Fast Approximate 2D Ray Casting for Accelerated Localization

Corey H. Walsh[1] and Sertac Karaman[2]

***Abstract*— Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.**

I. Introduction

Determining a robot's location and orientation in a known environment, also known as localization, is an important and challenging problem in the field of robotics. Particle filters are a popular class of Monte Carlo algorithms used to track the pose of mobile robots by iteratively refining a set of pose hypotheses called particles. After determining an initial set of particles, the particle filter updates the position and orientation of each particle by applying a movement model based on available odometry data. Next, the belief in each particle is updated by comparing sensor readings to a map of the environment. Finally, the particles are resampled according to the belief distribution and algorithm repeats.

While particle filters provide a robust and general framework for solving the localization problem, they can be computationally expensive due to both the number of particles which must be maintained and the evaluation of the sensor model. In robots with range sensors such as LiDAR or Sonar, ray casting is frequently used to compare sensor readings with the ground truth distance between the hypothesis pose and obstacles in a map. Ray casting itself is a complex operation, and a single evaluation of the sensor model may require tens of ray casts. Many effective particle filters maintain thousands of particles, updating each particle tens of times per second. As a result, millions of ray cast operations may be resolved per second, posing a significant computational challenge for resource constrained systems.

Several well known algorithms exist for ray casting in two dimensional spaces such as Bresenham's Line algorithm [1] and ray marching [2]. Both algorithms work by iteratively checking points in the map starting at the query point and moving in the ray direction until an obstacle is discovered. This process may require hundreds of memory reads per ray cast depending on the distance to the nearest obstacle, and does not provide constant time performance.

To combat the computational challenges of ray casting while localizing in a two-dimensional map, Thrun et al. [3] suggest the use of a large three-dimensional lookup table (LUT) to store expected ranges for each discrete (x, y, θ) state. While this is simple to implement and does result in large speed improvements as compared to ray casting, it can be prohibitively memory intensive for large maps and/or resource constrained systems. In a 2000 by 2000 occupancy map, storing ranges for 200 discrete directions would require over 1.5GB. While this memory requirement may be acceptable in many cases, it scales with the square of map size - a 4000 by 4000 map would require over 6GB for the same angular discretization.

We propose an analogous data structure to the three-dimensional LUT called a Compressed Directional Distance Transform (CDDT) which requires significantly less memory and precomputation time while still allowing near constant time ray casting queries. The contributions of this work are the following. Firstly, we develop a problem formulation and provide details and analysis of our method. Secondly, we implement the proposed algorithm and compare its performance to alternate two-dimensional ray casting methods. Finally, we discuss potential extensions to the CDDT algorithm which could be the basis of future research. We observe two orders of magnitude improvement in memory requirements, with little sacrifice in computation time, when compared to the lookup table methods. Additionally, we observe a large speedup when compared to the other ray casting methods considered with similar memory requirements.

II. Related Work

Bresenham's line algorithm [1] incrementally determines the set of pixels that approximate the trajectory of query ray starting from the query point $(x, y)_{query}$ and progressing in the θ_{query} direction one pixel at a time. The algorithm terminates once the nearest occupied pixel is discovered, and the euclidean distance between that occupied pixel and $(x, y)_{query}$ is reported. This algorithm is widely implemented in particle filters due to its simplicity and ability to operate on a dynamic map. The primary disadvantage is that it is slow, potentially requiring hundreds of memory accesses for a single ray cast. While actual performance is highly environment dependent, Bresenhams Line algorithm is linear in map size in the worst case.

Similar to Bresenham's Line algorithm, ray marching [2] checks points along the line radiating outwards from the query point until an obstacle is encountered. The primary

[1]Corey H. Walsh is with the Department of Computer Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, USA `chwalsh@mit.edu`

[2]Sertac Karaman is with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139, USA `sertac@mit.edu`

difference is that ray marching makes larger steps along the query ray, thereby avoiding unnecessary memory reads. Beginning at $(x, y)_{query}$ the ray marching algorithm proceeds in the θ_{query} direction, stepping along the line by the minimum distance between each query point and the nearest obstacle. The algorithm terminates when the query point coincides with an obstacle in the map. A precomputed euclidean distance transform of the occupancy map provides the distance between each query point and the nearest obstacle in any direction.

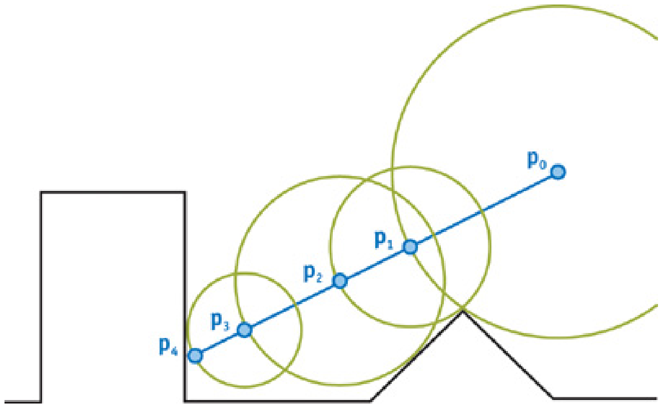


Fig. 1. Visualization of ray marching starting at p_0 towards p_4. Green circle around each query point represents the distance to the nearest obstacle from that point. Blue dots represent the query points, labeled in order of execution. From [5].

Ray marching is on average much faster than Bresenhams line, but edge cases exist in which the theoretically asymptotic runtime is equivalent. As noted in [4] the traversal speed of rays rapidly decreases as sampled positions approach obstacles. For this reason, rays which travel parallel to walls progress very slowly as compared to those passing through open areas. Thus the performance of ray marching exhibits a long tail distribution as seen in (Fig. 9, 10) which negatively impacts average runtime as compared to the median, and can be problematic for near real time algorithms.

As previously described, a common acceleration technique for two dimensional ray casting is to precompute the ray distances for every state in a discrete grid and store the results in a three dimensional LUT. Any ray casting technique may be used to compute the table. In the authors implementation ray casting is used to populate the LUT since it has low initialization time and is significantly faster than Bresenhams Line. Theoretically the LUT approach has constant query runtime, though actual performance is dependent on access patterns as CPU caching effects are significant in practice.

State space discretization implies approximate results, since intermediate states must be rounded onto the discrete grid. While the effect of rounding query position is small, rounding θ_{query} may have significant effects. As the ray moves further away from the query point along the theta direction, angular discretization error accumulates. For queries $(x, y, \theta)_{query}$ discretized into $\lfloor(x, y, \theta)\rceil$, the distance between the end of the ray $\lfloor(x, y, \theta)\rceil$ and its projection onto the line implied by $(x, y, \theta)_{query}$ becomes large as the length of the ray increases. Although the error induced by discretization may be unacceptable for computer graphics applications, it is generally acceptable for probabilistic algorithms such as the particle filter since error is expected and directly modeled.

More generally, due to varied handling of edge cases and ambiguities in the problem of determining distance between any point and the nearest obstacle in a particular direction, most 2D ray casting algorithms do not provide exactly consistent results for every query. However, on average error between results from each of the methods discussed is small. Any error that does exist is dominated by error inherent in distance sensors.

Rather than simply rounding to the nearest grid state, one may improve accuracy of queries which lie between discrete states by querying the neighboring states and interpolating the results. Since we are evaluating the performance of these methods for use in a particle filter, we do not perform interpolation since the increased computation reduces the number of particles that may be maintained in real time, thereby effectively negating any potential increase in accuracy with respect to the stored map.

III. Problem Formulation and Notation

We define the problem of ray casting in occupancy grids as follows. We assume a known occupancy grid map in which occupied cells have value 1, and unoccupied cells have value 0. Given a query pose $(x, y, \theta)_{query}$ in map space, the ray cast operation finds the nearest occupied pixel $(x, y)_{collide}$ which lies on the ray starting at the position $(x, y)_{query}$ pointing in the θ_{query} direction, and returns the euclidean distance d_{ray} between $(x, y)_{query}$ and $(x, y)_{collide}$.

$$d_{ray} = \left\| \begin{pmatrix} x \\ y \end{pmatrix}_{query} - \begin{pmatrix} x \\ y \end{pmatrix}_{collide} \right\|_2$$

We refer to $(x, y, \theta)_{query}$ as the query pose, and $(x, y)_{query}$ as the query point. We denote the discretized query pose as $\lfloor(x, y, \theta)_{query}\rceil$. A θ slice through the LUT refers to all table entries for that particular $\lfloor\theta\rceil$. The number of discrete $\lfloor\theta\rceil$ values is denoted $\theta_{discretization}$. Fig. 2 demonstrates our chosen coordinate system.

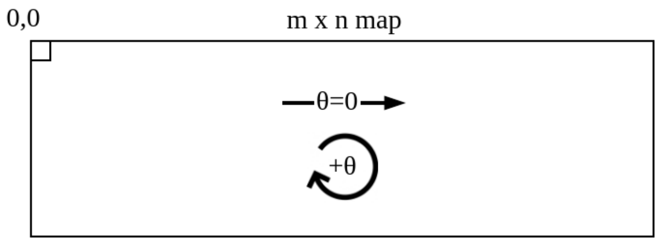
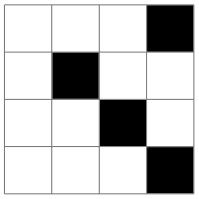


Fig. 2. Occupancy grid map coordinate system

IV. The Compressed Directional Distance Transform Algorithm

Although the three dimensional table used to store precomputed ray cast solutions for a discrete state space is inherently large, it is highly compressible. This is most apparent in the cardinal directions, in which adjacent values along a particular dimension of the table increase by exactly one unit of distance for unobstructed positions as in Fig. 3. Our data structure is designed to compress this redundancy while still allowing for fast queries in near constant time. We accomplish this though through what we refer to as a Compressed Directional Distance Transform described here.



3	2	1	0
1	0	-	-
2	1	0	-
3	2	1	0

Fig. 3. Occupancy grid (left) and associated LUT for $\theta = 0$ (right).

The euclidean distance transform of an occupancy grid map stores the distance to the nearest obstacle in any direction θ for each possible $\lfloor(x,y)\rceil$ in the map. In contrast, a directional distance transform (DDT) stores the distance to the nearest obstacle in a particular direction $\lfloor\theta\rceil$ for every $\lfloor(x,y)\rceil$. The key difference between a single θ slice of the LUT and a directional distance transform for the same θ is the way in which it is computed and indexed. To compute the LUT slice, ray casting is performed in the θ direction for every $\lfloor(x,y)\rceil$. At runtime, each $(x,y)_{query}$ is discretized to $\lfloor(x,y)_{query}\rceil$ and the LUT slice is directly indexed.

In contrast, to compute the DDT, the obstacles in the map are rotated about the origin by $-\lfloor\theta\rceil$ and ray casting is implicitly performed in the $\theta = 0$ direction, as demonstrated by Fig. 4. Then, to index the table for a query $(x,y,\theta)_{query}$ one discretizes to $\lfloor(x,y,\theta)_{query}\rceil$, rotates $\lfloor(x,y)_{query}\rceil$ about the origin, and uses the rotated coordinates $\lfloor(x,y,\theta)_{query}\rceil_{rot}$ to (implicitly) index into the DDT. Thus, roughly the same operation is computed in both the DDT and the LUT, but while one changes the ray cast direction to populate the LUT, one rotates scene geometry and ray casts in a constant direction to populate the DDT.

We define P_{DDT_θ} as the three by three transformation matrix which projects vectors into the coordinate space of the DDT in the θ direction. While it is true that the transformation of the scene geometry introduces small errors as compared to ray casting for each cell, as previously discussed, small errors do not significantly impact particle filter localization performance.

The distinction between the LUT slice and the DDT may be subtle, but it has an important effect. Since ray casting is always performed in the $\theta = 0$ direction to populate the DDT, all values in the same row of the DDT either increase by one unit with respect to their neighbor in the $\theta = 0$

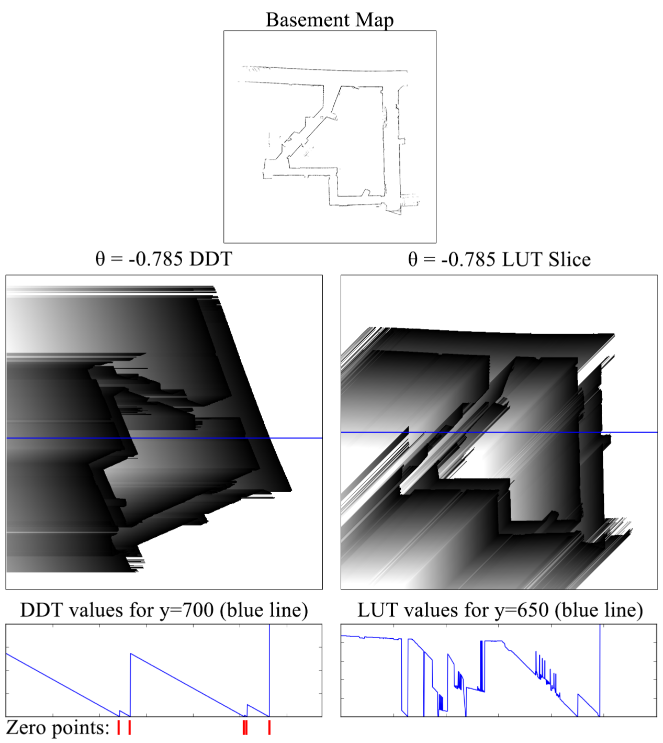


Fig. 4. Comparison between a DDT and a LUT slice for the same value of theta. Each row in the DDT is characterized by a sawtooth function.

direction or go to zero. Thus each row of the DDT may be characterized as a sawtooth function where the zero points correspond to obstacles in the map. This characterization as a sawtooth function provides a natural method of lossless compression: keep the zero points and discard the rest.

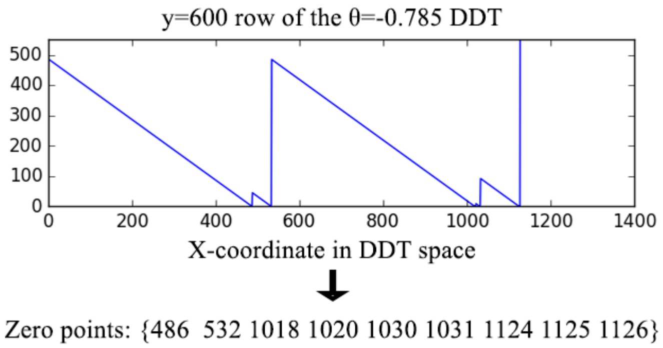


Fig. 5. Demonstration of compression from a sawtooth function to a list of zero points. Consecutively valued zero points exist where walls are greater than one pixel thick.

The conversion from a DDT to a CDDT slice is performed by storing the x coordinate of each zero point in the coordinate space of the DDT for every discrete row as shown in Fig. 5. At query time, the sawtooth function (i.e. exactly the distance to the nearest obstacle in the query direction) encoded in the DDT may be quickly recovered by performing binary search for nearest zero point in the correct row of the CDDT slice. We refer to the list of zero points for a single row as a CDDT bin. Similar to the LUT, the full CDDT is defined as every CDDT slice for all discrete values of θ.

For performance, it is not necessary to compute the full DDT, but rather the CDDT may be directly constructed by projecting each obstacle into the coordinate space of the DDT for every $\lfloor\theta\rceil$ and sorting its x coordinate in the CDDT bin corresponding to its y coordinate and $\lfloor\theta\rceil$. After all geometry has been projected into the CDDT, each bin is sorted to facilitate later indexing. Not only does the direct construction of the CDDT greatly reduce the amount of memory required to store the LUT, it also reduces data structure precomputation time since ray casting is not necessary.

While conceptually simple, implementing the CDDT data structure construction and traversal routines requires careful consideration in order to capture all edge cases and to minimize unnecessary computation. In this sense, it is more complex to implement than the alternatives considered, however there are many opportunities for optimization which yield real-world speed up. To ease this burden, we provide our implementation as well as Python wrappers with an Apache 2.0 license.

A. CDDT Construction Algorithm

```
$edge\_map \leftarrow map - morphological\_erosion(map)$
Initialize $\theta_{discretization}$ empty CDDT slices
**for** $\theta \in \{\lfloor\theta\rceil\}$ **do**
    **for** each occupied pixel $(x, y) \in edge\_map$ **do**
        $(x, y)_{DDT_\theta} = P_{DDT_\theta} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$
        **for** each CDDT bin overlapping with $y_{DDT_\theta}$ **do**
            bin.append($x_{DDT_\theta}$)
        **end for**
    **end for**
    **for** each CDDT bin **do**
        bin = sort(bin)
    **end for**
**end for**
```

B. CDDT Query Algorithm

```
**function ray_cast(**$(x, y, \theta)_q$**)**
    $(x, y)_{DDT_{\theta_q}} = P_{DDT_{\theta_q}} * \begin{pmatrix} x_q \\ y_q \\ 1 \end{pmatrix}$
    bin $\leftarrow$ zero points in row $y_{DDT_{\theta_q}}$ of CDDT slice $\theta_q$
    $x_{collide}$ = smallest element $x_{collide} > x_{DDT_{\theta_q}} \in$ bin
    **return** abs($x_{DDT_{\theta_q}} - x_{collide}$)
**end function**
```

In our implementation we switch between linear and binary search to find $x_{collide}$ in a given CDDT bin depending on the number of zero points in that bin. While binary search is theoretically optimal for sorted arrays, in practice linear search has a superior memory access pattern and is therefore faster in small searches.

C. Further Optimizations

A side effect of extracting zero points from each row of the DDT is the introduction of rotational symmetry. Not only can a row of the DDT in the θ direction be reconstructed from the zero points, but also a row of the DDT in the $\theta + \pi$ direction as in Fig. 6. Therefore, one only need compute CDDT slices for $0 \leq \lfloor\theta\rceil < \pi$ and the DDT for $0 \leq \lfloor\theta\rceil < 2\pi$ may be inferred, resulting in a factor of further two reduction in memory usage.

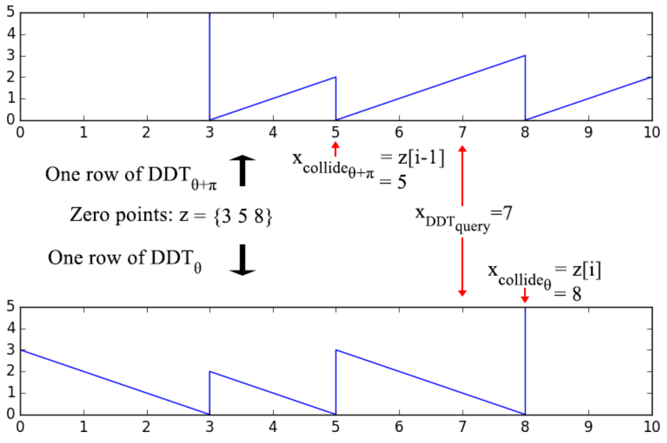


Fig. 6. Demonstration of reconstructing rows of two DDT slices from a single set of zero points.

In addition to the aforementioned reduction in memory usage, rotational symmetry may be exploited in scenarios where ray casts are performed radially around a single point. While traversing the data structure to resolve a ray cast query (x, y, θ), the ray cast for $(x, y, \theta + \pi)$ may be resolved with a single additional memory read. Once a search algorithm is used to discover the index i of $x_{collide_\theta}$ in the CDDT, the index of $x_{collide_{\theta+\pi}}$ is simply $i - 1$ as in Fig. 6. For example, this symmetry can be used in robots with laser scanners sweeping angles larger than 180° to reduce the number of data structure traversals required to compute the sensor model by up to a factor of two.

While the authors did not implement the necessary changes, we conjecture that the CDDT algorithm could be modified to allow incremental updates to the map. In both ray marching and the LUT approach, any change in the map introduces inconsistencies to the acceleration data structure requiring a full re-initialization. As demonstrated by our experiments, initializing the three dimensional LUT is a highly computational process. Similarly, computing a euclidean distance transform for ray marching can be expensive. Therefore neither ray casting nor the LUT can be used with dynamic maps such as in SLAM algorithms. Despite being the slowest option evaluated, Bresenham's Line algorithm is one of the most commonly used techniques since it operates directly on the grid map without an acceleration data structure.

To allow incremental update of the CDDT, the sorted vector data structure used in our implementation to store zero points should be replaced with an alternative which allows for $O(\log n)$ insertion and deletion, such as a randomized skip list. To aid in zero point deletion, pointers could be maintained correlating each cell in the occupancy grid with its associated zero points for each CDDT slice with a constant factor increase in memory usage. In this case, the cost of toggling the state of a cell in the occupancy grid would be $O(\theta_{discretization} \log n)$ where n is the number of elements in each associated CDDT bin. For most occupancy grid maps of real environments, the value of n is small, and in any case is bounded by the size of the map, so $\log n$ is bounded by a small constant factor. Therefore, the cost of update with becomes $O(\theta_{discretization})$ which is likely not prohibitive for real-time performance in dynamic maps.

Skip list traversal is likely to be less efficient than binary search on sorted vectors due to worse cache characteristics. However, the authors expect it to still be faster than performing Bresenhams Line algorithm, and therefore recommend that this modification be the subject of future research.

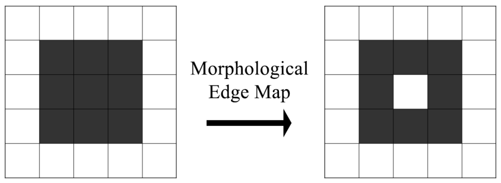


Fig. 7. Example map left, morphological edge map right.

By removing entries in the CDDT which can never possibly result in a ray collision, it is possible to further reduce the memory footprint of the data structure. Consider a 3x3 block of obstacles in an otherwise empty map as in Fig. 7. The center obstacle will never be the nearest neighbor in any ray casting query, because any such query would first intersect with one of the 8 surrounding obstacles. To exploit this, one can take the morphological edge map of the occupancy grid map prior to CDDT generation without loss of generality. Prior to performing ray casting on the CDDT data structure, we check the occupancy grid to ensure that the query point itself is not an obstacle and return a distance of zero if so in order to prevent incorrect results when ray casting from the middle of obstacles removed during the morphological operation. In dense maps, this results in a significant reduction in both memory usage and construction time.

Additionally, consider a line of obstacles aligned along the X-axis. Every element in this line will be projected into a single zero point bin in the $\theta = 0$ CDDT slice. However, the middle elements of the line will never result in a collision. Any ray cast from points on the line of obstacles will return early in the occupancy grid check, and any ray cast from non-overlapping points co-linear in the θ or $\theta + \pi$ directions will intersect one of the edge-most obstacles in the line. Therefore in the $\theta = 0$ CDDT slice, one may discard the zero points corresponding to the middle elements without introducing error. This form of optimization is simple to compute in the cardinal directions, but non-trivial for arbitrary $\lfloor \theta \rceil$ not aligned with an axis. Rather than analytically determining which obstacles may be discarded, it is simpler to prune the data structure by ray casting from every possible state, discarding any unused zero point.

Pruning does increase pre-computation time, rendering it incompatible with incremental update. However, the reduction of memory usage is worthwhile for static maps. In addition to memory reduction, we find that pruning slightly improves runtime performance, likely as a result of improved caching characteristics and the reduced number of zero points.

V. Analysis

The memory usage of the CDDT data structure is $O(n * \theta_{discretization} + |map|)$ where n is the number of occupied pixels in the edge map. The occupancy grid map must be stored to check for overlaps between $(x, y)_{query}$ and obstacles prior to searching CDDT bins. For each edge pixel, a total of $\theta_{discretization}$ float values are stored in the CDDT bins. Due to the fact that we must sort each bin after CDDT construction, pre-computation time is at worst $O(n * \theta_{discretization} + S^2 * \theta_{discretization} * log(S))$ for the same definition of n, where $S = norm(\text{map width}, \text{map height})$. In practice it is closer to $O(n * \theta_{discretization} + S * \theta_{discretization})$ since each CDDT has a small number of elements on average, as evidenced by the high demonstrated compression ratio.

The pruning operation described in IV-C reduces memory requirement, with a computational cost of $O(|map| * \theta_{discretization} * O(calc\_range)_{CDDT})$. The precise impact of pruning is scene dependent and difficult to analyze in the general case.

The ray cast procedure has three general steps: projection into CDDT coordinate space, the search for nearby zero points, and the computation of distance given the nearest zero point. The first and last steps are simple arithmetic, and therefore are theoretically constant time. The second step involves searching for a value in a sorted array of zero points. As previously discussed in IV-C, the number of zero points in each CDDT bin tends to be small and is bounded in map size. Thus, at worst the search operation requires $\log(norm(\text{map width}, \text{map height}))$ which is a constant value for a fixed map. Therefore, for a given map, our algorithm provides constant time performance.

VI. Experiments

We have implemented the proposed algorithm in the C++ programming language, as well as Bresenhams Line, ray marching, and the LUT approach for comparison. Our source code is available for use and analysis [TODO add a link to github], and Python wrappers are also provided for easier usage. All benchmarks were performed on a desktop computer on a Intel Core i5-4590 CPU @ 3.30GHz with 16GB DDR3 ram @ 1333MHz, running Ubuntu 14.04.

Basement Map, $\theta_{discretization}$: 108		
Method	Memory Usage	Init. Time
Bresenham's Line	1.37 MB	0.006 sec
Ray Marching	5.49 MB	0.16 sec
CDDT	6.34 MB	0.067 sec
PCDDT	4.07 MB	2.2 sec
Lookup Table	296.63 MB	15.3 sec

Synthetic Map, $\theta_{discretization}$: 108		
Method	Memory Usage	Init. Time
Bresenham's Line	1 MB	0.004 sec
Ray Marching	4 MB	0.13 sec
CDDT	2.71 MB	0.03 sec
PCDDT	1.66 MB	0.74 sec
Lookup Table	216 MB	9.1 sec

Fig. 8. Construction time and memory footprint of each method.

We evaluate algorithm performance in two synthetic benchmarks, using two different maps. The so called Synthetic map was created with photoshop, whereas the basement map was created via a SLAM algorithm on the RACECAR platform [cite this?]. The first benchmark computes a ray cast for each point in a uniformly spaced grid over the three dimensional state space. The second benchmark performs a large number of ray casts for randomly generated states.

To simulate real world operating conditions, we have implemented a particle filter localization algorithm using a beam mode sensor model. We provide statistics about the ray cast performance of each algorithm while being used to compute the sensor model. While care was taken in implementing the particle filter, there are undoubtedly many optimizations which could further improve end-to-end performance. Thus, we do not present end-to-end particle filter statistics.

Our sensor model is designed for the Hokuyo UST-10LX lidar scanner used aboard the RACECAR platform, which features a 270° field of view. Since this FOV is in excess of 180° we exploit radial symmetry discussed in the subsection IV-C to simultaneously ray cast in the theta and $\theta + \pi$ direction when possible. This optimization reduces the number of data structure traversals required by a third while still resolving the same number of ray casts. As is standard in such applications, we down-sample the laser scanner resolution to reduce the number of ray casts per sensor model evaluation, and to make the probability distribution over the state space less peaked.

VII. Conclusions

This work demonstrates that the proposed CDDT algorithm may be used in mobile robotics to accelerate sensor model computation when localizing in a two dimensional occupancy grid map. While the precomputed LUT approach is generally 1.1 to 1.7 times faster than the proposed algorithm, the memory footprint of the proposed data structure is approximately two orders of magnitude smaller, which

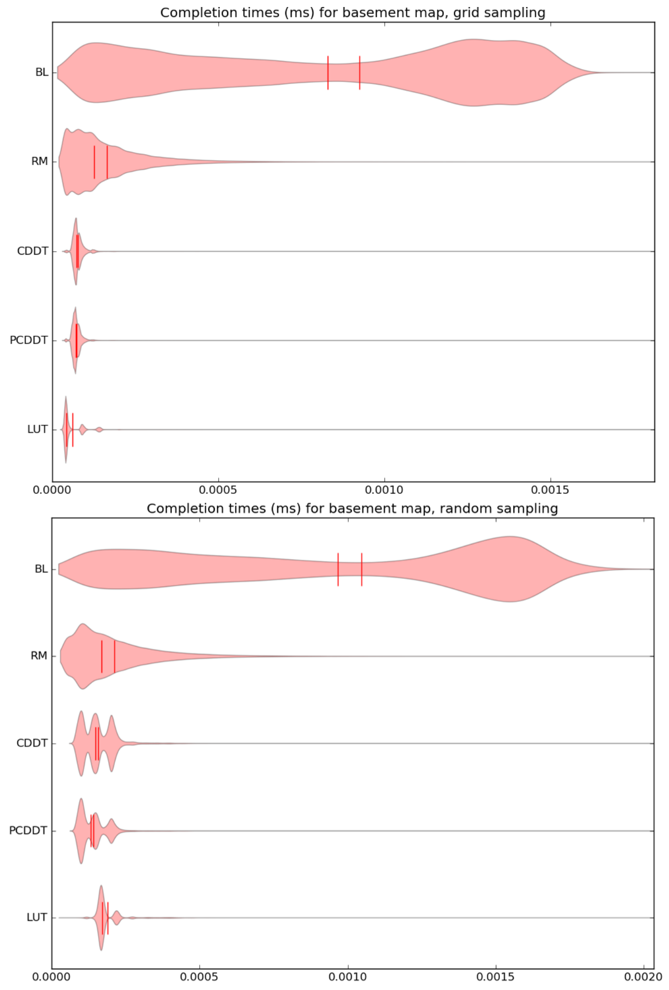


Fig. 9. Violin plots demonstrating histogram of completion time over a large number of queries for each ray cast method. Basement map. X axis shows time in milliseconds, and Y axis shows the number of queries that completed after that amount of time.

may be desirable for resource constrained systems. Other than the LUT approach, CDDT is significantly faster than the other methods evaluated. Ray marching is the next best, ranging from 1.28 to 2.25 times slower than PCDDT depending on access patterns and environmental factors. The comparison with the widely used Bresenham's Line algorithm is more stark, ranging from a factor of 5.39 to 14.8 in our benchmarks.

While our experiments reveal a multi-modal distribution of completion times for the CDDT algorithm, it does not exhibit a significant long tail distribution, confirming our constant time analysis. We suspect the various modes in completion time are due to short circuit cases in our implementation, as well as caching and kernel interrupt effects.

In future work we aim to further develop our approach to allow for incremental map updates without requiring a full reconstruction of the underlying acceleration data structure, a feature which currently only the slowest evaluated method supports.

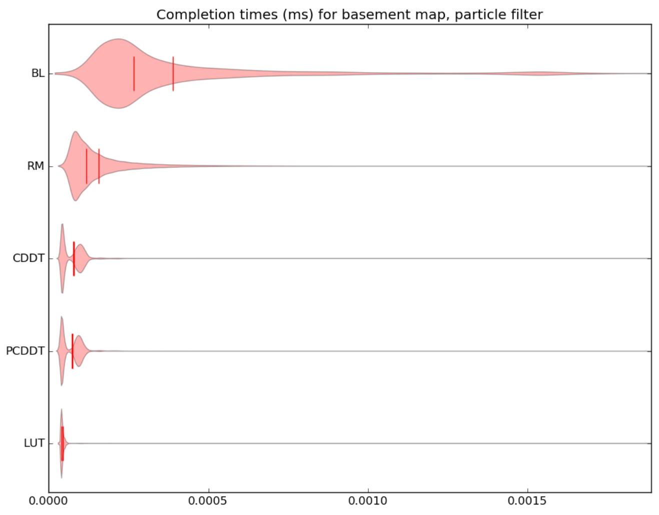


Fig. 10. Violin plots for ray casting completion times during the execution of a particle filter. Basement map. Ray casting is primarily done inside of hallways, reducing the mean ray length and thereby improving Bresenham's Line performance. CDDT and PCDDT exploiting radial symmetry while resolving the sensor model.

Synthetic Map Ray Cast Benchmarks

Random Sampling				
Method	Mean	Median	IQR	Speedup
BL	1.19e-06	1.41e-06	7.71e-07	1
RM	1.52e-07	1.25e-07	1.05e-07	7.81
CDDT	1.24e-07	1.05e-07	5.3e-08	9.59
PCDDT	1.19e-07	1.01e-07	5e-08	10.02
LUT	1.82e-07	1.68e-07	1.4e-08	6.55

Grid Sampling				
Method	Mean	Median	IQR	Speedup
BL	1.03e-06	1.20e-06	6.79e-07	1
RM	1.27e-07	1.03e-07	1.06e-07	8.06
CDDT	7.02e-08	6.8e-08	1e-08	14.63
PCDDT	6.94e-08	6.8e-08	9e-09	14.80
LUT	6.33e-08	4.2e-08	4.6e-08	16.21

Fig. 12. All times listed in seconds, speedup relative to Bresenham's Line

Appendix

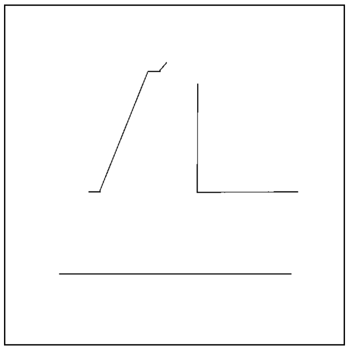


Fig. 11. Synthetic map occupancy grid. 1024x1024

Basement Map Ray Cast Benchmarks

Method	Mean	Median	IQR	Speedup
Random Sampling				
BL	9.66e-07	1.05e-06	1.08e-06	1
RM	2.12e-07	1.68e-07	1.64e-07	4.56
CDDT	1.58e-07	1.49e-07	9.1e-08	6.13
PCDDT	1.41e-07	1.32e-07	6.5e-08	6.83
LUT	1.89e-07	1.7e-07	2.1e-08	5.10
Grid Sampling				
Method	Mean	Median	IQR	Speedup
BL	8.29e-07	9.24e-07	9.53e-07	1
RM	1.65e-07	1.26e-07	1.34e-07	5.02
CDDT	7.69e-08	7.2e-08	1.6e-08	10.78
PCDDT	7.32e-08	7e-08	1.4e-08	11.33
LUT	6.13e-08	4.3e-08	4.6e-08	13.53
Particle Filter				
Method	Mean	Median	IQR	Speedup
BL	3.89e-07	2.67e-07	2.31e-07	1
RM	1.56e-07	1.18e-07	9.4e-08	2.49
CDDT	7.67e-08	7.8e-08	5.7e-08	5.07
PCDDT	7.22e-08	7.4e-08	5.4e-08	5.39
LUT	4.35e-08	4.1e-08	5e-09	8.94

Fig. 13. All times listed in seconds, speedup relative to Bresenham's Line

References

[1] J. Bresenham, Algorithm for Computer Control of a Digital Plotter, IBM Systems Journal, vol. 4, no. 1, pp. 25-30, 1965

[2] K. Perlin and E. M. Hoffert, Hypertexture. Computer Graphics, vol 23, no. 3, pp. 297-306, 1989.

[3] Sebastian Thrun, Dieter Fox, Wolfram Burgard and Frank Dellaert. Robust Monte Carlo Localization for Mobile Robots. Artificial Intelligence Journal. 2001

[4] Zuiderveld, Karel, Koning, Anton, and Viergever, Max, Acceleration of ray-casting using 3D distance transforms, Proceedings of the SPIE Visualization in Biomedical Computing 1992, Vol.1808, pp.324-335, 1992.

[5] Pharr, M., and R. Fernando. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation: Addison Wesley Professional. 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter08.html