because readings from vision sensors can be somewhat unreliable and because it is hard to ensure that the robot will be able to see the requisite number of landmarks at all times. Because of this, many systems for localization in robotic soccer such as the one implemented by Beetz et al. are not pure instantaneous localization systems but instead use a combination of multiple landmark locations, assumed position knowledge and odometry data in the context of a probabilistic localization system as described in section 3.2. Purely instantaneous localization systems are sometimes used when an abnormally wide range of view is available for the robot, such as in the work done by Marques and Lima using robots with omnidirectional cameras.

## 3.2 Continuous localization

There are also alternatives to using an instantaneous approach for localization. Many algorithms do not use only what is seen by the robot at one or two points in time, but instead use some sort of record of past observations in order to maintain a running record of the robot's most likely location.

Many of these methods are probabilistic, which is to say that they do not compute the actual location of the robot at every step with certainty, but instead keep track of the probability that the robot is in any particular position and update that continuously. When the robot is relatively certain of its location, the probabilities will be very high at a particular point and low everywhere else. (Russel 908) There are many probabilistic localization methods, such as dynamic Bayesian networks, Kalman filters, and Monte Carlo Localization. The localization system explained in chapter 5 is based off of the Monte Carlo Localization algorithm, so we will examine that particular algorithm in greater detail here.

### 3.2.1 Monte Carlo Localization

Probably the most common and easily implemented probabilistic localization method is Monte Carlo Localization (MCL). MCL can draw on any kind of sensor data: some kind of range sensor such as laser or sonar is most common, but in section 4 we will look at how it can be used in conjunction with environment data derived from visual cues.

MCL keeps track of a probability density function for the robot's current location based on a set of possible locations and how well its stored map data match up with current sensor input. This probability density function is then updated based on continuous observations of the robot's environment.

It would be difficult to keep track of a continuous probability density function over the entire map area, so a collection of representative points are kept instead of the entire function. The density of these sample points should correspond to the probability at any one point, i.e. the samples will cluster at the location of greatest probability. An example of a typical sample point distribuition can be seen in Figure 2.

The task of the algorithm is then to maintain and update these samples in the following manner:

1. Update the samples in accordance with some known robot motion model $P(X1|X0, A0)$ where $X$ is the robot's position and A is the robot's movement since its previous location.

2. Give a weight $W_i$ to each sample proportional to the probability that the sample is correct. This is accomplished by using a sensor noise model $P(Z|Z')$ to compare the reading with expected readings at the sample point.

3. Generate a new set of sample points based on these weights. New sample points $S'$ are taken from existing samples $S$ where each sample $S'_x$ is set to a sample $S_y$ with probability $\frac{W_y}{\sum W_i}$.

This algorithm is typically used with range sensor data, because with range sensors current information can always be compared with a 2-D map for localization. This is in contrast to visual landmarks, which may only be occasionally visible to the robot. Having range sensors be able to compare data with a 2-D map also has the added benefit during tracking of ensuring that the robot's expected position always stays within corridors and does not deviate across walls. Of course, there are several disadvantages to using range sensor data for Monte Carlo localization: the system will still have all of the weaknesses of range sensors such as vulnerability to unexpected objects and difficulty accomplishing global localization.

A standard algorithm for Monte Carlo Localization using range sensors is shown in Figure 1.

Figure 2 is a good example of what the distribution of the samples in MCL looks like when there is little sensor data to go on and therefore all samples are given roughly equal weight. The samples are spread out along the

```
function Monte-Carlo-Localization(a,z,N,model,map)
returns a set of samples
inputs:
a, the previous robot motion command
z, a range scan with M readings z1...zM
N, the mumber of samples to be maintained
Model, a probabilistic environment model with pose prior P(X0),
motion model P(X1|X0,A0), and range sensor noise model P(Z|Zhat)
Map, a 2D map of the environment
   for i = 1 to N do
      S[i] = sample from P(X1|X0 = S[i], A0 = a)
      W[i] = 1
      For j = 1 to M do
         Zhat = Exact-Range(j,S[i],map)
         W[i] = W[i] * P(Z = zj | Zhat = zhat)
         S = Weighted-Sample-With-Replacement(N,S,W)
      return S
```

Figure 1: MCL algorithm using range sensors (Russel 908)

direction perpendicular to the robot's recent movement because the majority of position error comes from error in the robot's orientation, which naturally causes perpendicular position error after moving a long distance. When useful sensor data is available to the robot, the samples will usually converge to a small area as seen in Figure 3.

# 4 RUPART

I developed several workable localization schemes in conjunction with my summer 2005 research on RUPART. The primary focus of the research was to develop a robot capable of navigating Olin hall at Macalester College by using case-based reasoning to select immediate wandering behaviors as well as long-term routes for the robot. However, since these "routes" consisted of a number of waypoints expressed in absolute coordinates that the robot had to reach on its way to the goal, an effective localization scheme was necessary for any kind of long-term navigation. (Fox 1) All research on RUPART was done using ActivMedia Pioneer 2 robots, and testing was done within the Olin science hall at Macalester College.

## 4.1 A first attempt at localization

The original localization scheme I developed for RUPART was an instantaneous localization system that used standard 8.5x11 pieces of brightly colored paper carefully placed at regular intervals throughout the environment.

The robot ordinarily ran entirely on dead reckoning, and occasionally paused at a marker to update its location based on the observation of a marker. In an ideal run the robot conducted localization only when a goal point with a nearby marker was reached in the course of normal navigation. Once such a goal was reached, the localization routine consisted of facing the marker and moving to within 75cm of the wall, at which time the robot turned perpendicular to the wall using sonar and then calculated its position based on a combination of its distance from the wall and the angle between the robot and the marker.

However, this system depended heavily on markers being put up frequently and with each marker paired with a goal location that signaled when the robot must stop to localize. Furthermore, using sonar to orient the robot with respect to the wall was not necessarily reliable, so unless markers were put up at virtually every map point the robot would almost certainly lose track of its next goal at some point due to accrued dead reckoning errors.

In this eventuality, the robot is forced to localize to the next marker that it sees. It is assumed that the robot's location is not in error enough to be closer to a different marker than the one it is actually next to, so the marker with the closest absolute location is assumed to be what the robot is seeing, and localization is conducted in the same way as before. (Fox 4)

The most significant disadvantage to this kind of localization scheme is that the robot is forced to stop for a significant amount of time whenever a marker is seen. This can make any kind of navigational task very slow, which prompted me to attempt a more elegant localization system as described in the next section.

## 4.2 A probabilistic localization scheme for RUPART using artificial landmarks

While stopping movement to establish a known position at every landmark can be a successful localization scheme, a more attractive solution would be to make a system that was able to extract localization information from landmarks without altering the physical movement of the robot in any way. Even without modifying the robot's movement it will still naturally see and recognize one of the landmarks fairly often, but such an observation yields only an angle measurement between the robot's orientation and the direction of the currently observed landmark.

One way to transfer this information into a localization system is to use Monte Carlo Localization with the observed angle of the marker as an alternative to the standard range sensor data described in section 3.2. This has several immediate advantages over the previous system used for RUPART: the robot will be able to localize without first needing to reach a goal point near the marker, and it will no longer be necessary to interrupt robot movement for the sake of localization.

It might also seem preferable to stop using visual landmarks altogether and adopt a standard range sensor-based MCL system. However, there are several distinct advantages of using visual landmarks that make it worth investigating:

1. It is not necessary to build a detailed map of the environment in order to use visual landmarks: all that is necessary is to place occasional landmarks and record the absolute location of each of them.

2. There are none of the unreliability issues seen with range sensor data when unexpected objects are introduced to the environment. As long as the landmarks are visible, localization will be unhindered.

3. Since artificial landmarks can be made to be uniquely identifiable, global localization is much easier than it is when using range sensor data.

However, in order accommodate landmark data, the standard MCL algorithm must be changed significantly. As before, given an initial set of samples (either set to the robot's initial position or distributed throughout the environment) we keep track of the movement of the robot by adjusting every sample according to the robot motion model $R(p)$, which in this case is based off of estimates of the dead reckoning error of the robot. The robot has a position that it updates continuously with regards to wheel motion sensors; we can assume that this is correct to within some error margin $E$ so that the perceived motion from the dead reckoning sensors is $\Delta X' = dX + E$. For the Pioneer robots, the average is about plus or minus 2%.

Based on this, we build our robot motion model for translating a sample $S = (S_x, S_y, S_\theta)$ to a new sample $S' = (S_x + \Delta S_x, S_y + \Delta S_y, S_\theta + \Delta S_\theta)$, given previous assumed position $R = (R_x, R_y, R_\theta)$ and new assumed position $R' = (R_x + \Delta X, R_y + \Delta Y, R_\theta + \Delta\theta)$ as follows:

We first let

$$\Delta Parallel = \Delta X cos(R_\theta) + \Delta Y sin(R_\theta)$$

$$\Delta Perp = \Delta Y cos(R_\theta) - \Delta X sin(R_\theta)$$

Then,

$$\Delta S_x = ((\Delta Parallel * cos(\theta)) - (\Delta Perp * sin(\theta))) + E_x$$

$$\Delta S_y = ((\Delta Perp * cos(\theta)) + (\Delta Parallel * sin(\theta))) + E_y$$

$$\Delta S_\theta = \Delta\theta + E_\theta$$

However, since E for a particular sample is necessarily unknown we instead apply a random error value from -7% to +7% to the sample. This will yield an average deviation of 3.5%, which is about right given a real-world average error of 2% since it is best to slightly overestimate error values. An average error rate of 3.5% should allow for higher than average position error without causing the samples to disperse too rapidly. Given this, we substitute adding $E$ with multiplying the perceived $\Delta S$ by a random value between .93 and 1.07. Then we have

$$\Delta S_x = ((\Delta Parallel * cos(\theta)) - (\Delta Perp * sin(\theta))) * (1 + ((random[0,1] * .02) - .01))$$

$\Delta S_y = ((\Delta Perp * cos(\theta)) + (\Delta Parallel * sin(\theta))) * (1 + ((random[0,1] * .02) - .01))$

$\Delta S_\theta = \Delta\theta * (1 + ((random[0,1] * .02) - .01))$

If there are no visual environment data at a particular step, then all that is necessary is to update the samples in accordance with the robot motion model. However, this alone will not localize the robot, so if some sort of visual cue is present then we must adjust the samples to give preference to the ones that fit the sensor data more closely. In our system, all that we know is the absolute position of the marker that is currently being observed within the environment (markers are assumed to be uniquely identifiable from their colors) and what the observed angle is between the marker and the forward orientation of the robot. We then need to assign weights to the samples based on how well they match this limited information.

In certain cases it is easy to tell that the robot has no chance of being in the position described by the sample: this is always the case when the robot is positioned behind the marker, as all markers are assumed to be flat pieces of paper and therefore not visible except from the 180 degrees in front of them. We also assume that markers are not visible from more than 5 meters away (generous for 8.5"x11" pieces of paper,) so samples with a Euclidean distance of more than 5 meters are similarly given a zero weight.

Otherwise, we assume that there is some possibility that the robot is seeing it from the position designated by the sample, and then compare the observed angle to the marker with the expected angle between the position sample and the marker location. A reasonable method for assigning a weight $W_S$ to a sample $S$ is to use $\frac{1}{|th-th'|}$, where th is the expected angle between the robot and the marker (calculated from the relative positions of the sample and the marker,) and $\theta'$ is the angle observed by the robot. This is not a direct estimation of $P(\theta|\theta')$, since for small values of $|\theta - \theta'|$ the weight will be much greater than one, but it should be proportional to the probability and does give a heavy emphasis to values that match up closely, which is a desirable trait.

However, there is a problem with this in that $W_S$ approaches infinity as $\theta$ approaches $\theta'$. This is easily fixed by letting the final weight for a sample be $\frac{1}{max(|\theta-\theta'|,\theta_0)}$, where $\theta_0$ is assumed to be the lowest possible meaningful measurement of theta for the marker. A value of $\theta_0 = .1$ should be practical for most applications.

Once the weights are established, the weights must be converted to probabilities for the purposes of creating the new sample set. We let $P_x = \frac{W_x}{\sum W_i}$, and then for each sample in the new sample set, set it to one of the old samples $S_x$ with probability $P_x$.

There is also one other contingency that must be accounted for. It is possible, either through having an initially unknown position or through sensor accidents, that the robot will lose track of its position entirely and all samples will be given a weight of zero on a particular pass. In this case the samples must somehow be reset because otherwise it is unlikely that meaningful localization will become possible again.

To solve this problem, whenever this happens we assign all samples to random positions within the range of positions that would be given a high weight for the currently observed marker. This means that all samples will be within the 180-degree arc in front of the currently observed marker and all orientations will be facing more or less in the direction of the marker. Unfortunately, we cannot eliminate positions that are impossible because of being behind or inside of a wall because the localization system does not have a map of the environment. The effects of this system for global localization are discussed further in section 6.2.

If the samples are updated in this fashion often enough, then the sample set should quickly converge upon the actual location of the robot. However, simply maintaining an accurate sample set is not sufficient for a completely functioning localization scheme, as most navigation programs run on the hardware position of the robot, rather than on the set of samples used for localization. Thus, it is necessary to periodically set the hardware position of the robot to whatever seems like the most probable position based on the current sample set. Furthermore, if the samples are too spread out or if none of them match current landmark data well, it would probably be better to avoid changing the hardware position at all and instead wait until the samples have had a chance to converge further.

An easy way to find the most probable location is to take the arithmetic mean of the samples in the x, y, and theta dimensions. Of course, there may be a number of outliers, so we also need to calculate the Euclidean distance of each sample from this mean point and then remove the 10% of the samples that are farthest from the mean point. Then, the mean is recalculated based on this reduced set.

An alternative method for finding the most probable location can be used if a marker is visible to the robot at the time it is being calculated. Samples nearest to the correct position will naturally be given high weights, so the mean position can be calculated based on only those positions that would be given a high weight if localization were to be conducted at that time.

If more than about 10% of the samples fall more than 1 unit (for simplicity, 1 radian is equal in magnitude to 1 meter) away from the calculated mean point, then the samples are probably too spread out to extract any useful data from and the robot should wait to reset its position. Furthermore, we need to make sure that the sample matches observed data well (after all, it is possible that the samples are clustered at an erroneous point.) This means that if the mean point is compared with an observed marker and the sensor data does not match what is expected (i.e. the mean point would be given a low weight) then it is not reliable enough to use for updating. Otherwise, if the samples are well clustered and the mean point appears to be accurate, then the mean point is probably a good estimation of the actual robot location and we can set the hardware position of the robot.

# 5 Method

All tests of the system were conducted in Olin hall at Macalester College using one of Macalester's ActivMedia Pioneer 2 robots. The robot's sensors consisted of 16 sonar sensors and one Sony PTZ camera. Markers for localization consisted of 8.5x11 inch pieces of colored paper that were placed about 6 inches above the floor at 7 locations throughout hallways in Olin hall.

RUPART was used as the primary behavior controller for the robot and is a behavior-based control system implemented in python with the Pyro system (Pyro) that uses case-based reasoning to select routes and behaviors for the robot. Behaviors dictate primary motion, avoidance, and immediate goal acquisition, while the route ensures that the next immediate goal will be both easily attained using reactive control and will bring the robot closer to its final destination. The localization task was treated as a behavior that was always active and that performed localization once per second for the duration of the run. The code for RUPART is in Appendix B. The MCL system was implemented in c++ using SWIG to generate a wrapper for use in the RUPART code, and can be seen in Apendix A.

Primary tests were conducted from the Macalester robotics lab to the west end of the south hallway in Olin hall. Three markers were passed in this case, two of which were approached by the robot head-on and one of which was only seen alongside the robot as it went down the first hallway.

All three markers used in this test were uniquely identifiable by their colors. In all cases 50 samples were maintained for localization.

The robot's assumed position, currently observed marker (if one was present,) and the position of all samples contained in the localization system were recorded once per second for the duration of each run. Results were analyzed by viewing them graphically using the display system listed in Appendix C.

# 6 Results

The system works well in that the samples usually cluster near the robot's actual position, especially just after a marker has been seen. Using this sort of data for probabilistic sampling has some inherent disadvantages, since the information that can be compared to the environment is limited (all we know is the angle between the robot and the marker.) This means that at any particular point there are many points that could satisfy the angle measurement, far more so than with range sensor based localization. The robot must be able to see the marker while traveling past it in a way that produces significant parallax in order to eliminate other possible locations near the marker.

Furthermore, the system behaves quite differently when the localization problem is tracking, where the robot's initial position is known, versus global localization, where the robot's initial position is unknown.

## 6.1 Tracking

For reliable tracking, it is necessary to either have a navigation system that is minimally dependent on localization, or to have a robot with low motion sensor error. In the robot that was used for testing this system, motion sensor error was relatively low, typically causing about 1 degree of error per 360 degree rotation. This enabled the robot to travel easily from one marker to the next without having a significant loss of position. In this kind of low-error environment, the best system for tracking is to rely on the motion sensors most of the time, and update only when samples are closely clustered together.

In the current system, tracking is possible as long as markers are seen frequently enough to allow the samples to converge on the correct location. After passing a single marker, the estimated position of the robot is usually accurate to within 50cm and the robot's estimated absolute orientation is accurate to within about 5 degrees. However, an error of only a few degrees can cause significant loss of position after a long distance is traversed.

Also, even when the robot leaves a marker with little or no position error, the samples will quickly disperse. Figure 2 shows the samples 20 seconds after seeing a marker. The X represents the robot's position and all other dots are the 50 samples currently maintained by MCL.

This means that if the robot is localized to a position near a single marker, then it cannot be guaranteed of still having accurate position data after running on dead reckoning for a moderate distance. Fortunately, we can

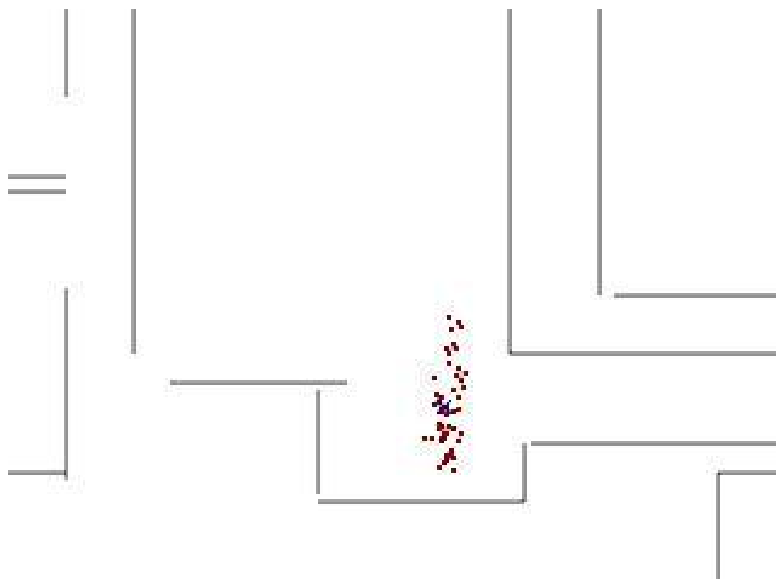

Figure 2: Distributed sample points surrounding robot

expect that every time a marker is passed, the samples will converge to a more precise grouping. A set of converged samples typical of what would be used for resetting the robot's position is seen in Figure 3.

It should be noted that while this kind of low motion sensor error rate is typical for wheeled robots on flat, even surfaces, the results would be very different for other systems in which legged robots were used or difficult terrain had to be traversed. This would create far more noticeable sensor error, and the same kind of sparsely marked environment would be very impractical to use. In addition to adjusting the motion model to account for the new environment, it would be necessary to significantly increase the frequency of markers. It is likely that this kind of sparsely populated marker-based localization would be impractical for such a system.

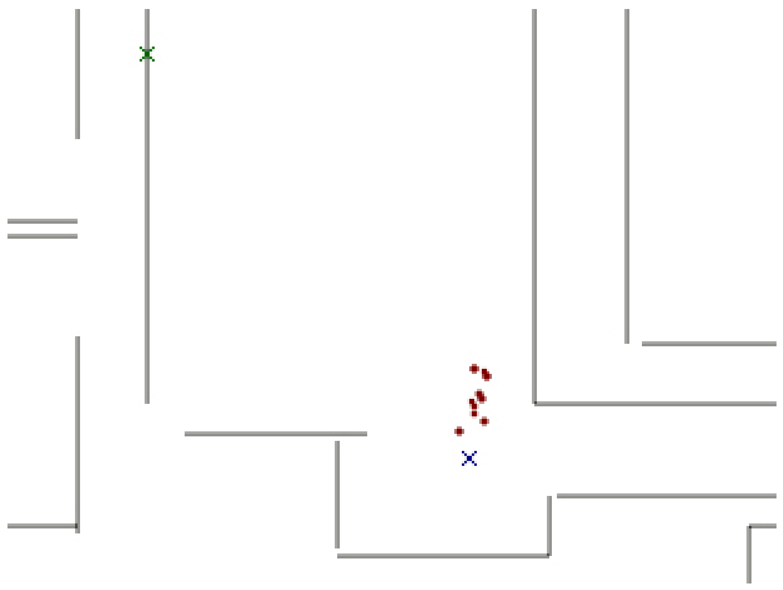

Figure 3: Sample points after resampling based on marker (Green X)

Most research involving mobile robots with high error rates bases localization off of either range sensor data (sonar or laser sensors) or involve environments where several markers are visible at all times. A classic example of a high-error environment would a robotic soccer event in which the robots undergo frequent collisions and the players are often legged robots. The designers of robotic soccer teams usually get around this difficulty by ensuring that several markers are visible at all times, such as in the work done by Beetz et al.

## 6.2 Global localization

Global localization is ostensibly one of the great strengths of using artificial markers for localization. Since it is possible to have every marker distinct or with only a few repetitions, as soon as a marker is seen there is only a small range of positions that the robot could possibly be in. This makes absolute localization far easier than in the case of range sensor-based localization or PCA techniques for vision-based localization, where given a single observation there are a great number of places that the robot could be.

The system as implemented can easily localize to an approximate position after observing a single marker, but only if there exists at least one sample near the robot's initial location. If such a sample exists, then upon seeing a marker all other samples will quickly drop away and the robot's position will be known. However, if there is not a sample nearby then most of the samples will merge to the sample most similar to the real robot, which likely will be significantly different from the actual location. If all samples are either behind the marker or more than 10 meters away from the marker, then the samples cannot be adjusted at all, because they will all be given a weight of zero. In such cases, the samples will probably never converge on the robot's location even after many markers are passed unless the samples are reset.

Then, of course, the problem is how to ensure that at least one sample is near to the actual marker location. The method that is used in most range sensor-based Monte Carlo Localization systems is to simply distribute all samples throughout the system when the robot is started. (Russel 908) This seems like the only option when nothing at all is known about the robot's location, but we would need a very large number of samples randomly distributed around the environment to have a decent chance of having one that

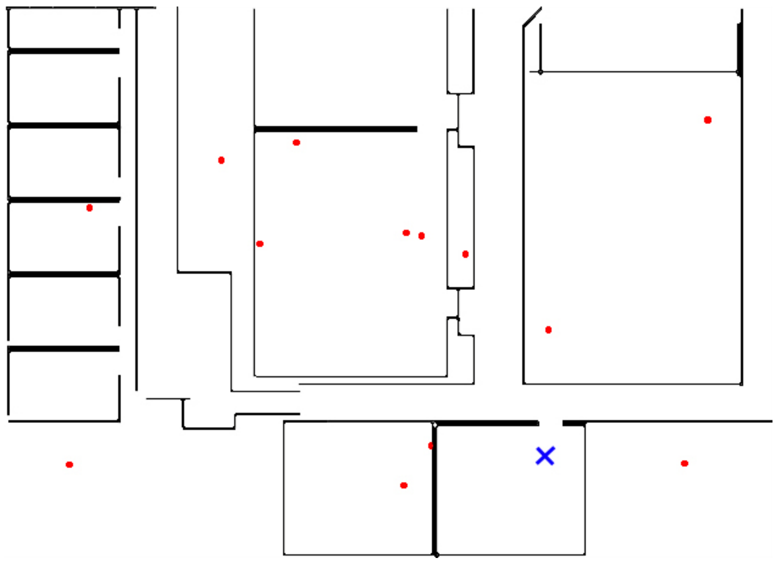

Figure 4: Initial distribution of points (samples are red dots, robot is blue X)

is close to correct. Figure 4 shows a random sample distribution throughout the Olin hall environment using 50 samples. These samples are obviously too sparsely distributed to localize the robot.

Obviously, the number of samples needed to get an accurate location in this size of an environment is very high.

In addition, this method is wasteful because as soon as a marker is observed we know that the robot must be close to that marker, in front of that marker, and in an orientation that gives the robot its observed angle to the marker. We know that all locations that do not fit those criteria have a zero probability of being correct and all positions within that area have a roughly equal probability of being correct, we can update the samples to reflect the new probability information. This gives a very high sample density within

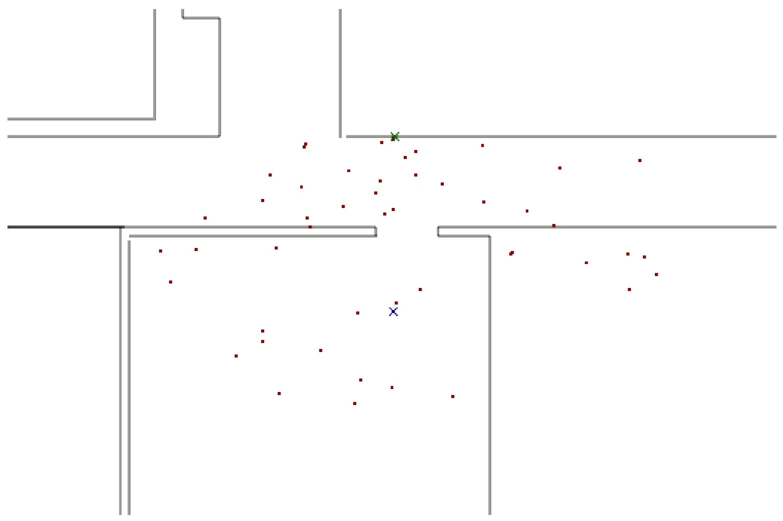

Figure 5: Possible sample positions after observing a marker

that area, as seen in Figure 5.

In the current system, this conditional reset of the sample set is performed whenever it is clear that the robot cannot be localized based on its current sample set. If all samples are given a weight of zero during a particular resampling event, then all samples are discarded and they are reset to random positions within the semicircle in front of the marker, as shown in Figure 5.

This technique makes it often possible to accomplish global localization after only a few marker observations, using only a few dozen sample points. However, if the robot is only moving perpendicular to the marker, there will be nothing to prevent spurious samples from being selected by the MCL algorithm. The robot in Figure 5 is moving directly toward the marker, so there is nothing to support the most correct samples, the result of which is seen in Figure 6.

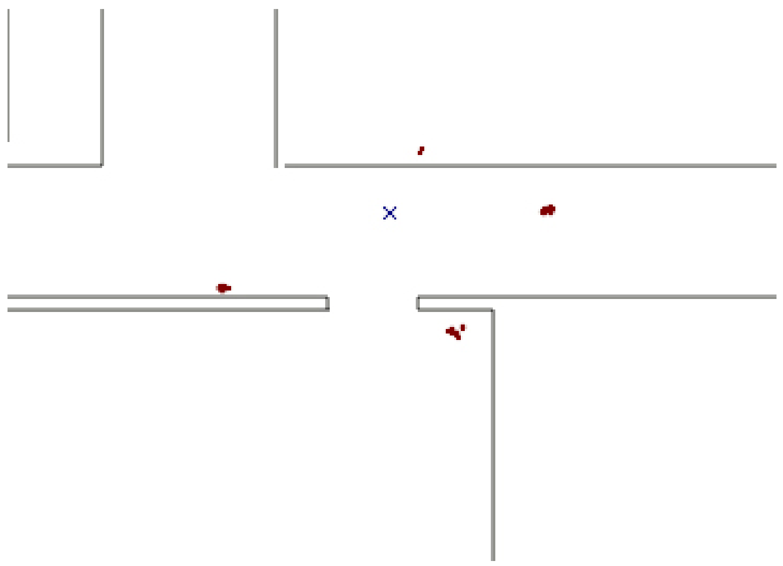

Figure 6: Erroneous sample clusters following global localization attempt

These sample clusters all have the same orientation relative to the marker as the real robot had when the marker was visible, but they are wildly incorrect in actuality and will not allow the robot to localize. Thus, global localization is impractical unless the robot has more to go on than a single marker.

## 6.3 Weaknesses of the system

### 6.3.1 Robot motion error

The robot motion model is faulty in that it assumes that motion sensor error is random and occurs constantly over time. In reality, however, the exact error seen by the motion sensors is very much dependent on its environment at a particular time.

Firstly, the random drift seen as a robot moves across a surface is dependent on the nature of the surface: smooth tile or concrete will generally produce very low error rates, while shag carpet will produce high error rates. The fact that average error rates vary depending on the environment means that it is difficult to have an error model that accurately reflects the robot's error at any given time. On low-error surfaces, the samples will spread unnecessarily to areas that the robot has no possibility of occupying, while on high-error surfaces the robot's real position will likely drift outside of the bounds of the sample set, making good localization very difficult.

Also, the majority of critical motion sensor error does not come in the form of random drift over time, but rather in the form of bursts of error that will occur when the robot encounters certain kinds of difficult terrain. For example, a robot might have little to no motion error when wandering down a hallway, but as it goes over the lip of a doorway its position may change significantly. Another good example of this kind of error is when a robot mistakenly collides with an object, the robot will usually be noticeably pushed or rotated, and when it backs up and continues on its way its position may no longer be accurate. If the error is within the error rate of the motion sensor model, then there is no difficulty with varying rates of error. However, certain bursts of error such as those observed during collisions will undoubtedly outstrip the capacity of the motion model to estimate and cause a loss of position.

This problem could probably be partially fixed by adjusting the error included in the robot motion model to allow for these bursts of error. If the error rate for the samples followed a nonlinear distribution, then with a sufficiently large number of samples even severe errors could be recovered from. The robot motion model could also be designed to change when the robot encounters certain high-error environments.

### 6.3.2 Sample error

The system assumes that at all times at least one sample is very near to the robot's current location. This is a flaw intrinsic to all MCL-based systems: if it is believed that there is a zero probability of a robot being in a particular location, then samples will not be generated in that position even when evidence is presented that supports the robot being in that position. If the closest sample is more than a few meters from the robot's position, then the robot motion model will not provide enough variation in the samples to successfully obtain the robot's location selective sample drift. The robot's motion error is low, so there would be a loss of position if, for example, the robot were to be picked up and moved 2 meters in any direction. Then, even if only the samples which moved closer to the robot were selected during resampling, it would be a long time before the motion error of the robot allowed the samples to catch up with the robot's position. The current system relies on situations like this eventually encountering a marker for which all samples represent positions with zero probability (lying behind or very far away from the marker) and thus forcing a sample reset as described in section 4.2.

### 6.3.3 Vision sensor error

Landmark readings are not accurate enough to perform precise localization. The "position" of the landmark as seen by the robot is based off of the center of the blob of color that is observed, but there are many factors that can cause a marker to be only partially visible. Clipping from the edge of the camera frame and the presence of glare or shadows might cause a portion of the marker to go unrecognized. This is particularly noticeable when the robot is passing a marker at short range, since when the visual diameter of the marker is large a failure to recognize as little as 25% of the marker can cause an error of several degrees in the perceived angle.

### 6.3.4 False markers

There is a slight hazard of false markers. If there is an object introduced to the environment that is sufficiently similar in color to one of the markers used for localization, then it is likely that the samples will be drastically skewed in some unexpected direction. The system I implemented had the advantage of giving a weight of zero to any sample that either appeared on the non-visible side of a marker or was more than 10 meters away from a marker, so a majority of accidental observations would be discounted. However, occasionally a seemingly valid marker will be seen, and in those cases the sample set will converge strongly toward whatever is closest to the spurious observation.

This will likely cause an irrecoverable loss of position and the robot will be forced to perform global localization at the next observed marker.

## 6.4 Conclusions and directions for future work

Probabilistic localization is a practical system to use in conjunction with visual landmarks. The nature of the landmarks makes it highly effective for global localization, but measurements based off of a single marker are not accurate enough to allow the robot to run for long distances without marker observations. This means that the robot must be minimally dependent on localization, the environment must be heavily marked, or some other method must be used to assist tracking.

Probably the best way to fix most of the problems associated with using visual landmarks would be to run the Monte Carlo algorithm off of a combination of range sensor and visual landmark data. It would be possible to base weights off of a combination of landmark data and sonar data when a marker was present, and run exclusively on range sensor data if no marker is present. This system would still be slightly vulnerable to some of the hazards of range-sensor localization (such as error generated by unexpected objects), but even if the robot's position were lost completely due to this, the robot would simply run on dead reckoning until global localization could be accomplished using the visual landmark system at the next marker. Such a hybrid system would eliminate most disadvantages of both systems, but would require both a full map of the environment and a number of markers placed at key locations.

Other possibilities for improvement might involve changing the nature of the markers to minimize error. If the markers were composed of more than one color, or there were several markers placed in close proximity, it would be possible to reduce the error caused by partial views of a marker. Localizing based on several markers in close proximity would also allow for more accurate localization than is possible using a single marker.

In conclusion, probabilistic robot localization using sparsely distributed visual landmarks could be a useful component of a working localization scheme, but measurement error makes it impractical to such a system by itself. The sensor data provided by occasional landmarks are simply not sufficient to maintain an accurate guess of the robot's position: a workable system would require dense landmark placement. However, there are significant advantages in global localization and tolerance of unexpected objects, which make visual landmarks attractive to use in conjunction with other sensor data in probabilistic localization systems.

## A Source code for Monte Carlo Localization

```
/*MonteCarlo.h
 *Base class for Monte Carlo localization
 *maintains and updates a set of samples
 *and provides a set of localization utilities*/

#ifndef MONTE_CARLO_H
#define MONTE_CARLO_H

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <vector>

using namespace std;

/*Keeps track of a robot position (coordinate and orientation) *
 * Used for robot and marker locations and orientations */
class Pos
{
public:
   Pos(double _x = 0, double _y = 0, double _th = 0): x(_x), y(_y), th(_th) {}
   double x,y,th;
};

/*Base class for monte-carlo localization
  is initialized with markers and, optionally,
  the number of samples to maintain*/
class MonteCarlo
{
public:
   //MonteCarlooczliation initializer
   //Ideally allow for start Pos to be initially undefined
   MonteCarlo(Pos startPos, unsigned int nSamples = 20);
 ~MonteCarlo();

   //Sample computations
   double update(Pos newPos, double markerTh, Pos markerPos);
   double update(Pos newPos);
   void setToPos(Pos newPos);
   void setRandPos(double minx,double miny,double maxx,double maxy);
   void setRandPos(Pos markerPos, double markerTh = 0, double minWt = 0);
   double getPrecision(Pos p, double threshold);
   double getAccuracy(Pos p, Pos markerPos, double markerTh);
   Pos getExpectedPos(double minWt, Pos markerPos, double markerTh);
   Pos getExpectedPos();

   //Logging
   int startLogging(char *filename);
   void stopLogging();

   //Window functions
   //void showWindow();

   //accessors
   void setNumSamples(int n);
   const int getNumSamples();
   const vector<Pos> getSamples();
   void setErrorRate(double r) {errorRate = r;}
   double getErrorRate() {return errorRate;}
   static double random() {
      return (double)rand()/(RAND_MAX);
   }


private:
   void updatePos(Pos newPos);
   double updateSamples(double markerTh, Pos markerPos);
   const Pos getMeanPos(vector<Pos> ps);
   const double getEuclideanDist(Pos p1, Pos p2);

   double errorRate;
   double maxVisibleDist;
   unsigned int numSamples;
   Pos currentPos;
   vector < Pos > samples;
   //MCLWindow *window;
   clock_t logStartTime;
   FILE *logFile;
};

#endif
```

```
/* MonteCarlo.cpp
 * Provides functions for MonteCarlo localization using visual landmarks*/

#include "MonteCarlo.h"

MonteCarlo::MonteCarlo(Pos startPos, unsigned int nSamples):
/* Takes a starting position for the robot and
 * the number of samples to maintain
 * many environmental constants are hard-coded here
 * They are set to reasonable values for the Olin hall environment but will
 * need to be changed if this is used in a different environment.*/
      errorRate(.08), //8% error rate
      maxVisibleDist(5), //Markers not visible from more than 5 meters
      numSamples(nSamples),
      currentPos(startPos),
      samples(nSamples),
      logFile(NULL) {
   srand(time(NULL));
}

MonteCarlo::~MonteCarlo() {
   fclose(logFile);
}

double MonteCarlo::update(Pos newPos, double markerTh, Pos markerPos) {
/* Applies the robot motion model and resamples with respect to a
 * known marker position*/
   if(logFile)
      fprintf(logFile,"Time: %f\n",
               (double)(clock()-logStartTime)/CLOCKS_PER_SEC);
   updatePos(newPos); double d = updateSamples(markerTh, markerPos);
   if(logFile) {
      fprintf(logFile,"Pos: %f %f %f\n", newPos.x, newPos.y, newPos.th);
      fprintf(logFile,"Marker %f %f %f at %f radians\n",markerPos.x,
        markerPos.y,markerPos.th,markerTh);
      fprintf(logFile,"Samples: %d\n",samples.size());
      for(int i = 0; i < samples.size(); i++) {
         fprintf(logFile,"%f %f %f\n",
                 samples[i].x,samples[i].y,samples[i].th);
      }
      fputc('\n',logFile);
   }
   return d;
}

double MonteCarlo::update(Pos newPos) {
```

```
/* Applies the robot motion model only*/
   if(logFile)
      fprintf(logFile,"Time: %f\n",
             (double)(clock()-logStartTime)/CLOCKS_PER_SEC);
   updatePos(newPos);
   if(logFile) {
      fprintf(logFile,"Pos: %f %f %f\n", newPos.x, newPos.y, newPos.th);
      fprintf(logFile,"Samples: %d\n",samples.size());
      for(int i = 0; i < samples.size(); i++) {
         fprintf(logFile,"%f %f %f\n",samples[i].x,samples[i].y,samples[i].th);
      }
      fputc('\n',logFile);
   }
   return 0;
}

void MonteCarlo::setToPos(Pos newPos) {
/*Sets all samples to pos newPos */
   for (unsigned int i = 0; i < samples.size(); i++) {
      samples[i] = newPos;
   }
}

void MonteCarlo::setRandPos(double xmin, double ymin, double xmax, double ymax) {
/* Randomly distributes samples across a space
 * defined by the rectangle (xmin,ymin) to (xmax,ymax)*/
   for(int i = 0; i < samples.size(); i++) {
      Pos p(random()*(fabs(xmax-xmin))+xmin,random()*(fabs(ymax-ymin))+ymin,
        (random()*M_PI*2)-M_PI);
      samples[i] = p;
   }
}

void MonteCarlo::setRandPos(Pos markerPos, double markerTh, double minWt) {
/* Randomly distributes samples across the polar space emanating
 * from a marker defined by markerPos
 * This does mean that samples will be denser nearer the marker, but we
 * probably want that anyway*/
   double mx = markerPos.x;
   double my = markerPos.y;
   double mth = markerPos.th;
   for(int i = 0; i < samples.size(); i++) {
      double r = random()*maxVisibleDist; //Out to maxVisibleDist
      double th = (random()*M_PI)+(mth-M_PI_2); //Up to pi/2 radians
      double robx = mx+(r*cos(th));
      double roby = my+(r*sin(th));
      double robth = (random()*2*M_PI)-M_PI; //atan2(roby-my,robx-mx)+
        M_PI/3+(random()*M_PI*(4/3));
      Pos p(robx,roby,robth);
      double wt = getAccuracy(p,markerPos,markerTh);
      if(wt < minWt) {
         i--;
         continue;
      }
      samples[i] = p;
   }
}

void MonteCarlo::setNumSamples(int n) {
/*Changes the number of samples maintained
 * Fills in with samples from old sample set if needed */
   samples.resize(n);
   for (int i = numSamples; i < n; i++) { //fill in the rest, if needed
      samples[i] = samples[i%numSamples];
   }
   numSamples = n;
}

const int MonteCarlo::getNumSamples() { return numSamples; }
const vector<Pos> MonteCarlo::getSamples() { return samples; }

void MonteCarlo::updatePos(Pos newPos) {
/*Applies robot motion model, applying localization if marker is present
 *Assume that Position is updated fairly often
 *(at least 1/sec should be enough)
 *but actual marker-based localization may have long intervals inbetween
 *also, marker Positions are assumed to be known for now but this
 *also should be changed if global localization is desired */
   double dX = newPos.x - currentPos.x;
   double dY = newPos.y - currentPos.y;
   double dTh = newPos.th - currentPos.th;

   //start by getting last travel in terms of last direction traveled:
   //dParallel is forward with respect to the robot, and dPerp is
   //orthogonal to that
   double dParallel = dX*cos(currentPos.th) + dY*sin(currentPos.th);
   double dPerp = dY*cos(currentPos.th) - dX*sin(currentPos.th);

   //current movement error model has each coordinate (x,y,th)
   //independent in error
   //In reality the vast majority of error comes from th error, but
   //this should be easily seen as long as updating is frequent
   //Percentages are based on estimates, and should be updated more
   //rigorously at some point
   double th;
   for (unsigned int i = 0; i < samples.size(); i++) {
      th = samples[i].th;
      samples[i].x += ((dParallel*cos(th))-(dPerp*sin(th)))*
        (1+(random()*errorRate*2 - errorRate));
      samples[i].y += ((dPerp*cos(th))+(dParallel*sin(th)))*
        (1+(random()*errorRate*2 - errorRate));
      samples[i].th += dTh*(1+(random()*errorRate*2 - errorRate));
   }

   currentPos = newPos;
}

double MonteCarlo::updateSamples(double markerTh, Pos markerPos) {
/* Applies particle filtering to samples
 * Samples behind the marker and distant samples are given 0 weight,
 * other samples are given 1/deltaTheta
 * Returns the average weight*/
   vector < double > wts(samples.size());
   unsigned int i, j;
   double d;
   for (i = 0; i < samples.size(); i++) {
      double dX = markerPos.x-samples[i].x;
      double dY = markerPos.y-samples[i].y;
      //if the sample is behind the marker, it can be ruled out entirely
      if ((dX*cos(markerPos.th)) + (dY*sin(markerPos.th)) > 0 ||
            sqrt(pow(dX,2)+pow(dY,2)) > maxVisibleDist)
         wts[i] = 0;
      else {
         //Using P(th|th') = 1/|th-th'|
         d = 1/fabs(markerTh-(atan2(dY,dX)-samples[i].th));
         wts[i] = (d > 10) ? 10 : d; //limit the weight to 10
      }
   }
   //Normalize weights
   double wtSum = 0;
   for (i = 0; i < numSamples; i++)
      wtSum += wts[i];
   if(wtSum == 0) {
   //Let's reset to random positions near the marker
   //if there are no valid samples
      fprintf(stderr,"No possible samples, resetting near marker
       (%f,%f,%f)\n",markerPos.x,markerPos.y,markerPos.th);
      this->setRandPos(markerPos,markerTh,8);
      return 0;
   }
   for (i = 0; i < numSamples; i++)
```

```
      wts[i] /= wtSum;
   //And resample
   vector < Pos > oldSamples(samples);
   double r = 0.0;
   for (i = 0; i < samples.size(); i++) {
      r = random();
      for(j = 0; r > 0; j++) {
         r -= wts[j];
      }
      samples[i] = oldSamples[j-1];
   }
   return wtSum/numSamples;
}

const Pos MonteCarlo::getMeanPos(vector<Pos> ps) {
/* Gets the mean of all samples */
   int i;
   int nPoses = ps.size();
   double xsum = 0;
   double ysum = 0;
   double thsum = 0;
   for (i = 0; i < nPoses; i++) {
      xsum += ps[i].x;
      ysum += ps[i].y;
      thsum += ps[i].th;
   }
   Pos meanPos(xsum/nPoses, ysum/nPoses, thsum/nPoses);
   return meanPos;
}

Pos MonteCarlo::getExpectedPos(double minWt, Pos markerPos, double markerTh) {
   /* Expected pos is the mean of all samples better than minWt */
   int i;
   Pos meanPos;
   int nSamples;

   meanPos = getMeanPos(samples);
   nSamples = samples.size();

   vector<Pos> goodSamples(0);

   for (i = 0; i< nSamples; i++) {
      if(getAccuracy(samples[i],markerPos,markerTh) >= minWt)
         goodSamples.push_back(samples[i]);
   }

   if (goodSamples.size() == 0) {
      return Pos(-1,-1,-1); //No can do
   }

   meanPos = getMeanPos(goodSamples);
   return meanPos;
}

Pos MonteCarlo::getExpectedPos() {
   /* Alternatively, expected pos can be the mean with 10% thrown away to
    * account for outliers*/
   int i,j;
   Pos meanPos;
   int nSamples;

   meanPos = getMeanPos(samples);
   nSamples = samples.size();

   vector<double> dists(nSamples);
   vector<Pos> goodSamples(samples);
   for (i = 0; i< nSamples; i++) {
      dists[i] = getEuclideanDist(meanPos,samples[i]);
   }
   //grab the closest 90%
   for (i = 0; i < int(nSamples*.1); i++) {
      int maxIndex = 0;
      double maxVal = 0;
      for (j = 0; j < nSamples; j++) {
         if (maxVal<dists[j]) {
            maxVal = dists[j];
            maxIndex = j;
         }
      }
      goodSamples.erase(goodSamples.begin()+maxIndex);
      dists[maxIndex] = 0;
   }
   meanPos = getMeanPos(goodSamples);
   return meanPos;
}

double MonteCarlo::getPrecision(Pos p, double threshold) {
/*returns the percent of samples that are within threshold
 * distance of a particular location */
   int count = 0;
   for (unsigned int i = 0; i < samples.size(); i++) {
      if (getEuclideanDist(samples[i],p)<threshold)
         count++;
   }
   return ((double)count/samples.size());
}

double MonteCarlo::getAccuracy(Pos p, Pos markerPos, double markerTh) {
/* Gets the weight that would be assigned to pos p if it were a sample
 * This will be in the range [0,10] */
   double w;
   double dX = markerPos.x-p.x;
   double dY = markerPos.y-p.y;
   //if the sample is behind the marker, it can be ruled out entirely
   if ((dX*cos(markerPos.th)) + (dY*sin(markerPos.th)) > 0 ||
         sqrt(pow(dX,2)+pow(dY,2)) > maxVisibleDist)
      w = 0;
   else {
      //Using P(th|th') = 1/|th-th'|
      double d = 1/fabs(markerTh-(atan2(dY,dX)-p.th));
      w = (d > 10) ? 10 : d; //limit the weight to 10
   }
   return w;
}

const double MonteCarlo::getEuclideanDist(Pos p1, Pos p2) {
/* Returns the euclidean distance from pos1 to pos2
 * within 3-dimensional space (x,y,th) where 1 radian = 1 meter */
   return sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2)+pow(p1.th-p2.th,2));
}

//logging functions

int MonteCarlo::startLogging(char *filename) {
/* Starts writing log output to filename
 * will stop when object is destroyed or stopLogging is called */
   logFile = fopen(filename, "w");
   if (logFile) {
      logStartTime = clock();
      return 1;
   }
   return -1;
}

void MonteCarlo::stopLogging() {
/* Stops logging and closes the logfile */
   fclose(logFile);
}
```

## B Source code for navigation and localization in RUPART

```
# CBRBrain.py
# A Behavior-based control system based on pyro
#(http://www.pyrorobotics.org)

from pyro.brain.fuzzy import *
from pyro.brain.behaviors import *
from pyro.brain import select
from math import *
from behaviors import *
from BehavCBR import CBRRetriever
from copy import copy
from AriaPy import ArACTS_1_2, ArACTSBlob
from string import find
from time import sleep, time
from threading import Thread, Lock
from pathmap import *

#sony range -> ZOOM = 0,1023 PAN = -95,95 TILT = -25,25

#------------------------------------------
#Brain

class CBRBrain(BehaviorBasedBrain):
    """ A behavior-based brain that stores
        additional information and CBR capabilities"""
#---------------------------------------
#Initialization and destruction routines
    def __init__(self,behaviors = {},engine = 0,startLoc = [0,0,0],\
                 goal = [],current_sonar = 16*[0],threshhold = 10,timeout = 7):
        BehaviorBasedBrain.__init__(self,behaviors,engine)
        self.history_size = 5
        self.sonar_history = self.history_size*[current_sonar]
        self.last_average = reduce(lambda x,y: x+y, current_sonar)
        self.threshhold = threshhold
        self.timeout = timeout
        self.finalGoal = goal
        self.markerLoc = []
        self.goal = []
        self.route = []
        self.lastUpdate = time()
        self.startLoc = startLoc #let's save it for future reference
        self.robot.localize(startLoc[0],startLoc[1],startLoc[2])
        self.transVel = .2 #.3
        self.rotVel = .2 #.3
        self.camera = self.get('/devices/ptz0/object')
        self.acts = ArACTS_1_2()
        self.acts.openPort(self.robot.dev)
        self.numChannels = 4
        self.lock = Lock()
        self.map = buildOlinMap()
        self.addMarkers()
        self.retriever = CBRRetriever(self.map, doNormalize = True)
        self.lastMarker = ''
        self.prevGoalDist = 1000

    #must disconnect from ACTS to prevent double-opening
    def __del__(self):
        BehaviorBasedBrain.__del__(self)
        self.acts.closePort()

    #(destroy is called on close and on brain Reload)
    def destroy(self):
        BehaviorBasedBrain.destroy(self)
        self.acts.closePort()

    #List markers here until a better system is devised
    def addMarkers(self):
        #current colors: {'Pink':1, 'TGreen':2, 'Orange':3, 'Yellow':4}
        #markers are x,y,th,[color]
        self.map.addMarker("Lab hall", [22,7.0,270,1])
        self.map.addMarker("East T", [20,5.5,90,3])
        self.map.addMarker("Study center", [5.25,10.2,0])
        self.map.addMarker("East L", [32.2,6.5,180])
        self.map.addMarker("Atrium center hall", [19,39,0])
#        self.map.addMarker("Atrium south hall", [5.25,41,0])
        self.map.addMarker("Atrium north hall", [30,38.5,0])
#        self.map.addMarker("227 hall", [7,26.1,180])


#--------------------------------------
#CBR functions

    #Adds a new case to the caselib
    #Without args, will use current state
    def grabCase(self,body = 'closest',x = 'current',y = 'current'):
        if(x == 'current'):
            x = self.robot.get('/robot/x')
        if(y == 'current'):
            y = self.robot.get('/robot/y')
        activeStates = []
        for s in self.states.keys():
            if self.states[s].status == 1:
                activeStates.append(self.states[s].name)
        ind = [self.getLastSonar(),self.getAvgSonar(),self.robot.dev.getVel(),\
               self.robot.dev.getRotVel(),[x,y],self.goal,activeStates]
        if(body == 'closest'):
            body = apply(self.retriever.retrieve,ind)[0]

        print "Added case:"
        print "Index:"
        print ind
        print "Body:"
        print body

        ind.append(body)
        apply(self.retriever.addCase,ind) #Add the case to caseLib

    #saves current case library to fname
    def save(self,fname):
        self.retriever.saveCaseLib(fname)

    #loads current case library from fname
    def load(self,fname):
        self.retriever.loadCaseLib(fname)

    #find which marker is nearest the current location
    def findNearestMarker(self,color = False):
        print "Color = ", color
        closest = ''
        closeVal = 1000
        rPose = [self.get('/robot/x'),self.get('/robot/y')]
        for loc in self.map.getMarkedLocs():
            mPose = self.map.getMarker(loc)
            if(len(mPose) >= 4 and color and color != mPose[3]):
                print "Skipping..."
                continue  #if the marker is the wrong color, skip
            dist = hypot(mPose[0]-rPose[0],mPose[1]-rPose[1])
            if(closeVal > dist):
                closeVal = dist
                closest = mPose
        return closest, closeVal

    #Update if the average sonar distance has changed
    #significantly since last CBR execution
    def needUpdate(self,current_state):
        sonar_reading = current_state.get('/robot/range/all/value')
        self.sonar_history.append(sonar_reading)
        self.sonar_history.pop(0)
        avg = reduce(lambda x,y: x+y, sonar_reading)
        blobSize = self.getBlobSize()
        #if the robot has no current goal, get it from the route
        #print "Current state is ", current_state.name
        if(self.goal == []):
            current_state = self.getNextGoal(current_state)

        #if the robot has reached its goal, get the next one,
        #or localize if there is a marker here
        elif(self.reachedGoal()):
            visited = self.route[0]
            print "Reached Goal ", self.route.pop(0)
#            if(self.markerLoc and visited != self.lastMarker):
#                current_state.goto('ReLocalize',current_state.name)
#                self.lastMarker = visited
#                self.goal = []
#                return -1
#            else:
            current_state = self.getNextGoal(current_state)

        #If we find an unexpected marker, we have probably missed the last goal
#        elif(blobSize > 200 and self.route != []):
#            print "Testing for marker.."
#            nearest = self.findNearestMarker()
#            if(nearest[0] != self.lastMarker and \
#                nearest[1] <
# hypot(self.get('/robot/x')-self.goal[0],self.get('/robot/y')-self.goal[1])):
#                print "Unexpected marker found.
# Relocalizing to marker near", nearest[0]
#                self.markerLoc = self.map.getMarker(nearest[0])
#                self.lastMarker = nearest[0]
#                current_state.goto('ReLocalize',current_state.name)
#                self.route = []
#                self.goal = []
#                return -1
#
        #Call the CBR system if appropriate
        if((fabs(avg - self.last_average) > self.threshhold) \
        or (time()-self.lastUpdate > self.timeout))\
            and isinstance(current_state,CBRState):
            self.lastUpdate = time()
            self.last_average = reduce(lambda x,y: x+y, sonar_reading)
            return self.CBR(current_state)

        return -1
```

```
#Tests whether the goal is reached: in order
#for it to be reached the following must occur:
#1: The robot must be within 1m of the goal
#2: The robot must be traveling away from the goal
#(Thus finding the closest point to the goal that the robot traverses)
def reachedGoal(self):
    dist = self.goalDist()
    dx = dist-self.prevGoalDist
    self.prevGoalDist = dist
    return (dx > 0 and dist < 1.0)

#CBR retriever; currently never causes state change but
#will adjust speeds and weights
def CBR(self,current_state,update = True):
    body, matchVal =\
    self.retriever.retrieve(self.getLastSonar(),self.getAvgSonar(),\
                self.robot.dev.getVel(),self.robot.dev.getRotVel(),\
           [self.robot.get('/robot/x'),self.robot.get('/robot/y')],\
                       self.goal,[current_state.name],self.finalGoal)
    if(update):
        if(body[0] == "behavior"):
            print "Decision: staying in state Wander."
            self.transVel = body[2]
            self.rotVel = body[3]
            current_state.goto('Wander',body[4])
        elif(body[0] == "route"):
            print "Set route", body[2]
            while type(body[2][0]) != str:
            #Get rid of extraneous location coords
                body[2].pop(0)
            self.route = body[2]
            self.getNextGoal(current_state)
    return matchVal

#Trash the current goal/markerLoc and get the next one from the route
#If current_state is provided, trigger the appropriate behavior
def getNextGoal(self, current_state = 0):
    if(self.route == []):
    #if there is no route, cannot go anywhere,
    #so stop the robot and wait for inspiration
        self.goal = []
        self.markerLoc = []
        if(self.states['Wait'].status == 0):
            print "Waiting..."
            if(current_state):
                self.states[current_state.name].goto('Wait')
                return self.states['Wait']
            return current_state
        else:
            print "Setting goal..."
            print "Route = ", self.route
            self.goal = self.map.getLoc(self.route[0])
            self.markerLoc = self.map.getMarker(self.route[0])
            self.prevGoalDist = 1000
            print  "Goaldist = ", self.goalDist(), "\n";
            if(current_state and self.goalDist() > .6):
                self.states[current_state.name].\
                    goto('FaceGoal',current_state.name)
                return self.states['FaceGoal']
            return current_state



#--------------------------------------------------
#Localization and Vision utilities

    #Locates a blob if one exists, returns False otherwise
    def findBlob(self):
        b = ArACTSBlob()
        maxArea = 30
        blobExists = False
        channel = 0
        for i in range(self.numChannels):
            #print "Channel ", i, " ", self.acts.getNumBlobs(i+1)
            if(self.acts.getNumBlobs(i+1) > 0):
                self.acts.getBlob(i+1,1,b)
                if(b.getArea() > maxArea):
                    blob = b
                    channel = i+1
                    maxArea = b.getArea()
                    blobChannel = i+1
                    blobExists = True
        if(blobExists):
            #print "maxArea = " + str(maxArea)
            return blob, channel
        return False, False

#(deprecated)
    #determine whether the camera can see a blob or not
#    def blobExists(self):
#        self.lock.acquire()
#        be = self.findBlob
#        self.lock.release()
#        if(be):
```

```
#            return True
#        return False

    #Returns th, color of blob
    #(current color vals are Pink:1, Green:2, Orange:3, Yellow:4)
    def getBlobInfo(self):
        self.lock.acquire()
        blob, color = self.findBlob()
        self.lock.release()
        if(type(blob) == ArACTSBlob):
            return int((blob.getXCG()-80)*\
              ((48.8-((self.camera.getZoom())*(44.5/1023)))/160)),color
        else:
            return False,False

    #Gets the color of the currently tracked blob
    def getBlobColor(self):
        self.lock.acquire()
        color = self.findBlob()[1]
        self.lock.release()
        return color

    #Gets the angle that the currently tracked blob is on
    def getBlobTh(self):
        self.lock.acquire()
        blob = self.findBlob()[0]
        self.lock.release()
        if(type(blob) == ArACTSBlob):
            return int((blob.getXCG()-80)*\
              ((48.8-((self.camera.getZoom())*(44.5/1023)))/160))
        else:
            return False

    #Gets the size of the current blob
    def getBlobSize(self):
        self.lock.acquire()
        blob = self.findBlob()
        self.lock.release()
        if(type(blob) == ArACTSBlob):
            return blob.getArea()
        else:
            return False

    #gets the angle and size of the current blob [th, size]
    def getBlobAll(self):
        self.lock.acquire()
        blob = self.findBlob()
        self.lock.release()
        if(type(blob) == ArACTSBlob):
            return int((blob.getXCG()-80)*\
              ((48.8-((self.camera.getZoom())*(44.5/1023)))/160)), blob.getArea
        else:
            return False

#assumes the robot is facing a wall in position as per ReLocalize/ReLocalize2
   #Will set th to the angle facing the wall, and will approximate x and y
   #FIX: x and y are off because of the camera's placement at the front
   #of the robot-
   #a few calculations should take care of this, though it is not imperative

   #Camera X,Y is [.11,0]
   def doLocalization(self):
       markerLoc = self.getMarkerLoc()
       th = markerLoc[2]+180
       if(th >= 360):
           th -= 360
       frontDist = self.get('/robot/range/front/value')
       frontTh = self.get('/robot/range/front/thr')
       frontPos = self.get('/robot/range/front/ox,oy')
       for d in frontPos:  #divide all by 1000 (sonar pos given in mm..)
           for i in d.keys():
               d[i] /= 1000
       #increase the distance to reflect the distance
       #from the center of the robot to the sonar
       frontDist = [x+frontPos[0]['oy']/sin(radians(frontTh[0])) \
        for x in frontDist]
       #increase the percieved distance by the distance
       #from the vertex of the triangle to the
       #center of the robot
       dist = (cos(radians(frontTh[0]))*((frontDist[0]+frontDist[1])/2))+\
              (frontPos[0]['ox']-(frontPos[0]['oy']/tan(radians(frontTh[0]))))
       blobTh = self.getBlobTh()
       if(not blobTh):
       #Scan the viewable area if the blob is not visible for some reason
           print "Blob not found..."
           self.camera.pan(-95)
           sleep(2) #wait for camera to pan
           blobTh = self.getBlobTh()
           while(not blobTh):
               self.camera.panRel(5)
               sleep(.4) #wait for camera to pan
               blobTh = self.getBlobTh()
               if(self.camera.getPan() > 90):
                   blobTh = 1
        blobTh = -1*(self.camera.getPan()+blobTh)
        if(blobTh <= -90 or blobTh >= 90):
            print "Blob angle " + str(blobTh) + " out of bounds."
            return -1
        #find the distance from the camera
        #(located dist-.11 away from the wall) to the blob
        markerDist = fabs((dist-.11)/cos(radians(blobTh)))
        #find the real x and y of the robot (camera location, adjusted)
        x = (markerLoc[0]-(cos(radians(th+blobTh))*markerDist))\
         -(cos(radians(th))*.11)
        y = (markerLoc[1]-(sin(radians(th+blobTh))*markerDist))\
         -(sin(radians(th))*.11)
        self.robot.localize(x,y,th)
        print "Dist = ", dist
        print "markerDist = ", markerDist
        #print "blobTh = ", blobTh
        print "Localized at " + str(x) + ' ' + str(y) + ' ' + str(th)


#------------------------------
#General Utility Functions

    #deactivates all states
    def deactivateAll(self):
        for s in self.states.keys():
            self.deactivate(self.states[s].name)

    #Finds the direction corresponding with a particular angle
    def direction(self, angle):
        if(angle > 0):
            return 1
        if(angle < 0):
            return -1
        return 0

    #Prints the current sonar information
    def printSonar(self):
        print self.getLastSonar()
        print "\n"
        print self.getAvgSonar()
        print "\n"

#------------------------------
#Accessors

    #accessor for marker location
    def getMarkerLoc(self):
```

```
    return self.markerLoc

#accessor for goal location
def getGoalLoc(self):
    return self.goal

#accessors for translational velocity
def getTransVel(self):
    return self.transVel

def setTransVel(self,val):
    self.transVel = val

#find how far it is to the immediate goal
def goalDist(self):
    return hypot(self.get('/robot/x')-self.goal[0],\
      self.get('/robot/y')-self.goal[1])

#accessors for rotational velocity
def getRotVel(self):
    return self.rotVel

def setRotVel(self,val):
    self.rotVel = val

def setPos(self,x,y,th):
    self.states['Track'].behaviors['MCLocalization'].setPos(x,y,th)
    self.robot.localize(x,y,th)

def setRandPos(self):
    self.states['Track'].behaviors['MCLocalization'].setRandPos(0,0,40,50)
    #we run in OLRI for now

#accessor for latest sonar reading
def getLastSonar(self):
    return self.sonar_history[-1]

#get the average sonar value for each of the 16 sensors
def getAvgSonar(self):
    a = len(self.sonar_history[0])*[0]
    for l in self.sonar_history:
        for i in range(len(self.sonar_history[0])):
            a[i] += l[i]
    return map(lambda x: x/self.history_size,a)

def printSamples(self):
    for i in range(20):
            p = self.states['Track'].behaviors['MCLocalization'].\
             MCL.getSamples()[i]
            print p.x, ", ", p.y, ", ", p.th

    def logSamples(self,filename):
        self.states['Track'].behaviors['MCLocalization'].\
                             MCL.startLogging(filename)

#--------------------------------
#init
def INIT(engine): # passes in robot, if you need it
    s = filter(lambda x: find(x,'ptz') >= 0, engine.robot.get('/devices'))
    if(s == []):
        ptzID = engine.robot.startDevice("ptz-sony")
    else:
        ptzID = s[0][:-1]
    ptz = engine.robot.get("devices/%s/object" % ptzID)
    #22,4,90
    startpos = [22,4,90]
    #startpos = [20,13,270]
    destination = '222 room'
    #destination = 'Study door'
    brain = CBRBrain({'translate' : engine.robot.translate, \
                      'rotate' : engine.robot.rotate, \
                      'update' : engine.robot.update, \
                      'pan' : ptz.panRel }, engine, startpos, destination)

    # add the state
    #print "Attempting pan..."
    #engine.robot.set('/devices/ptz0/pan', 90)

    brain.add(Wander(1))
    brain.add(Track(1))
    brain.add(Wait())
    brain.add(Halt())
    brain.add(Stall())
    brain.add(FaceGoal())
    #brain.add(ReLocalize())
    #brain.add(ReLocalize2())
    #sleep(10)
    #brain.states['Wander'].goto('Wander',\
      {'Avoid': {'translate': .15,'rotate': .2}, 'Escape': {'rotate': .6}})
    return brain
```

```
# behaviors.py
# A collection of behavior classes for a
# a Behavior-based robot control system
# Several experimental but unused behaviors can be found in backup/behaviors.py

from pyro.brain.fuzzy import *
from pyro.brain.behaviors import *
from pyro.brain import select
from math import *
import AriaPy
import random
from CBRBrain import *
import time
from localization import *

#CBRBehavior supplies a few basic behaviors that all behaviors will want to have
class CBRBehavior (Behavior):
    """A base class for behaviors using the CBR system"""

    #initialize 4/5 the maximum range of a sonar sensor
    def setup(self):
        self.max_sensitive = self.get('/robot/range/maxvalue')*.8

    #clean way of activating the behavior
    def activate(self):
        self.status = 1

    #clean way of deactivating the behavior
    def deactivate(self):
        self.status = 0

#Basic behavior, usually active, moves forward when
#there are no obstacles in the way
#and turns to avoid the closest obstacle
class Avoid (CBRBehavior):
    """A simple avoid behavior"""

    def update(self):
        if(self.get('/robot/stall') > -258):
            self.state.goto('Stall',self.state.name)
        close = select(min, "value",\
          self.get('/robot/range/front-all/value,thr'))
        close_all, angle = close["value"], close["thr"]
          #require half the distance to slow
          #down because of the front-side sensors
        close_front = min(self.get('/robot/range/3,4/value')\
              + map(lambda x: x*2,self.get('/robot/range/2,5/value')))
        self.IF(Fuzzy(0.2, self.max_sensitive/2) >> \
                close_front, 'translate',self.brain.transVel, "Ok")
        self.IF(Fuzzy(0.2, self.max_sensitive/2) << \
                close_front, 'translate',0, "TooClose")
        #self.IF(Fuzzy(0.1, self.max_sensitive/2) << close_all,\
                 'translate', 0.0,"TooCloseOther")
        self.IF(Fuzzy(0.1, self.max_sensitive) << close_all, 'rotate',\
          -1*self.brain.direction(angle)*self.brain.rotVel, "TooClose")
        self.IF(Fuzzy(0.1, self.max_sensitive) >> close_all,\
                'rotate', 0.0, "Ok")

#The reverse of avoid, moves in reverse and is only called when
#the robot detects a stall
class Backup (CBRBehavior):
    """Back up, avoiding obstacles if possible"""

    def update(self):
        close = select(min, "value",\
                self.get('/robot/range/back-all/value,thr'))
        close_all, angle = close["value"], close["thr"]
        #require half the distance to slow
        #down because of the front-side sensors
        close_back = min(self.get('/robot/range/11,12/value') + \
          map(lambda x: x*2,self.get('/robot/range/10,13/value')))
        self.IF(Fuzzy(0.1, self.max_sensitive/2) >> close_back, \
          'translate', -1*self.brain.transVel, "Ok")
        self.IF(Fuzzy(0.1, self.max_sensitive/2) << close_back, 'translate', \
          .05, "TooClose")
        self.IF(Fuzzy(0.1, self.max_sensitive) << close_all, 'rotate',\
          -1*self.brain.direction(angle)*self.brain.rotVel, "TooClose")
        self.IF(Fuzzy(0.1, self.max_sensitive) >> close_all,\
                'rotate', 0.0, "Ok")

#Behavior used to go through doorways and other passages
#that the robot would ordinarily avoid
#Goes towards the most prominent 1-2 sensor wide "gap"
#(1 or 2 long readings surrounded by short readings)
#Use with caution, because sonar reflection errors on the walls can
#easily cause fake gaps to appear and
#result in the robot being driven into the wall
class Escape (CBRBehavior):
    """A behavior to escape close, trapped situations"""
    def update(self):
        sonar = self.get('/robot/range/left,front-all,right/value')
        maxDiff = [0,-1]
        for i in range(1,8):
```

```
        #see what sensor has the most pronounced
        #length compared to those around
            j = min(sonar[i]-sonar[i-1],sonar[i]-sonar[i+1])
            if(j > maxDiff[0]):
                maxDiff = [j,i]
        for i in range(1,7): #check for 2sensor-wide gaps
            k = min(sonar[i]-sonar[i-1],sonar[i+1]-sonar[i+2])
            if(k > maxDiff[0]):
                maxDiff = [k,i+.5]
        self.IF(Fuzzy(0, self.max_sensitive*.75) >> maxDiff[0], 'rotate',\
          self.brain.direction(4.5-maxDiff[1])*self.brain.rotVel, "IsExit")

#Behavior to follow the right wall
#Not necessary in the current CBR
#implementation because path planning is point-based
class FollowRight (CBRBehavior):
    """A right-hand wall following behavior"""
    def update(self):
        close = select(min, "value", self.get('/robot/range/6,7/value,thr'))
        close_right, angle = close["value"], close["thr"]
        self.IF(Fuzzy(0.5, self.max_sensitive) >> close_right, 'rotate',\
          -1*self.brain.rotVel,"TooFar")

#Behavior to follow the left wall
#Not necessary in the current CBR implementation
#because path planning is point-based
class FollowLeft (CBRBehavior):
    """A left-hand wall following behavior"""
    def update(self):
        close = select(min, "value", self.get('/robot/range/0,1,2/value,thr'))
        close_left, angle = close["value"], close["thr"]
        self.IF(Fuzzy(0.5, self.max_sensitive) >> close_left, 'rotate',\
          self.brain.rotVel, "TooFar")

#Have the robot face the goal, and slow down when very close
#Effect of the behavior is inversely proportional
#to the current distance
#This is slightly undesirable, as points in large,
#open areas can easily be missed
#Further work should allow the sphere of
#influence for a point to be defined
#individually for each point
#(Currently, the behavior is at base strength at a number of
#meters equal to self.baseMeters, hard-coded)
class ToGoal (CBRBehavior):
    """A behavior that orients the robot towards the goal"""
    def setup(self):
        CBRBehavior.setup(self)
        self.baseMeters = 1.5

    def update(self):
        if(self.brain.goal == []):
        #shouldn’t happen much, but don’t want a crash when goal is killed
            return -1
        xdif = self.brain.goal[0] - self.get(’/robot/x’)
        ydif = self.brain.goal[1] - self.get(’/robot/y’)
        #don’t want the weighting to more than
        #quadruple; want to avoid /0 errors
        dist = max(hypot(xdif,ydif),self.baseMeters*.25)
        th = degrees(atan2(ydif,xdif))
        th_dif = th-self.get(’robot/th’)
        if(th_dif <= -180):
            th_dif += 360 #make sure that -180 < th_dif <= 180
        th_abs = fabs(th_dif)
        self.IF(Fuzzy(.25,2) << dist, ’translate’, 0, "Nearby")
        self.IF(Fuzzy(0, 180) >> (self.baseMeters*th_abs)/(dist), ’rotate’,\
                self.brain.direction(th_dif)*self.brain.rotVel, "WrongDir")
        self.IF(Fuzzy(0, 180) >> (self.baseMeters*(90-th_abs))/(dist), \
          ’rotate’, 0, "OnTrack")

#When a blob is visible, tracks it with the camera
#If no blob is present, behaves in a fashion determined
#by the search variable in Track, its state
#When searching, the camera moves more or
#less at random to find a blob
#When not searching, the camera faces
#directly ahead and assumes it will find a blob
class TrackBlob (Behavior):
    """A behavior to have the camera point towards the nearest blob"""
    def onActivate(self):
        self.nextMove = 0

    def update(self):
        #print "Getting BlobTh..."
        th = self.brain.getBlobTh()
        #print "BlobTh =", th
        if(th != 0):
            self.nextMove = time.time()+1.5
        elif(time.time() > self.nextMove):
            #when searching, the camera alternates
            #between a random point on the left,
            #and a random point on the right
            pan = self.brain.camera.getPan()
            if(self.state.search):
                panTo = int(pow(random.random(),2)*90) #randint(0,90)
                if(pan > 0):
                    panTo *= -1
            else:
                panTo = 0
            self.brain.camera.pan(panTo)
            self.nextMove = time.time()+1.7
            #.7+((1/80)*fabs(pan-panTo))+time.time()

        self.IF(Fuzzy(5,25) >> fabs(th), 'pan', \
          int(self.brain.direction(th)*2), "TooFar")


class MCLocalization (Behavior):
    """A behavior to run the Monte-carlo localization scheme"""
    def setup(self):
        self.p = Pos()
        self.markerPos = Pos()
        self.p.x = self.brain.startLoc[0]
        self.p.y = self.brain.startLoc[1]
        self.p.th = (self.brain.startLoc[2])*(pi/180)
        print "Initializing MCL to ", self.p.x,\
              ", ", self.p.y, ", ", self.p.th
        self.MCL = MonteCarlo(self.p,50)
        self.MCL.setToPos(self.p)
        #This will hold the last time the robot's hardware pos was updated
        self.lastPosUpdate = time.time()
        self.minDelay = 5 #This starts the cycle of updating the hw pos

    def onActivate(self):
        self.tick = 0

    def update(self):
        self.tick += 1
        #let's not update every tick, that'd be a bit of a waste--
        #try every third for now
        if (self.tick > 3):
            self.tick = 0
            self.p.x = self.get('/robot/x')
            self.p.y = self.get('/robot/y')
            self.p.th = self.get('/robot/thr')
            th, color = self.brain.getBlobInfo()
            if (th == 0):
                self.MCL.update(self.p)
            else:
                print "Marker observed, resampling.."
                m = self.brain.findNearestMarker(color)[0]
                self.markerPos.x = m[0]
                self.markerPos.y = m[1]
                self.markerPos.th = m[2]*(pi/180)
                markerTh = (-1*(self.brain.camera.getPan()+th))*(pi/180)
                self.MCL.update(self.p,markerTh,self.markerPos)

                #And now, update the hardware position
                #if the samples are fairly clear
                recency = time.time()-self.lastPosUpdate
                if(recency>self.minDelay):
                    threshold = .15 #try this for now
                    expectedPos = Pos()
                    expectedPos = self.MCL.getExpectedPos(3,\
                                  self.markerPos,markerTh)
                    #we want at least 90% of samples to be within the threshold
                    #and we want the current
                    #marker reading to be within .1 radians
                    accuracy  = \
                      self.MCL.getAccuracy(expectedPos,self.markerPos,markerTh)
                    precision = self.MCL.getPrecision(expectedPos,threshold)
                    print "Precision = ", precision, ", Accuracy = ", accuracy
                    if(precision>=.6 and accuracy >= 6):
                        degth = expectedPos.th*(180/pi)
                        print "Localized robot to ", expectedPos.x, ", ",
                         \expectedPos.y, ", ", degth, "\n"
                        #self.brain.printSamples()
                        self.robot.localize(expectedPos.x,expectedPos.y,degth)
                        self.lastPosUpdate = time.time()

    def setPos(self,x,y,th):
        self.MCL.setToPos(Pos(x,y,th))

    def setRandPos(self,x1,y1,x2,y2):
        self.MCL.setRandPos(x1,y1,x2,y2)

#Moves toward the current marker
#Transfers to faceWall when it is close to the wall and facing the blob
class ToBlob (CBRBehavior):
    """Behavior to go toward marker \
      (Followed by faceWall, then localization)"""
    def update(self):
        blobTh = self.brain.getBlobTh()
        blobExists = (blobTh != 0)
        if(blobExists):
            #print "Blob exists!"
            th_dif = -1*(blobTh+self.brain.camera.getPan())
            rotVel = self.brain.getRotVel()
            self.state.lastBlob = time.time()
        elif(self.state.locKnown):
            markerLoc = self.brain.getMarkerLoc()
            if(not markerLoc):
                raise ValueError, "No marker for current location"
            xdif = markerLoc[0] - self.get('/robot/x')
            ydif = markerLoc[1] - self.get('/robot/y')
            th = degrees(atan2(ydif,xdif))
            th_dif = th-self.get('robot/th')
            if(th_dif <= -180):
                th_dif += 360 #make sure that -180 < th_dif <= 180
            rotVel = 1
        else:
            th_dif = 50
            rotVel = .3
        th_abs = fabs(th_dif)
        close = min(self.get('/robot/range/front-all/value'))
        goForward = blobExists and (close > .75) and (th_abs < 30)
        self.state.done = blobExists and (close < .75) and (th_abs < 30)
        self.IF(Fuzzy(0, 180) >> th_abs, 'rotate', \
          self.brain.direction(th_dif)*rotVel, "WrongDir")
        self.IF(Fuzzy(0, 180) << th_abs, 'rotate', 0, "OnTrack")
        self.IF(Fuzzy(0,1) >> goForward,'translate',.1,"Go towards blob")
        self.IF(Fuzzy(0,1) << goForward,'translate',0,"Stop to find blob")

#Localization routine part 2
#Lines up with the wall, in preparation for localization
class FaceWall (CBRBehavior):
    """Behavior to face the wall, so localization can commence"""
    def update(self):
        close = select(min, "value",\
           self.get('/robot/range/front-all/value,thr'))
        closeVal, angle = close["value"], close["thr"]
        front2 = self.get('/robot/range/3,4/value')
        diff = fabs(front2[0]-front2[1])
        self.IF(Fuzzy(0.1, 1) >> closeVal, 'rotate', \
          self.brain.direction(angle)*.2, "Closer")
        self.IF(1,'translate',0,"No movement in this state")
        self.IF(Fuzzy(0, .5) << diff, 'rotate', 0, "Lined up")

#-------------------------------------
#States

#All cases controlled by the CBR system should call
#needUpdate to determine whether a CBR call is needed
class CBRState (State):
    """ Basic syntax for all states """
    def update(self):
        self.brain.needUpdate(self)

#not part of CBR, always active
#Runs TrackBlob
class Track (State):
    """ Keeps the camera pointing at the closest blob """
    def setup(self):
        self.add(TrackBlob(1, {'pan': .3}))
        self.add(MCLocalization(1))
        self.search = True
        print "initialized state", self.name

    def doSearch(self):
        self.search = True

    def noSearch(self):
        self.search = False

#Right wall-following behavior; currently unused
class RightWall (CBRState):
    """ Right-hand wall-following state """
    def setup(self):
        self.add(Avoid(1, {'translate': .3, 'rotate': .3}))
        self.add(FollowRight(1, {'rotate': .3}))
        print "initialized state", self.name

#Left wall-following behavior; currently unused
class LeftWall (CBRState):
    """ Left-hand wall-following state """
    def setup(self):
        self.add(Avoid(1, {'translate': .3, 'rotate': .05}))
        self.add(FollowLeft(1, {'rotate': .3}))
        print "initialized state", self.name

#Primary state, allows parameters to be passed in specifying the weights of
#Avoid, Escape, and ToGoal as well as the robot's base velocity
class Wander (CBRState):
    """ Wander aimlessly """
    #Weights passed in as {bName: {eName: wt, ...}, ...}
    def onGoto(self,weights = ()):
        if 'Track' in self.brain.states.keys():
            self.brain.states['Track'].doSearch()
        if(not(weights == ())):
            wts = weights[0]
            #onGoto is passed params as a tuple, so must extract arg
            for bName in wts.keys():
                b = self.behaviors[bName]
                if(wts[bName] == 0):
                    b.deactivate()
                else:
                    b.activate()
                    for eName in wts[bName].keys():
                        b.Effects(eName,wts[bName][eName])
                        #set the effect of eName to the provided value

    def setup(self):
        self.add(Avoid(1, {'translate': .3, 'rotate': .3}))
        self.add(Escape(1, {'rotate': .1}))
        self.add(ToGoal(1, {'rotate': .05, 'translate': .05}))
        print "Initialized state", self.name

#Unused for the most part, called when the robot needs to stop
class Halt(State):
    """Stop the robot entirely and halt pyro execution"""
    def onGoto(self,args):
        self.robot.move(0,0) #halt the robot
        self.brain.needToStop = True #signal pyro to stop

    def setup(self):
        print "Initialized state", self.name

#Does nothing, but will continue to
#call needUpdate so that the robot can resume
class Wait(CBRState):
    """Stop the robot and wait for CBR to change state"""
    def onActivate(self):
        self.robot.move(0,0)
        #stop moving, wait for needUpdate to change state

    def setup(self):
        print "Initialized state", self.name

#Paired with ReLocalize2, gets near the
#wall with the marker and rotates so that
#the robot is perpendicular to the wall
class ReLocalize (State):
    """Rotate to find the closest match"""
    def setup(self):
        self.add(ToBlob(1, {'translate': .2, 'rotate': .2}))
        self.calling_state = "Halt"
        print "Initialized state", self.name

    def onGoto(self,calling_state):
        self.calling_state = calling_state[0]

    def onActivate(self):
        if 'Track' in self.brain.states.keys():
            self.brain.states['Track'].noSearch()
        self.timeout = 7
        self.lastBlob = time.time()
        self.done = False
        self.locKnown = True

    def update(self):
        if(self.done):
            print "On Phase 2!"
            self.goto('ReLocalize2',self.calling_state)
        elif(time.time()-self.lastBlob > self.timeout*3):
            print "No blob found, resuming..."
            self.goto(self.calling_state)
        elif(time.time()-self.lastBlob > self.timeout and \
             self.locKnown == True):
            print "No blob found, trying full circle..."
            self.locKnown = False

#Second phase of localization,
#orients the robot perpendicular to the wall
class ReLocalize2 (State):
    def setup(self):
        self.add(FaceWall(1, {'rotate': .2, 'translate': .2}))
        print "Initialized state", self.name

    def onGoto(self,calling_state):
        print "In phase 2..."
        self.calling_state = calling_state[0]

    def update(self):
        front2 = self.get('/robot/range/3,4/value')
        diff = fabs(front2[0]-front2[1])
        if(diff < .005 and diff != 0):
        #exact match means neither sonar is responding
        #(and so we obviously aren't there yet:D)
            self.robot.rotate(0)
            time.sleep(.5)
            #wait and double check to eliminate errors found due to rotation
            front2 = self.get('/robot/range/3,4/value')
            diff = fabs(front2[0]-front2[1])
            if(diff < .005):
                self.brain.robot.disableMotors()
                time.sleep(1) #wait for any motion to settle
                self.brain.doLocalization()
                self.brain.robot.enableMotors()
                self.goto(self.calling_state)

#Turns the robot toward the goal
#Does not have an associated behavior,
#as it merely turns the robot toward the
#goal and returns to the calling state
class FaceGoal (State):
    """ Turn toward the goal; get within 10 degrees  """
    def onActivate(self):
        print "Trying to face goal..."
        if(not self.brain.goal):
            print "No goal, resuming..."
            self.goto(self.calling_state)
        xdif = self.brain.goal[0] - self.get('/robot/x')
        ydif = self.brain.goal[1] - self.get('/robot/y')
        self.thGoal = degrees(atan2(ydif,xdif))
        if(self.thGoal < 0):
            self.thGoal += 360
        self.robot.translate(0)
        #Stop any previous motion, as ToGoal does not specify trans vals
        th_dif = self.thGoal-self.get('robot/th')
        if(th_dif > 180):
            th_dif -= 360
        if(th_dif <= -180):
            th_dif += 360 #make sure that -180 < th_dif <= 180
        self.robot.rotate(.3*self.brain.direction(th_dif))

    def onGoto(self,calling_state):
        self.calling_state = calling_state[0]

    def update(self):
        diff = fabs(self.get('/robot/th')-self.thGoal)
        #print "diff: ", diff, "\n"
        if(diff < 5 or diff > 355):
            print "Resuming..."
            self.robot.rotate(0)
            self.goto(self.calling_state)

    def setup(self):
        print "Initialized state", self.name

#Activated whenever a stall is detected
#Returns to the calling state after 2 seconds, or after a stall in reverse
class Stall (CBRState):
    """ Back up and try again on collisions """
```

```
def onActivate(self):
    self.start_time = time.time()
    time.sleep(1)
    print "In state Stall"

def onGoto(self,calling_state):
    self.calling_state = calling_state[0]

def update(self):
    CBRState.update(self)
    if(time.time()-self.start_time > 2.0 or \
       self.get('/robot/stall') > -258):
        print "Trying again..."
        self.goto(self.calling_state)

def setup(self):
    self.add(Backup(1, {'translate': .2, 'rotate': .05}))
    print "Initialized state", self.name
```

# C Source code for test run display window

```
/* Display.h
 * creates a DisplayWindow and a kit to open it */

#ifndef DISPLAY_H
#define DISPLAY_H

#include "DisplayWindow.h"
#include <gtkmm/main.h>

class Display
{
public:
   Display(int argc, char **argv);

   void show();

private:
   Gtk::Main kit;
   DisplayWindow window;
};

#endif
```

```
/* Display.cpp
 * creates a kit for showing a displayWindow */

#include "Display.h"

Display::Display(int argc = 0, char **argv = NULL):  kit(argc,argv), window()
{}

void Display::show()
{
   Gtk::Main::run(window);
}

int main(int argc, char **argv)
{
   Display d(argc,argv);
   d.show();
   return 0;
}
```

```
/* Displaywindow.h
 *Class definition for base window of sample displayer*/

#ifndef DISPLAYWINDOW_H
#define DISPLAYWINDOW_H

#include <gtkmm/button.h>
#include <gtkmm/window.h>
#include <gtkmm/box.h>
#include <gtkmm/entry.h>
#include "MapArea.h"
#include <vector>

class DisplayWindow: public Gtk::Window
{
public:
   DisplayWindow();
   virtual ~DisplayWindow() {}

protected:
   virtual void onLoadMapClicked();
   virtual void onPrevSampleClicked();
   virtual void onNextSampleClicked();
   virtual void onLoadSamplesClicked();

   Gtk::HBox hBox;
   Gtk::VBox vBox;

   MapArea map;
   Gtk::Entry dataFile;
   Gtk::Button loadMap;
   Gtk::Button prevSample;
   Gtk::Button nextSample;
   Gtk::Button loadSamples;

   std::vector<std::vector<std::vector<double> > > samples;
   std::vector<std::vector<double> > markers;
   std::vector<std::vector<double> > poses;
   std::vector<double> times;
   int currentSample;

};

#endif
```

```
/* DisplayWindow.cpp
 * For window class of displayer for MonteCarlo samples */

#include "DisplayWindow.h"
#include <fstream>
#include <iostream>
#include <string>

DisplayWindow::DisplayWindow()
   :  loadMap("Load Map"),
      loadSamples("Load Samples"),
      prevSample(" << "),
      nextSample(" >> "),
      currentSample(0)
{
   set_border_width(10);

   loadMap.signal_clicked().connect
     (sigc::mem_fun(*this, &DisplayWindow::onLoadMapClicked));
   loadSamples.signal_clicked().connect
     (sigc::mem_fun(*this, &DisplayWindow::onLoadSamplesClicked));
   nextSample.signal_clicked().connect
     (sigc::mem_fun(*this, &DisplayWindow::onNextSampleClicked));
   prevSample.signal_clicked().connect
     (sigc::mem_fun(*this, &DisplayWindow::onPrevSampleClicked));

   add(vBox);

   dataFile.set_max_length(50);

   hBox.pack_start(prevSample,Gtk::PACK_SHRINK);
   hBox.pack_start(nextSample,Gtk::PACK_SHRINK);
   hBox.pack_start(loadMap,Gtk::PACK_SHRINK);
   hBox.pack_start(loadSamples,Gtk::PACK_SHRINK);
   hBox.pack_start(dataFile,Gtk::PACK_SHRINK);
   vBox.pack_start(hBox,Gtk::PACK_SHRINK);
   vBox.pack_start(map);

   prevSample.show();
   nextSample.show();
   loadMap.show();
   loadSamples.show();
   dataFile.show();
   map.show();
   hBox.show();
   vBox.show();
```

```
}

//See the previous sample
void DisplayWindow::onPrevSampleClicked()
{
   if(currentSample > 0) {
      currentSample--;
      map.setSamples(samples.at(currentSample));
      map.setPos(poses.at(currentSample));
      map.setMarker(markers.at(currentSample));
   }
}

//See the next sample
void DisplayWindow::onNextSampleClicked()
{
   if(currentSample < samples.size()-1) {
      currentSample++;
      map.setSamples(samples.at(currentSample));
      map.setPos(poses.at(currentSample));
      map.setMarker(markers.at(currentSample));
   }
}

//Load a map from a file
void DisplayWindow::onLoadMapClicked()
{
   std::ifstream in;
   in.open(dataFile.get_text().c_str());
   if(!in) {
      std::cerr << "Error reading map file\n";
      return;
   }

   std::string str("");
   int maxx,maxy,minx,miny,numLines;

   while(str != "LineMinPos:") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
   }

   in >> minx;
   in >> miny;
   while(str != "LineMaxPos:") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
   }

   in >> maxx;
   in >> maxy;

   while(str != "NumLines:") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
   }

   in >> numLines;

   while(str != "LINES") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
   }

   std::vector<std::vector<int> > lines(numLines);
   for(int i = 0; i < numLines; i++) {
      lines[i].resize(4);
      in >> lines[i][0] >> lines[i][1] >> lines[i][2] >> lines[i][3];
   }

   map.setMapLines(lines,maxx,maxy,minx,miny);
}

//Load sample logfile
void DisplayWindow::onLoadSamplesClicked()
{
   std::ifstream in;
   in.open(dataFile.get_text().c_str());

   samples.resize(0);
   markers.resize(0);
```

```
times.resize(0);
poses.resize(0);

if(!in) {
   std::cerr << "Error reading map file\n";
   return;
}

std::string str("");
int i,nSamples;
double time;


while(str != "Time:") {
   in >> str;
   if(in.eof()) {
      std::cerr << "Formatting error in file..\n";
      return;
   }
}

in >> time;
times.push_back(time);

while(!in.eof()) {
   std::vector<std::vector<double> > v;
   std::vector<double> pos(3);
   std::vector<double> mpos(3);
   bool m = 0;

   while(str != "Pos:") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
   }


   in >> pos[0] >> pos[1] >> pos[2];

   while(str != "Samples:") {
      in >> str;
      if(in.eof()) {
         std::cerr << "Formatting error in file..\n";
         return;
      }
```

```
            if(str == "Marker") {
               std::cerr << "Marker entry found..\n";
               in >> mpos[0] >> mpos[1] >> mpos[2];
               m = 1;
            }
         }

         if(!m) {
            mpos.resize(0);
         }

         in >> nSamples;

         v.resize(nSamples);
         for (i = 0; i < nSamples; i++) {
            v[i].resize(3);
         }

         for (i = 0; i < nSamples; i++) {
            in >> v[i][0] >> v[i][1] >> v[i][2];
         }

         while(str != "Time:" && !in.eof()) {
            in >> str;
         }
         if(!in.eof()) {
            in >> time;
            times.push_back(time);
         }

         poses.push_back(pos);
         samples.push_back(v);
         markers.push_back(mpos);
      }

      map.setSamples(samples.at(0)); //start with the first set of samples
      map.setPos(poses.at(0));
      map.setMarker(markers.at(0));
      currentSample = 0;
   }
```

```
/* MapArea.h
 * Class for the drawing area where the map and samples are drawn */

#ifndef MAP_AREA_H
#define MAP_AREA_H

#include "gtkmm/drawingarea.h"
#include <vector>

class MapArea: public Gtk::DrawingArea
{
   public:
      MapArea();
      virtual ~MapArea();

      void setMapLines(std::vector<std::vector<int> >
        &lines, int xmax, int ymax, int xmin, int ymin);
      void setSamples(std::vector<std::vector<double> > &samples);
      void setPos(std::vector<double> &p);
      void setMarker(std::vector<double> &m);

   protected:
      virtual bool on_expose_event(GdkEventExpose* event);
      double lineWidth;

      void forceRedraw();

      int maxX,maxY,minX,minY,zoom;
      std::vector<std::vector<int> > mapLines;
      std::vector<std::vector<double> > samples;
      std::vector<double> pos;
      std::vector<double> marker;
};

#endif
```

```
/*MapArea.cpp
 * Definitions for drawing the map for displaying samples */

#include "MapArea.h"

#include "cairomm/context.h"
#include "stdio.h"

MapArea::MapArea():
   minX(0),
   minY(0),
   maxX(0),
   maxY(0),
   lineWidth(1),
   zoom(25)
{}

MapArea::~MapArea() {}

bool MapArea::on_expose_event(GdkEventExpose* event)
{
   // This is where we draw on the window
   Glib::RefPtr<Gdk::Window> window = get_window();
   if(window)
   {
      Gtk::Allocation allocation = get_allocation();
      const int width = allocation.get_width();
      const int height = allocation.get_height();

      Cairo::RefPtr<Cairo::Context> cr(new
        Cairo::Context(gdk_cairo_create(window->gobj()), true));

      if (event)
      {
         // clip to the area indicated by the expose event so that we only redraw
         // the portion of the window that needs to be redrawn
         cr->rectangle(event->area.x, event->area.y,
               event->area.width, event->area.height);
         cr->clip();
      }

      // scale to unit square and translate (0, 0) to be (0,0)
      // cr->scale(width, height);
      cr->scale(1,1);
      cr->translate(0, 0);
```

```
cr->set_line_width(lineWidth);

//draw the map lines
for(int i = 0; i < mapLines.size(); i++) {
   cr->move_to((mapLines[i][0]-minX)/zoom,
       height-(mapLines[i][1]-minY)/zoom);
   cr->line_to((mapLines[i][2]-minX)/zoom,
       height-(mapLines[i][3]-minY)/zoom);
   cr->save();
   cr->set_source_rgb(0,0,0);
   cr->stroke();
   cr->restore();
}

//draw the samples
for(int i = 0; i < samples.size(); i++) {
   int x = (int(samples[i][0]*1000)-minX)/zoom;
   int y = (int(samples[i][1]*1000)-minY)/zoom;

   cr->move_to(x-1,height-(y-1));
   cr->line_to(x+1,height-(y+1));
   cr->move_to(x-1,height-(y+1));
   cr->line_to(x+1,height-(y-1));
   cr->save();
   cr->set_source_rgb(.5,0,0);
   cr->stroke();
   cr->restore();
}

//draw the robot pos
if(!pos.empty()) {
   int x = (int(pos[0]*1000)-minX)/zoom;
   int y = (int(pos[1]*1000)-minY)/zoom;

   cr->move_to(x-3,height-(y-3));
   cr->line_to(x+3,height-(y+3));
   cr->move_to(x-3,height-(y+3));
   cr->line_to(x+3,height-(y-3));
   cr->save();
   cr->set_source_rgb(0,0,.5);
   cr->stroke();
   cr->restore();
}

//draw the marker
if(!marker.empty()) {
   int x = (int(marker[0]*1000)-minX)/zoom;
```

```
            int y = (int(marker[1]*1000)-minY)/zoom;

            cr->move_to(x-3,height-(y-3));
            cr->line_to(x+3,height-(y+3));
            cr->move_to(x-3,height-(y+3));
            cr->line_to(x+3,height-(y-3));
            cr->save();
            cr->set_source_rgb(0,.4,0);
            cr->stroke();
            cr->restore();
         }

         return 1;
      }
   }

   void MapArea::setMapLines(std::vector<std::vector<int> >
     &map, int xMax, int yMax, int xMin = 0, int yMin = 0)
   {
      mapLines = map;
      maxX = xMax;
      maxY = yMax;
      minX = xMin;
      minY = yMin;
      forceRedraw();
   }

   void MapArea::setSamples(std::vector<std::vector<double> > &v)
   {
      samples = v;
      forceRedraw();
   }

   void MapArea::setPos(std::vector<double> &p)
   {
      pos = p;
      forceRedraw();
   }

   void MapArea::setMarker(std::vector<double> &m)
   {
      marker = m;
      forceRedraw();
   }

   void MapArea::forceRedraw()
   {
       Glib::RefPtr<Gdk::Window> win = get_window();
       if (win)
       {
           Gdk::Rectangle r(0, 0, get_allocation().get_width(),
                   get_allocation().get_height());
           win->invalidate_rect(r, false);
       }
   }
```

# References

[1] Arkin, Ronald. ”Behavior-based Robotics” The MIT Press. Cambridge, MA: 1998.

[2] Beetz, Michael et al. ”‘The AGILO autonomous robot soccer team: computational principles, experiences, and perspectives.” In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, 2002.

[3] Fox, S. and Anderson-Sprecher, P., ”Robot Navigation Using Integrated Retrieval of Behaviors and Routes,” to appear in *Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference*, 2006.

[4] Isard, Michael and Blake, Andrew. ”Contour tracking by stochastic propagation of conditional density,” *Proceedings of the European Conference on Computer Vision*, vol 1, pp. 343-356, Cambridge UK, 1996

[5] Krose, B.J.A, Vlassis, N., Bunschoten, R., Motomura, Y. ”A probabilistic model for appearance-based robot localization,” *Image and Vision Computing*, 19, 2001, 381-391.

[6] Marques, C.F. and Lima, P.U. ”Vision-based self-localization for soccer robots,” *Proceedings of the 2002 International Conference on Intelligent Robots and Systems*, vol 2, pp. 1193-1198, Takamatsu, Japan, 2002.

[7] Murphy, Robin R. ”Introduction to AI Robotics” Cambridge, MA: MIT Press, 2000.

[8] ”Pryo: Python Robotics” http://www.pyrorobotics.org.

[9] Russel, Stuart, and Norvig, Peter. ”Artificial Intelligence: A Modern Approach, Second Edition” New Jersy: Prentice Hall, 2002.

[10] Smith, Lindsay I. ”A tutorial on Principal Components Analysis,” http://csnet.otago.ac.nz/cosc453, 2002.