2D Scanning Sonar Modeling and Algorithm Development for Robots Conducting Underwater Tunnel Mapping

Austin E. Walker

Class of 2013

Prof. Christopher Clark

Submitted to: Princeton University

Department of Computer Science

1 Introduction

A variety of enclosed underwater environments are of interest to many different disciplines. For example, underwater caves are especially interesting to biologists because they provide a very unique environment for life; these caves are home to many rare species that little is known about. Archaeologists and historians are interested in the information that shipwrecks and ancient cisterns can provide about human history. Thus, exploring and mapping underwater environments offers unique opportunities to learn more about the world we live in past and present.

However, underwater environments are often difficult, expensive, or dangerous to visit in person. Human divers need an air supply, can only dive so deep, and require extensive training. Manned submersibles are costly and bulky. To avoid these issues, remotely operated underwater vehicles (or ROVs) are commonly used to explore these environments and gather data from them. A typical ROV (see Figure 1) is connected to the operator via a tether which provides power to the vehicle, allows the data it collects to be passed up to the operator, and allows the operator to control the ROV. Cameras and sensors including underwater sonar allow for data to be collected while the ROV moves.

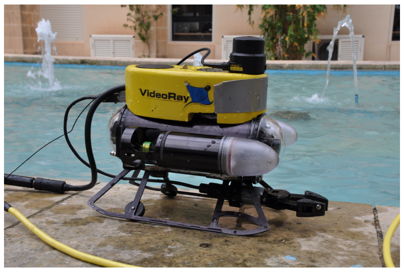


Figure 1: A typical small ROV used for underwater exploration.

This project aims to develop a system for mapping marine caves and other similar environments using a small ROV equipped with scanning sonar, and in doing so, demonstrate that this is a feasible objective. The work done here is based on past work done as part of the Malta Cistern Mapping Project, which aims to produce maps of ancient wells and cisterns throughout Malta and Sicily.

1.1 Background Work

Previous work in underwater robotic mapping has focused on algorithms for Simultaneous Localization And Mapping (SLAM, sometimes referred to as Concurrent Mapping and Localization or CML). Localization is the task of determining where an object (in this case, the robot) is within a given map, and mapping is the task of making that map. SLAM algorithms are designed to use input data to simultaneously build a map and allow the robot conducting the mapping to determine where it is within that map. Williams et al. (2000) describes one of the first applications of SLAM to underwater navigation and mapping and used a large, custom-build robot with sophisticated sonar equipment for mapping.

Another major theme in previous work is the use of particle filter based SLAM algorithms. A particle filter works by simulating many (hundreds or thousands of) different possible states of the ROV at the same time; each instance is a "particle". As data (sonar measurements and location information) is fed into the particle filter, particles are assigned weights based on how well the input data matches up with the previous data with respect to the state that each individual particle maintains for the ROV. As the simulation progresses, particles with low weights are likely to be removed, and particles with high weights are likely to survive and be replicated. Chen (2003) gives a very in-depth history and survey of particle filtering techniques.

Different ways of representing the map of an environment have been used in past work. Tardós et al. (2002) in which a wheeled robot was used for indoor mapping used a feature-based mapping representation which specific features such as corners were tracked. The underwater work in Williams et al. (2000) also used a feature-based representation. More recent work has used an approach known as occupancy grid mapping. An occupancy grid map discretizes the world into a regular grid of cells in which a map is created. Each cell in this grid represents a regular section of space (a square in two dimensions or a cube in three). At each cell, a value is stored to indicated the presence or lack of an obstruction in the space that cell represents. Martin and Moravec (1996) gives a good review of the use of occupancy grids in sonar-based robotic mapping.

Very little work has been done that attempts to do underwater, robotic, three-dimensional occupancy grid mapping. The major project which has pursed this task is a tunnel exploration project in Mexico. Fairfield et al. (2006), building on work in in Fairfield et al. (2005) used a large underwater robot equipped with a multi-directional sonar system and a Doppler Velocity Logging system (which is used to determine how far the vehicle moves in what direction very accurately) to explore and map an underwater tunnel, Fairfield et al. (2008) describes the project in great detail. The work focused on the development and validation of a real time mapping system. The system created occupancy grid maps that used an advanced tree structure to allow for large maps to be created quickly and efficiently. The project was able to produce full three-dimensional maps of a tunnel that is hundreds of meters deep in real time.

Some of the most recent (and still ongoing) work with underwater mapping was done as part of the Malta Cistern Mapping Project as described in White et al. (2010). The project focused on creating two-dimensional, top-down maps of ancient cisterns in Malta using a small ROV and is a continuation of of the work done described in Clark et al. (2008). The project explored several methods of creating maps including stitching together a series of stationary sonar scans by hand, stitching scans together using information from a special tether for the ROV known as a smart tether which frequently reports an estimate of the vehicle's location, and using a particle filter-based SLAM algorithm known as FastSLAM. The project also did some work with creating maps from sonar measurements taken while the ROV was in motion. While this project produced some excellent results, it did not work with three-dimensional maps and identified the move to three-dimensions as a logical next step. The Malta Cistern Mapping Project is of particular relevance to this work since the same equipment was be used for this project as was used in that project.

1.2 Problem Definition

Formally, the problem at hand can be described as follows: given an ROV that is capable of moving about in an underwater environment and which is equipped with a sonar scanner, build a full three-dimensional map of the environment. The ROV's state is described by

$$X = [x\ y\ z\ \theta\ \sigma\ \alpha]$$

where (x, y, z) are the ROV's position coordinates in three dimensions relative to some initial location (x_0, y_0, z_0), θ is the rotation around the vertical axis where zero is defined to be north-facing, σ is the estimated error in the position and orientation of the ROV, and α describes the orientation of the sonar head on the ROV with respect to the ROV's coordinates (see Figure 2 and Figure 3). The sonar head can either be

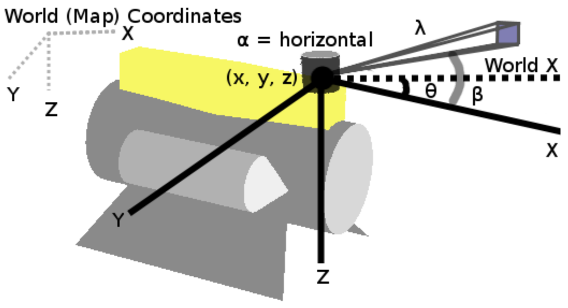


Figure 2: Diagram of ROV State for a horizontally mounted sonar head.

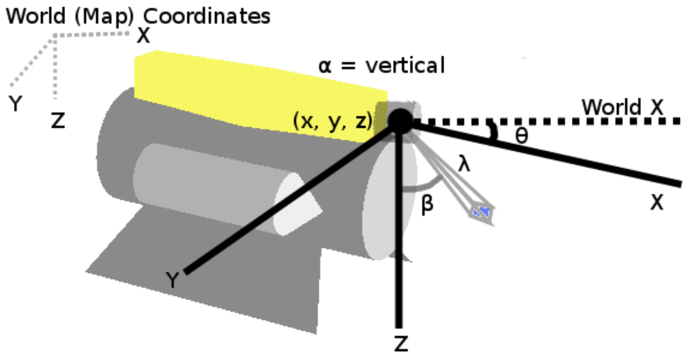


Figure 3: Diagram of ROV State for a vertically mounted sonar head.

mounted horizontally, in which case the sonar's coordinates are the same as the ROV's, or vertically, in which case the sonar's coordinates are perpendicular to the ROV's (with an angle of zero corresponding to the *z* direction of the ROV). To simplify the problem, the ROV is assumed to have zero pitch and zero roll at all times, which means that the sonar will scan in a plane that is either perfectly horizontal or perfectly vertical in the world coordinate system. This assumption is based on the fact that the ROV is ballasted in order to keep it level, and that its depth is controlled via a vertically oriented thruster and not using angled control surfaces which tilt the vehicle in order to change depth.

As the ROV moves, it takes measurements that report the level of sonar reflectance. These measurements take the form of sonar rays; a ray r_i is described by

$$r_i = [t\, X_t\, \beta\, \lambda\, n \{d_1, d_2, d_3, \ldots, d_n\}]$$

where *t* is the time the ray was measured, X_t is the state of the ROV at that time, β is the angle of sonar ray with respect to the orientation of the sonar head α (for α = horizontal, β is measured from the ROV's X axis, for α = vertical, β is measured from the ROV's Z axis), and λ is the range of the ray. The main data of each ray consists of an array of values that indicate reflectance measured at evenly spaced locations along a straight path extending from the ROV; there are *n* values given by d_1 through d_n (see Figure 4). Each reflectance value d_i ranges from 0 to 1, where 0 indicates no reflectance (the sonar is passing through empty space), and 1 indicates very high signal reflectance (the sonar is reflecting off something solid).

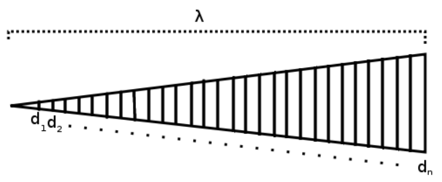


Figure 4: Diagram of a sonar ray.

Sonar rays are collected as parts of scans. A scan *s* is comprised of a continuous series of rays that are collected as the sonar head rotates around:

$$s = \{r_1, r_2, r_3, \ldots\}$$

where r_1, r_2, etc. are all temporally sequential sonar rays that line within the same two-dimensional plane. For this research, X_t is assumed to be the same for each r_i in a scan; that is, the ROV is assumed to be nearly stationary throughout each scan. The conducted experiments took scans by either parking the ROV at the bottom or by carefully hovering in order to maintain this assumption. The basic mapping problem can is defined as follows: given a set S of scans s_1, s_2, etc., create a best-estimate map M of what the environment around the ROV looks like based on the information in S.

Since sonar scans only provide information about a two-dimensional plane, no reasonable number of scans can provide enough information by themselves in order to create a full three-dimensional map of an environment. Therefore, in order to get a complete map, it is necessary to fill in portions of the map for which there is no sonar data. This must be done in such a way as to reflect the most likely layout of the environment.

Finally, any system for creating a map must be reasonably efficient in order to be useful. The primary efficiency problem is one of memory: a map representation must not take up so much memory that it is unusable. As mentioned in section 1.1, past work has focused on the use of particle filters to perform mapping. The efficiency aspect of this problem is therefore even more important because a particle filter must store many different maps (one for each particle). Memory efficiency is a useful metric for measuring the effectiveness of any mapping solution because any useful mapping solution must be able run without using specialized computing hardware in order to be useful for field work.

2 Mapping

In order to solve the mapping problem, a software system was developed. This system draws from ongoing work on the Malta Cistern Mapping Project. This system's map creation process consists of two main steps. First, by mapping from the coordinates of the each sonar scan to the coordinates of the map, use the information in the scans to directly fill in as much of the map as possible. This requires tracking the ROV state so that the ROV position and rotation can be used to properly place each scan in the map.

The second step in creating a map is to do some post-processing on the result of the first step. Since sonar scans only provide two-dimensional slices of information about the environment, large parts of the resulting map will not be filled in simply because it would require a massive number of two-dimensional scans in order to fill three-dimensional space. However, some missing sections can be filled in by extending the existing map fragments that were filled in by the scans to create a full three dimensional map.

2.1 System Architecture

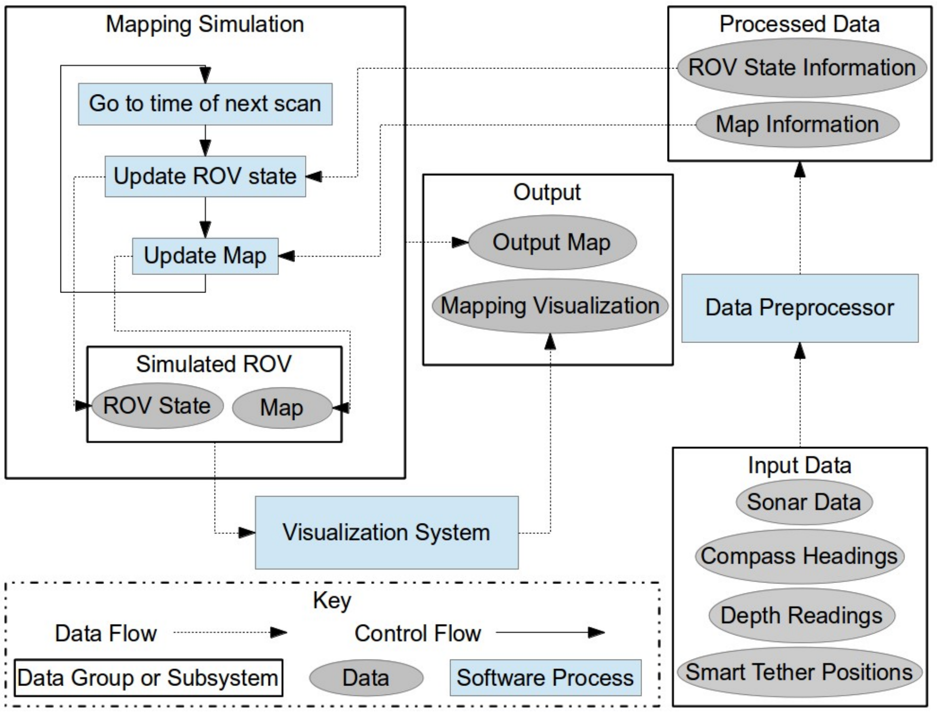


Figure 5: Overall Mapping System Architecture

Figure 5 shows the architecture of the mapping system that was developed. The system essentially runs a time-based simulation of the ROV's movements using the data collected. Input data consists of sonar scans containing information about the environment, compass headings and depth measurements collected from the ROV's onboard sensors, and GPS coordinates that give the location of the ROV as determined by the smart tether. Before the simulation starts, this data is converted into more useful formats in order to be more easily incorporated into the simulation. Sonar measurements are converted from the proprietary format used by the sonar vendor into a the format described above in section 1.2, and smart tether GPS positions are converted into x, y, and z coordinates relative to the ROV's initial position.

Once the data is ready, the mapping simulation is started. This simulation moves through the sonar information in chronological order (the order in which it was collected). For each sonar ray, the system looks for data about the ROV's state. Any data that provides information about the ROV's location or rotation is then used to update the ROV state. Once the state is updated, the data in the sonar ray is used to update the map. This process continues until all of the provided sonar data has been incorporated into the map. At that point, the final map is recorded and outputted.

A visualization subsystem is run in parallel with the main mapping system. Visualization is done using OpenGL to draw a full three-dimensional representation of the current map, the sonar ray being processed, and the position and orientation of the ROV within the environment (see Figure 6). The user can manipulate the viewpoint and control the progress of the simulation.

2.2 Mapping Overview

To represent the environment which the ROV is mapping, an occupancy grid is used. The world around the ROV is discretized into a regular three-dimensional grid in which a map is created. This grid is defined to have some resolution r_M which determines how much space each cell represents by defining the edge length of each cell. For each cell M_{xyz} in the grid, a probability p_{xyz} is stored. p_{xyz} expresses the certainty that a wall (or some other obstruction) exists in the space represented by M_{xyz}; a value of 1 indicates absolute certainty that a wall or other obstruction exists in the space, a value of 0 indicates absolute certainty that the space is empty. Since initially there is no information about the environment

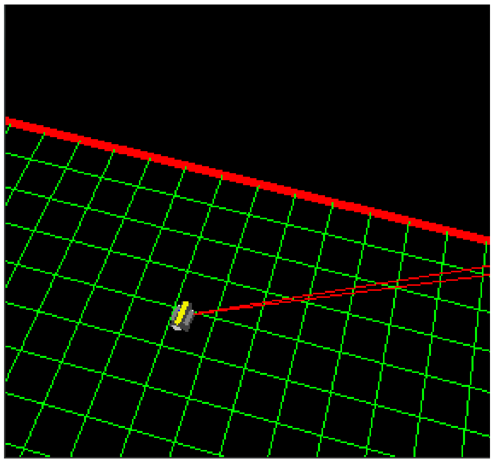


Figure 6: Mapping Visualization. Here, the ROV is shown in an empty map. The red lines extending from the ROV represent the area currently being scanned by sonar.

being mapped, each cell in the map starts off with $p_{xyz} = 0.5$.

As the ROV gathers more data about the world via sonar measurements, the cells of the gird are updated to reflect new information. A sonar measurement at the location of a particular cell M_{xyz} at coordinates (x, y, z) in the map is used to modify the map via a log-odds approach; this technique is adapted from that used in both Farfield et al. (2006) and in White et al. (2010). If the initial value at M_{xyz} is p_{xyz} and sonar reports a value of $\mathrm{p_s}$ at (x, y, z), then the new value p_{xyz}' for M_{xyz} will be calculated as follows:

$$p_m' = \frac{p_{xyz}}{1-p_{xyz}} * \frac{p_s}{1-p_s} \qquad p_{xyz}' = \frac{p_m'}{1+p_m'}$$

This ensures that the value in each map cell stays between 0 and 1 while allowing for multiple scans which overlap to still be incorporated in the map (as opposed to simply filling the map directly with the most recent data available).

2.3 Map Algorithms and Data Structures

In order to actually do any mapping, a concrete implementation is needed. Such an implementation must be reasonably efficient in both the space it takes up and the speed at which it can be updated. These issues are considered here.

To illustrate just how important efficiency is, consider how much space a map of a given size takes up. If the map has cubic cells that are r_M units on edge and represents a volume of size x by y by z units, then $\frac{x*y*z}{r_M^3}$ cells are needed in order to represent the map. Sonar such as that used in the experiments described later in this report can provide a resolution in the tens of centimeters and have a range of up to 80 meters, which means that a single map could require over a millions cells.

By itself, this is not an impossibly large number of cells, but it is also necessary to consider how such an implementation could be used. As noted in section 1.1, previous work uses particle filters in order to simultaneously map an environment and localize the ROV within that map. Since each particle maintains a unique state for the ROV, each particle must have its own map so that input data can be transferred into that map based on the ROV's state within that particular particle. Suddenly, instead of millions of cells, billions of cells must be stored. Without some form of optimization, this would not be possible using anything other than a supercomputer, making it very impractical for field use.

Two different methods are employed to allow for such a large amount of data to be stored. The first is a rather straightforward optimization in which each cell is compressed such that it takes up only a fraction of the space it might be expected to otherwise. The second uses a data structure solution—octrees—to allow for only parts of the map which have actually been filled in to be stored and to avoid storing multiple copies of the same information.

2.3.1 Cell Compression

The simplest way to store a regular gird of cells is as a simple multi-dimensional array. One possible approach is to have each member of the array hold a reference to a dynamically allocated cell structure which holds both the value in that cell and other information about the cell. This has the advantage that cells in the map which have not yet been used can remain unallocated and assumed to have the default starting value of $p_{xyz} = 0.5$. However, this approach suffers form the major drawback that each cell in the map requires memory for the reference pointer to be allocated *even for empty cells* because a null pointer still takes up room.

Instead of having each member of the array reference a cell structure allocated elsewhere, the array can directly hold the certainty p_{xyz} of each cell directly. This creates a tradeoff between space and precision. If each member of the array is allocated *b* bits of space to hold the cell value, then each cell can store 2^b different values. At one extreme, each cell can be though of as either certainly filled or certainly unfilled (ie $b = 1$). At the other end, if *b* is taken to be 32 (which is the case if each cell stores a floating point number), then an extremely fine gradation of different values are possible, but the map will take up 32 times as much space. In practice, using $b = 8$ (using one byte per cell) seems to work well. This allows for 256 different values for a map cell, which is approximately the level of precision that the sonar used is able to provide.

The major drawback to this method is that it makes it difficult to store extra information in each cell. For example, previous work in in both White et al. (2010) and Fairfield et al. (2006) stored the precomputed log of the cell value in order to speed up the computation of updating the map. One could also imagine wishing to store information about the state of the ROV or the data source that provided the information that filled a cell in each cell. While it would be possible to do this, it would require space for that information to be allocated from the start of the simulation (as opposed to when it set for the first time).

2.3.2 Octrees and Hybrid Maps

The second method that used to increase map efficiency is to move from storing the cells of the map in a simple array to using a more complex octree based data structure. An octree is an eight-way branching tree which divides up three-dimensional space by creating a sub-tree for each of the eight coordinate octants. This continues recursively until the space has been divided up into some desired resolution. At the bottom of the tree, each leaf on the tree stores whatever information is to be associated with the space that leaf represents, which is a cell value in this case.

While they do have additional overhead that a simple array does not, octrees are advantageous for mapping because they only require memory to be allocated for parts of the map that have actually been modified. Subtrees within the octree only need to be allocated once some data needs to be inserted into them; subtrees that are not allocated are simply assumed to contain only cells in the default state ($p_{xyz} = 0.5$ for all x, y, and z within the subtree). For sparse data sets, this represents a huge savings over a flat array, and even for very dense data, it can only be slightly worse.

In Fairfield et al. (2006) a method was described to use octrees to implement an occupancy grid map. The method described in that work was called “deferred reference counting octrees” (abbreviated DRCOs) and allows for multiple maps to share some common parts. It works by storing two values at each octree node. The first value, the reference count (“refCount”), stores the number of different maps which directly contain a node within the octree. The second value, the deferred reference count (“defCount”), holds the number of maps which reference a node *and* all of the nodes below it in the tree. Thus, the total number of maps which contain a node is the sum of that node's refCount and the defCounts of it and every node above it in the tree.

The two typical operations to be performed on this type of tree are requests to get the value of a cell in the map and to set the value of a cell. Getting a cell is quite simple: just walk down the tree until the cell that was requested is found. However, setting a cell in the map is more complicated due to the possibility that multiple maps share sections of the map. Since modifying a cell in one map should not affect other maps, the set operation must make a separate copy of the node it intends to change and all nodes above it in the octree if those nodes are shared between maps. The exact workings of the set operation are shown in Figure 7.

Figure 7: Pseudocode for ***DROC_Set(Node N, Value V)*** *as described in Fairfield et al. (2006)*

```
Node p = root //parent node, starts as root
for each Node n in path to N:
    if n.defCount > 0:
        for each child Node c of n:
            c.defCount += n.defCount;
        n.defCount = 0;
    if n.refCount > 0:
        newN = allocateNewNode();
        newN.refCount = 1;
        newN.defCount = 0;
        n.refCount--;
        newN.children = n.children;
        newN.value = n.value;
        //change parent's ref to n to be newN
        if parent != null:
            parent.updateChild(n, newN);
        n = newN;
    p = n;
N.value = V;
```

DRCOs clearly improve upon standard octrees by preventing the storage of redundant data that would happen when multiple maps contain the same information. However, Fairfield et al. (2006) describes another major advantage they offer: the ability to be copied quickly. As that work points out, a typical particle filter used for mapping occasionally goes through a re-sampling operation in which some particles are destroyed and new ones are created by making copies of existing ones. In order to copy a simple octree, the entire tree would have to be completely traversed and copied into a new map. However, by using DRCOs, this operation can be greatly simplified by simply incrementing the defCount of the root node of the octree that is to be copied. This records the fact that the entire tree is now included within a second map, but requires almost no work compared to traversing the entire octree. The work of propagating the fact that the tree has been copied down the tree is deferred until some part of the map is modified; at that point, the set operation described above will take care of propagating the change down the tree as far as it needs to go.

While DRCOs address many issues with map efficiency, they can be improved even further by considering how they will be used more carefully. First, it is observed that requests to update cells in a map exhibit spacial clustering. This is because a single sonar ray contains a sequential list of information that falls along a straight path through space. If one cell in the map is updated, it is extremely likely that the next update will be to an adjacent cell. (This is also true for requests to get parts of the map since the sequence of updating a map using sonar information requires the previous contents of each map cell that is to be updated to be factored in). Second, note that traversing an octree is more computationally expensive that simply getting a value from an array.

On the basis of these facts, the octree can be modified so that individual map cells are no longer stored in the map. Instead, a hybrid map that combines array-based maps with an octree can be created by having each leaf in the octree store a smaller instance of the array-based map implementation described in the previous section. These smaller maps are where the actual cells of the map are stored. This allows for caching: the top-level map holds a direct reference to the most recently accessed sub-map. When a request to perform an operation on the map is made (either a get or a set), the request is first checked to see if it is meant to operate on the cached sub-map. If it is, the the request can be satisfied immediately without having to traverse the octree. Thus, the amortized cost of clusters of nearby gets and sets is reduced because the cost of traversing the octree to find the proper sub-map is only paid once per cluster.

As in the previous section concerting array-based maps, a tradeoff is created; this time the tradeoff is between memory flexibility and speed. If the sub-maps are large, then caching will be more effective because more requests will be able to be filled from the cache without having to fetch another sub-map. Other the other hand, larger sub-maps also cause memory to be allocated in larger chunks, which could be wasteful if the sonar data is too sparse. In practice, a sub-map size of about 10 to 20 cells per side seems to work well.

2.4 Fitting Sonar Data to an Occupancy Grid

Using sonar data to update that map involves establishing a model of exactly how the sonar scans capture information about the environment. Sonar measures reflected sound waves, and those waves spread out as they travel further and further from the ROV. However, it would be quite complicated to calculate exactly how this happens based on physical laws. Instead, a simplified model is needed in order to allow the sonar measurements to be easily incorporated into a map.

Recall that each sonar ray *r* contains a set of measurements d_1, d_2, etc. that measure how much of the initial sonar pulse is reflected back from various distances. The simplest model is that each sonar ray doesn't spread out at all, but acts like a laser. Thus, each d_i can be thought of as measuring the reflectance of a specific point in the environment. This point falls inside a single map cell, so only that cell is updated. This model is very simple and allows for data to be added to the map very quickly, but it falls short because it begins to leave progressively larger holes in the map as the locations for which reflectance is measured fall farther away from the ROV. Figure 8 shows an example of how this manifests in a map: streaks of cells are filled in, but large gaps remain between them.

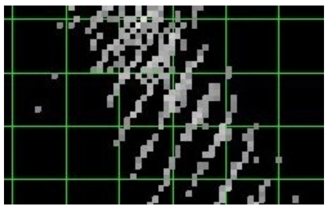


Figure 8: Sonar ray "streaks" created by the simple, laser-like sonar model.

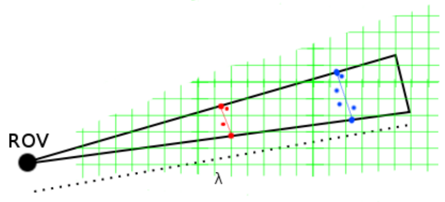


Figure 9: Updating the map using the "slice" sonar model shown at two different points along the sonar slice (one in red, one in blue). λ marks the range of the sonar scan. Grid cells that are updated by each of the measurements are marked using the corresponding color.

A more reasonable model of how the sonar captures information (as described in Faifield et al. (2006)) is that each measurement d_i from a sonar ray r holds information about a conic section of the surrounding environment. Since it is already assumed that the ROV is taking scans that sweep out a two-dimensional plane, this model is further simplified by modeling each sonar ray as measuring reflectance values along a triangular section of that plane (a “slice”).

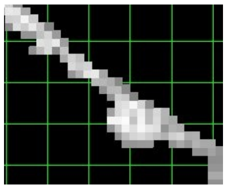


Figure 10: The improved sonar model creates smoother maps.

Using this better model, updating the map becomes more time consuming, but results in much better looking maps, especially when the sonar range is large. Instead of updating a single cell, several cells may need to be updated. For each d_i in r, the algorithm for doing this updating consists of getting the two points on the edges of the sonar slice at the proper distance λ / i along the ray and then using the value of d_i to update each map cell that the line segment connecting those two points intersects. Figure 9 shows a graphical representation of this algorithm, and Figure 10 shows a portion of a map that results from using this improved model.

2.5 Filling in Unknown Map Sections

Since sonar scans only provide information about two-dimensional planes, it is impossible to make a complete three-dimensional map directly. However, several sequential scans can provide a skeleton of the full map. It is then possible to then add extra data to the map in order to fill in the gaps between the scans by “growing” the defined sections of the map towards each other. This process is meant to move from a map that shows only a series of horizontal slices to a map which shows the full walls of the environment being mapped.

The method used to grow walls works by analyzing the existing map layer by layer from bottom to top. Since the map is stored as a grid of cells, this just involves iteratively applying the wall-filling algorithm at each possible depth coordinate. At each level, a three phase algorithm is applied.

The first step is an analysis step which is used to determine where parts of walls already exist in the map. This is accomplished by grouping cells which have a value higher than some

Figure 11: Pseudocode for ***Build_Cluster(Cluster c, Cell cell, Map M, Collection<Cells> used_cells)***

```
//make sure this cell has not been seen already
if used_cells contains cell:
      return;

//make sure this cell is part of a wall
if value of cell < min_cell_value:
      return;

//add this cell to the cluster and recursively look at its neighbors
c.add(cell);
used_cells.add(cell);
for each Cell next in cluster_radius around cell:
      Build_Cluster(c,next,M,used_cells);
```

set threshold into clusters based on their location. Nearby cells which have a high value are clustered together and treated as being a segment of a single wall. Figure 11 shows pseudocode for the algorithm that groups cells into clusters. The algorithm tracks a group of cells (the cluster) and a global set of all the cells that have been grouped into a cluster. For each cell in the map, if the cell has not already been grouped into a cluster and has a sufficiently high cell value, that cell is added to the cluster. If the cell was added, recursively check each other cell that is close to that cell. “Close” is defined to mean within a circle of set radius around the current cell. The radius of the circle can be adjusted to fit the data that is being processed: if the walls of the environment being mapped are very uneven and have holes in them, then the radius can be increased in order to provide a greater tolerance for gaps. On the other hand, if the walls are smooth, then the radius can be decreased to create a more precise grouping of cells, which in turn will produce a more accurate map.

After running the clustering algorithm over the cells within a layer of the map, a series of wall clusters will be produced. The second step of the wall-filling procedure is to create pairings between these clusters of cells in order to define where cells should be updated to create a wall. The pairings should be made such that both clusters are (to a best estimate) part of the same wall.

For each cluster that was found, that cluster is compared to each of the other clusters. For each comparison, a vector *V* is constructed from the center of the first cluster to the center of the second. Each cluster is paired with the cluster that is closest two as measured by the length of *V*. However, this is not sufficient to ensure that appropriate walls are created. Some environments (such as many marine caves) are shaped like narrow gorges. This creates a problem because it will often be the case the the closest cluster is actually part of a wall on the other side of the gorge.

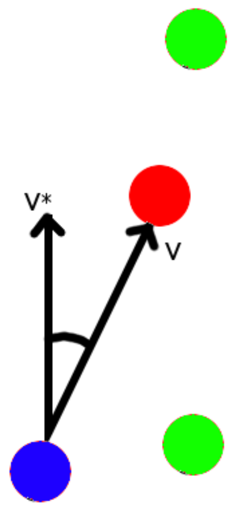


Figure 12: Cluster Matching. The colored regions represent cell clusters; the blue cluster is paired with the red one over either of the green ones because the green clusters are either too far away (the top one) or two out of line (the rightmost one).

In order to prevent walls from being drawn in inappropriate places, a vector *V** is defined in the direction in which the tunnel extends. The knowledge that tunnel walls are most likely going to be approximately parallel to this direction provides the guidance needed to prevent inappropriate cluster pairings. For each possible paring, the angle between the vectors *V* and *V**, that is, the angle given by

$$\cos^{-1}\left(\frac{|V \cdot V^*|}{\|V\|\|V^*\|}\right)$$

is calculated. If this angle is greater than some defined cut-off value, the paring is thrown out because it would create a wall in an inappropriate direction. Figure 12 shows a visual representation of this. As with the clustering radius before, the angle cut-off can be adjusted in order to account for differences in data sets. The map of a very straight and narrow tunnel will benefit from a small cut-off, while a wide tunnel with some turns in it will benefit from a large cut-off.

Finally, once cluster pairings have been created, the third step of the process is run which modifies the cells of the map between the two clusters. This is done by defining a quadrilateral

Figure 13: Pseudocode for ***Grow_Wall(Pairing pair)***

```
//the corner cells from the first cluster
Cell corner1, corner2;
max_angle = 0, min_angle = 0;
for each cell c1 from pair.cluster1:
      angle = angle_between(
            direction from c1 to pair.cluster2,
            direction from pair.cluster1 to pair.cluster2);
      if angle > max_angle:
            corner1 = c1;
      if angle < min_angle:
            corner2 = c1;

//now repeat for the second cluster with respect to the first
Cell corner3, corner4;
max_angle = 0, min_angle = 0;
for each cell c2 from pair.cluster2:
      angle = angle_between(
            direction from c2 to pair.cluster1,
            direction from pair.cluster2 to pair.cluster1);
      if angle > max_angle:
            corner3 = c2;
      if angle < min_angle:
            corner4 = c2;

//now fill in the quadrilateral
for each Cell c inside Quadrilateral(corner1,corner2,corner3,corner4):
      c.value = Interpolate(pair.cluster1,pair.cluster2,c.location);
```

between the two clusters and filling in all of the cells that fall inside of that quadrilateral using some form of interpolation between the cells in the clusters at the end points. The corners of the quadrilateral are determined by finding the maximum width of each cluster as measured along the axis perpendicular to the vector between the two clusters. Once the area to be filled in has been established, each cell inside of it is filled in according to some interpolation scheme. The pseudocode for this process is shown in Figure 13.

The exact method by which interpolation between the two clusters is done can be selected from a variety of different interpolation schemes. The simplest method is to use simple linear interpolation: if the clusters c_1 and c_2 have average cell values of p_{c1} and p_{c2}, then the cell to be filled in can be set to the value given by

$$\frac{p_{c1} * (1-i) + p_{c2} * i}{2}$$

where i is between 0 and 1 and is a linear measure of how close the cell to be filled in is to either cluster with $i = 0$ corresponding to being at cluster c_1 and $i = 1$ corresponding to being at cluster c_2. This interpolation scheme will produce even walls that smoothly blend between the end clusters. However, the map it produces will not necessarily reflect the confidence in the location of the walls. The locations of the walls are not known with absolute confidence, but rather, are a best estimate. Furthermore, the confidence with which a wall's location is known is greater when considering cells that are close to a cluster than when considering cells which are farther out in the space between clusters. For this reason, it is possible to use other interpolation schemes which take the distance to the nearest cluster into account and set cells that are in the middle of the gap to have a lower value that the cells that are near the existing clusters. While the map created with this type of interpolation does not look as smooth or complete as a map made with simple linear interpolation, it will better reflect the confidence with which the locations of the walls are known.

3 Field Work

In order to validate the mapping system developed, actual data had to be collected and fed to the system. Simple pool trials were first conducted in order to test basic system functionality and to determine how accurate the produced maps were. A data collection trip to Malta followed during which data was collected from a sea cave.

3.1 Equipment and Setup

Experiments were conducted using a VideoRay Pro III ROV (pictured in Figure 15) equipped with a Tritech scanning sonar system with an effective range of up to 80 meters. This ROV also houses a depth sensor that provides the depth of the ROV in 0.1 meter increments and a compass which gives the orientation of the ROV in degrees (with 0 corresponding to north). The ROV is tethered to and operated from a base station (typically on a boat) using a control box that allows the user to drive the ROV using a joystick an several control knobs (pictured in Figure 16). Sonar data is passed through the control

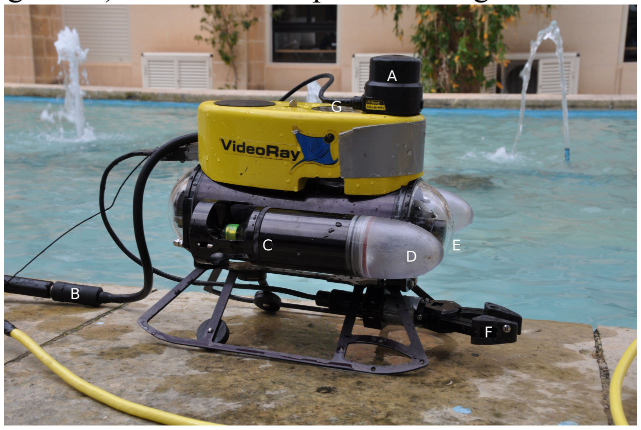
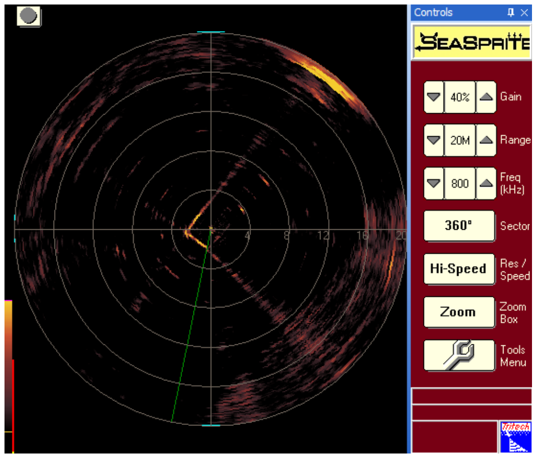


Figure 15: The Video Ray Pro III, a small ROV used for this project. Labeled parts: (A) sonar head, (B) tether, (C) forward thruster, (D) headlight, (E) video camera, (F) gripper for sample collection, (G) vertical thruster



Gain 40%
Range 20M
Freq (kHz) 800
Sector 360°
Res / Speed Hi-Speed

Figure 14: SeaSprite sonar software user interface. A sharp corner (part of the swimming pool) is visible in the sonar scan shown.

box to a laptop computer which uses Tritech's SeaSprite software to display and log sonar measurements (see Figure 14).

The ROV is connected to the control box using a smart tether from KCF Technologies. This tether powers the ROV, sends control signals to it, and transports video, sonar, and other data back from the ROV. Additionally, the smart tether is fitted with a series of six sensors along its length that are able to report the position of the ROV relative to the base station from which the operator is controlling the ROV. KCF's website (kcftech.com) reports that the smart tether is able to provide location data that is accurate to 1.5 meters at a rate of approximately 5 Hz.

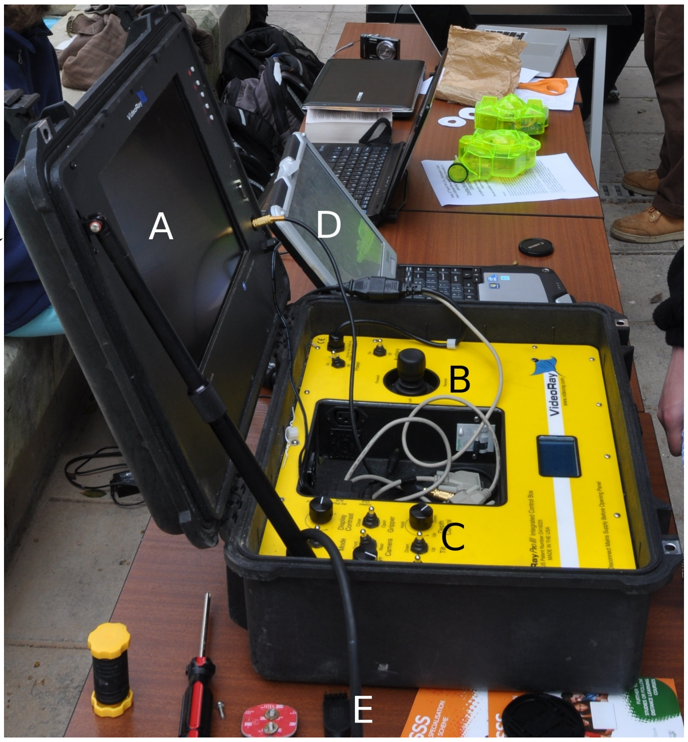


Figure 16: Video Ray Control Box. Labeled parts: (A) primary camera display, (B) directional control joystick, (C) depth and auxiliary controls, (D) laptop running sonar, video, and smart tether data capture software, (E) tether to ROV.

Finally, a USB GPS receiver is attached to the base station in order to provide an absolute measurement of the base station location. The GPS location provided by this receiver, along with the data from the smart tether, is intended to be enough information to provide the absolute GPS coordinates of the ROV. This is accomplished using KCF's proprietary smart tether software which runs on the base station laptop.

The smart tether is approximately 30 meters long, which is not enough to allow exploration of a large cave. To solve this, an 80 meter long extension tether was used. This tether is placed between the main smart tether and the ROV control box. Unlike the main smart tether, the extension is not fitted with location sensors.

3.2 Data Collection Technique

Two different data collection methods were employed. First, the sonar head was attached to the ROV so that it scanned in a plane parallel with the water's surface (“horizontal scanning”). A series of overlapping scans were then taken by moving the ROV through the environment and either landing it at various locations or carefully hovering in one position. At each scan location, sonar data was collected for at least one full rotation of the sonar head. The depth, compass reading, and approximate location of each scan were recorded and matched up with the sonar data.

Next, the sonar head was rotated so that it collected data from a plane perpendicular to the water's surface (“vertical scanning”), and a second set of scans were taken. In this case, the scans do not overlap like the horizontal ones but are instead arranged to get a full series of vertical sections of the site being mapped. These scans were taken with the ROV aligned to approximately the same compass bearing for each scan so that the scans will be as well suited to the wall-filling algorithm described in section 2.5. As with the horizontal scans, the depth, compass reading and approximate scan location were recorded for each scan.

During each of these scanning phases, the smart tether was enabled in order to provide precise information about the location of the ROV during each scan. Both the scans and the smart tether location were recorded using the same laptop computer to ensure that the the two types of data can be temporally matched.

3.3 Sites Visited

3.3.1 Pool Trials

In order to get some initial data to help evaluate and improve the mapping system, and to ensure that hardware components were functioning well, the ROV was used to gather data from DeNunzio Pool at Princeton University in late 2011. The pool is a standard Olympic sized pool (25 meters by 50 metes) and is 5.3 meters deep in the area scanned and is lined with ceramic tile (see Figure 17). Scans were taken at regular intervals (approximately 2.7 m apart) along the deep end of the pool with the ROV stationary on the bottom. 8 horizontal scans and 7 vertical scans were taken.

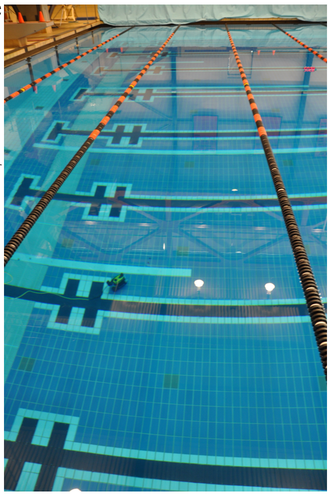


Figure 17: Photograph of the deep end of the pool at Princeton. The ROV is visible at the bottom as it moves between scan locations.

Unfortunately, the smart tether was not available for use during pool trials. However, since the pool is a relatively confined space with a simple geometry, mock smart tether data was generated in order to mimic the kind of data that the smart tether would normally provide. This ensured the sonar scans could be properly positioned relative to each other.

3.3.2 Gozo Sea Cave

The primary data collection for this project occurred in March of 2012 during an expedition to the Maltese islands. The sea cave visited is located within a cluster of similar caves on the north side of Gozo, the northernmost and second largest island in Malta. Figure 18 shows an image of the mouth of the cave as viewed from outside. The cave is carved out of the side the steep cliffs that characterize the shoreline in the area and is approximately 20 meters deep and extends into the island.

Below the surface, the cave is shaped like a canyon; it is wide at the top and tappers down towards the bottom. The cave is divided vertically into two parts; the first extends from the bottom about one third of the way to the surface, is fairly narrow, and has steep walls. The second half (the top half) is a bit wider and has walls which slope downward from the surface. The bottom of the cave is covered with debris mostly consisting of broken off chunks of rock with some coral.

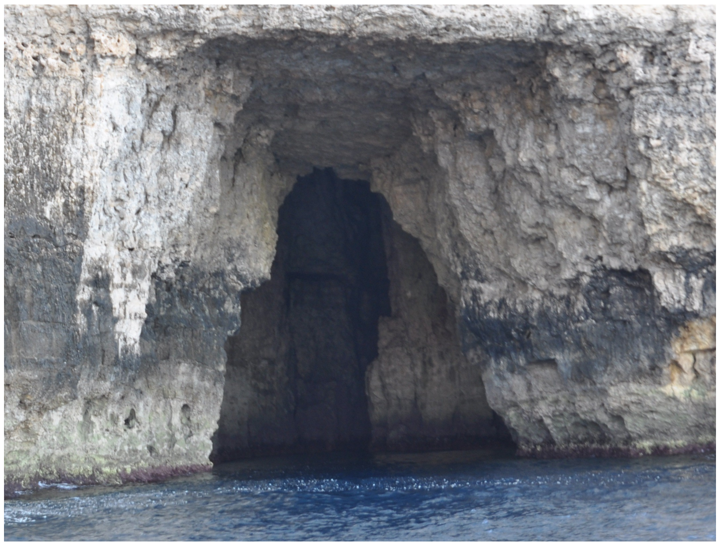


Figure 18: Above water view of the mouth of the Gozo sea cave.

The ROV was piloted from the cabin of the *Isis*, an 8.3 meter long research boat provided for this use by the Aurora Special Purpose Trust (a marine archeology organization). 4 sets of scans were taken: two sets of horizontal scans and two sets of vertical scans. The first set of horizontal scans was taken with the ROV landed on the bottom of the cave; the second set was taken while hovering at a depth of between 5 and 10 meters. The vertical scans were taken at a variety of depths, but all were taken with the ROV facing approximately along the length of the cave.

4 Results

4.1 Pool Trials

Data acquired from the pool trials at Princeton revealed several interesting things about the nature of the sonar data. One thing that was evident was the fact that the sonar was actually reflecting off the walls, hitting other walls, and then being reflected back along the same path. This creates false "echo" walls that were not actually there. This effect is visible in Figure 19, which is from a two-dimensional map of one end of the pool.

Additionally, sonar data from the pool trials exhibited a certain noise pattern. Generally, sonar data was extremely noisy close to the ROV, had little noise at medium ranges, and had some noise at long ranges (though not as much as the measurements that were very close to the ROV). Figure 20 shows a plot of the noise in sonar measurements from one of the horizontal scans. The graph shows the average of the absolute value of the difference between successive reflectance measurement from sonar rays as a function of the distance from the ROV. Note that the spikes in the center of the graph corresponds to a pool walls, so the data at that point is expected to exhibit large variance. Following pool trials, the system was updated to

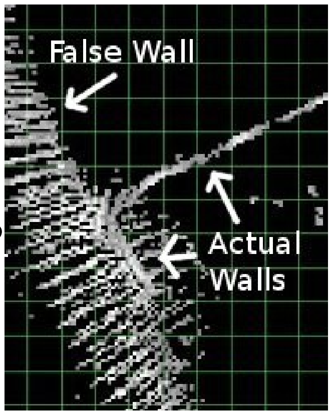


Figure 19: Example of a false wall created from sonar bouncing off the smooth tile in the swimming pool.

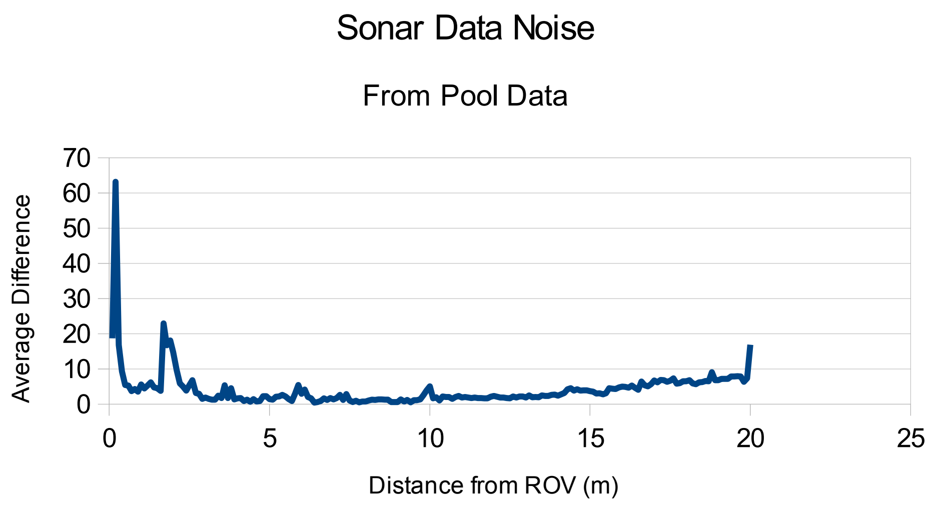


Figure 20: Data noise from sonar measurements during pool trials. Noise is calculated as the average difference between successive reflectance measurements in sonar rays. Note the extremely large peak near 0 and the more gradual increase between 15 and 20 meters.

incorporate this expected noise into the sonar model. The main change was that sonar data that was too close to the ROV was ignored entirely.

4.2 Gozo Sea Cave

Data from the sea cave was not nearly as nice as the data from the pool, but this makes sense given that the sea cave is a much less confined environment. The major difference between the sea cave data and the pool data was not the quality of the sonar data, but rather, the quality of the localization data (the data which provides estimates of the state of the ROV). In the pool, a very accurate location could be assigned for each sonar scan, but in the cave, the mapping system had to rely on the data provided by the smart tether.

The position data collected from the smart tether was usable, but suffered from some problems. First, tether locations were less accurate due to the fact that the tether was basing its location estimate off the location of the mapping base station, which was a boat in this case. Since the boat was able to drift around freely, the location of the ROV as reported by the smart tether was affected. As mentioned in the equipment and setup section, a GPS receiver was mounted on top of the boat in order to track the boat's location and ensure that coordinates and modify the calculated location of the ROV appropriately. This was successful for some of the scans, but at other times, the GPS receiver on the boat was not able to get a good signal.

Second, the smart tether's location estimates grew much less accurate as the ROV moved further into the cave and more tether had to be let out in order to allow the ROV to continue. Since the tether extension is not fitted with the same kind of location sensors as the main tether and because the location estimate had to be made over longer distances, the reported location of the ROV grew less accurate as the ROV moved further into the cave.

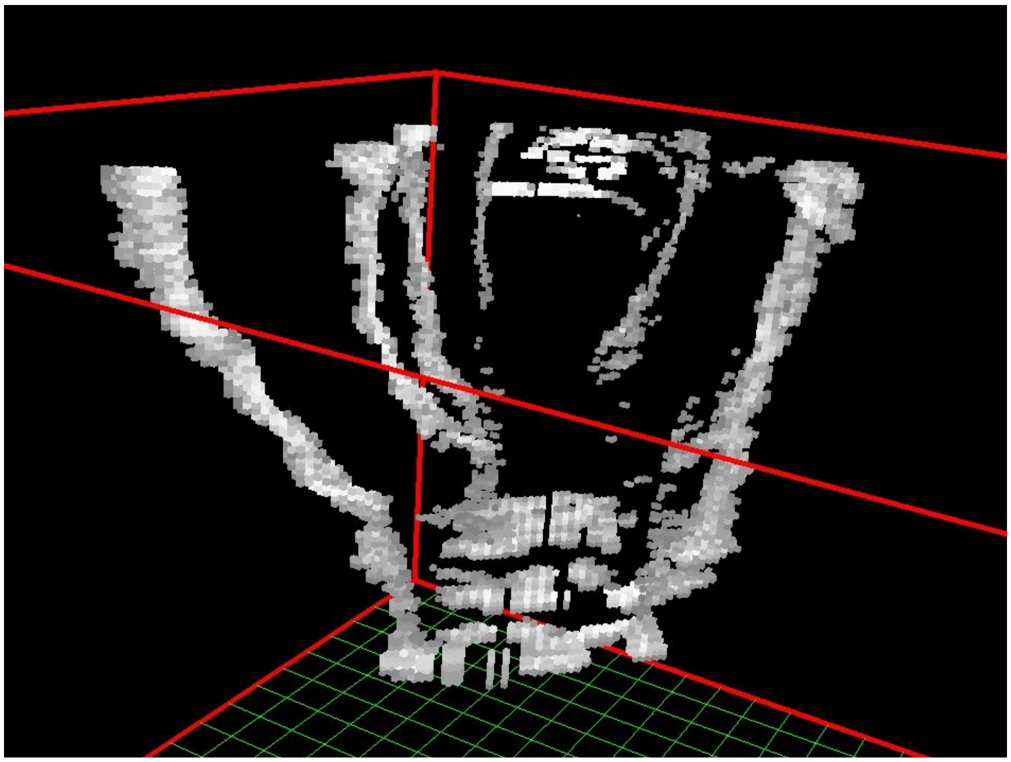


Figure 21: Image of a map created from several scans from the mouth of the sea cave. This is how the map looks before missing walls are filled in.

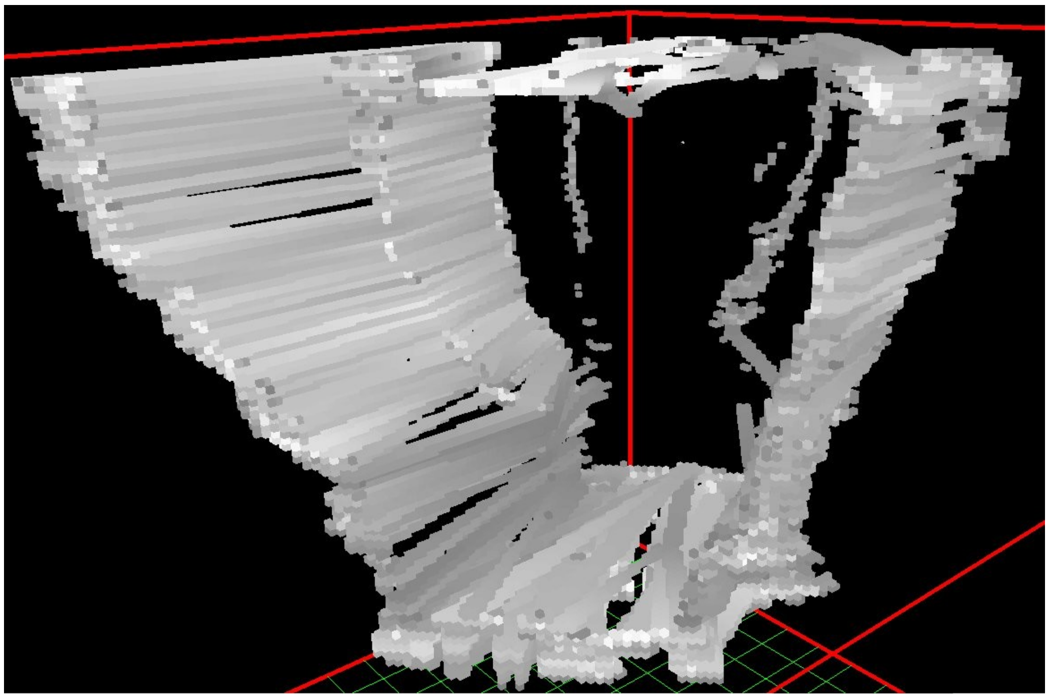


Figure 22: A map of the mouth of the cave with walls filled in.

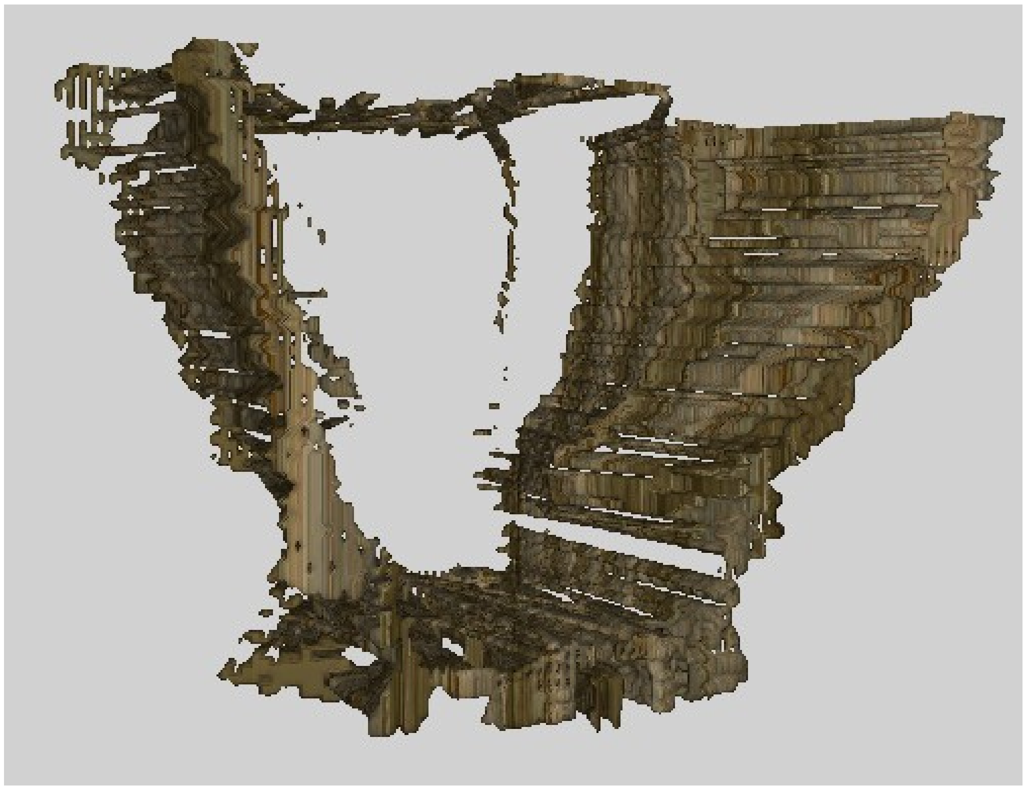


Figure 23: A textured visualization of the mouth of map generated using scan data from the mouth of the sea cave. Photo curtsey of Jeffrey Forrester at Cal Poly who developed software to create such visualizations from map data.

However, despite these problems, a good amount of usable data was collected and used to map the mouth of the cave. A map of the mouth of the sea cave is shown in Figure 21. The same map with walls filled in (using a linear interpolation scheme) is shown in Figure 22. A visualization of this data in which walls have been colored using a rock-like texture is shown in Figure 23. This map was made using a series of scans from the second set of vertical scans. The shape of the walls and floor of the cave are clearly visible.

In addition to the map, the data from the sea cave was used to measure the memory and computational efficiency of the mapping system. In order to get an idea of how the map representations might be used, a mock particle filter was implemented within the mapping system. This mock filter does not actually perform any filtering, but just goes through the process of creating a deleting multiple maps and updating each of those maps independently as though they were being used as maps for particles. While each of the mock particles has the same ROV state and thus doesn't accomplish any mapping goals, this is still a sufficient mechanism to test how well different mapping representations perform when under heavy use.

Figure 24 shows memory usage and timing statistics for a trial run of the system using one full set of vertical scans. As the table shows, the octree implementation performed quite well compared to a simple array-based map when more that just one particle was being simulated. Attempts to use a much larger number of particles (eg 100) resulted in the array-based map implementation running out of memory.

Map Type	**Hybrid Octree**	**Array-based**
Memory usage with 1 map	24.7%	16.4%
Mapping time with 1 map	45 seconds	30 seconds
Memory usage with 10 maps	29.9%	54.2%
Mapping time with 10 maps	210 seconds	230 seconds

Figure 24: Performance of different map data structures. Test were performed on a system using a dual core 2 GHz processor with 2GBs of system memory. Memory usage refers to the maximum percent of total system memory in use during mapping.

5 Conclusion and Possible Future Work

In this project, a system was developed that allowed for the use of a small sonar-equipped ROV to create full three-dimensional maps of underwater environments using only sparse two-dimensional sonar scans. The system uses an occupancy grid approach to represent a map of an environment, and employs an octree data structure to allow for much larger maps to be created than were previously possible. The system is also able to fill in missing data between scans by filling in walls that are likely to be present between actual sonar data clusters. After pool trials, a partial map of a sea cave in Malta was successfully created using this system following a data gathering expedition to the Maltese islands. This project has demonstrated that large-scale three-dimensional mapping using a small ROV is a feasible task.

In the future, work done in this project could be combined with an effective particle filter implementation to alleviate the issues with the quality of the ROV localization identified in the results section. However, such a particle filter will have to cope with the fact that using two-dimensional scanning sonar to create three-dimension maps leads to very sparse maps that offer less overlap between sonar scans that is typical for two-dimensional mapping.

Much more work can be done to explore different possible methods of filling in missing walls. In this project, wall filling depends on the user defining the direction in which the tunnel being mapped extends; future systems could investigate determine this direction automatically. Additionally, the wall-filling system does not work well for tunnels that are not generally straight; future work could be investigate finding ways to adapt to trends in the sonar data in order to better fill in missing map sections. Finally, future work could investigate systems of filling in walls that are truly three-dimensional and take into account information that is at above and below the current level being filled.

Acknowledgments

This work was made possible thanks to funds provided by Princeton University and by the International Computer Engineering Experience program at Cal Poly. Travel to Malta was made possible by Prof. Timmy Gambin of the University of Malta and the Aurora Special Purpose Trust. Use of the smart tether was made possible by KCF Technologies. The author would like to thank Prof. Christopher Clark for his excellent advising during this project, Andrew Boik for contributing to discussions about many aspects of this work, Billy McVicker for advise on and contributions to software development, Jeffrey Forrester for providing the textured visualization of the cave map, and all of the Princeton and Cal Poly students and faculty who participated in the 2012 Malta research trip for their encouragement and great company.

References

White, C., D. Hiranandani, C. Olstad, K. Buhagiar, T. Gambin, C. M. Clark. The Malta Cistern Mapping Project: Underwater Robot Mapping and Localization within Ancient Tunnel Systems. Journal of Field Robotics, 2010.

Fairfield, N., G. Kantor, D. Wettergreen. Real-Time SLAM with Octree Evidence Grids for Exploration in Underwater Tunnels. 29 November, 2006.

Fairfield, N., G. Kantor, D. Wettergreen. Three Dimensional Evidence Grids for SLAM in Complex Underwater Environments. 2005.

Fairfield, N., G. Kantor, D. Jonak, DD Wettergreen. DEPTHX Autonomy Software: Design and Field Results. July 2008.

Clark, C. M., C. S. Olstad, K. Buhagair, T. Gambin. Archaeology via Underwater Robots: Mapping and Localization within Maltese Cistern Systems. ICARV Proceedings, 2008.

Williams, S. B., P. Newman, G. Dissanayakc, H. Durrant-Whyte. Autonomous Underwater Simultaneous Localisation and Map Building. Proceedings of IEEE International Conference on Robotics and Automation, 2000.

Chen, Z. Bayesian Filtering: From Kalman Filters to Parricle Filters, and Beyond. 2003.

Tardós, J. D., J. Neira, P. M. Newman, J. J. Leonard. Robust Mapping and Localization in Indoor Environments Using Sonar Data. The International Journal of Robotics Research, 2002.

Martin, M. C., H. P. Moravec. Robot Evidence Grids. Technical Report CMU-RI-TR-96-06, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.