Compressed Directional Distance Transform for Fast 2D Ray Casting

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

The ray cast operation is an important sub-problem for robot localization [1] using distance sensors to detect the environment. In particle filter localization, particles are reweighted according to a Bayesian update equation which relies on ray casting to determine the ground truth distance between the hypothesis pose and obstacles in a map. Effective particle filters often maintain thousands of particles, each of which must be reweighted tens of times per second, and each re-weighting may require tens of ray casts. As a result, millions of ray cast operations may be resolved per second, posing a significant computational challenge for resources constrained systems.

To combat this challenge, Fox et al. [1] 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. We propose an analogous data structure called a Compressed Directional Distance Transform (CDDT) which requires significantly less memory and precomputation time while still allowing near constant time ray casting queries.

Many robotic applications require three dimensional maps, however that is considered outside the scope of this work since many data structures exist for accelerating 3D ray casting, such as the bounded volume hierarchy and related variants. The five dimensional table required to precompute expected ranges for a discrete state space (x, y, z, θ, ϕ) would be prohibitively expensive to both compute and store, so 3D acceleration data structures are exclusively used when 3D ray casting is required. Though our approach does extend to higher dimensionality and would be smaller than the analogous five dimensional table, it would still be prohibitively expensive in greater than two dimensions, thus we do not consider it here.

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

[2]Sertac Karaman is with Faculty of Computer Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, USA `sertac@mit.edu`

II. RELATED WORK

Several widely used algorithms exist for ray casting in a two dimensional space. 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.

A. Bresenham's Line

Bresenham's line algorithm [2] incrementally determines the set of pixels that approximate the trajectory of query ray starting from the query point (x, y) and progressing in the θ direction. The algorithm terminates once the nearest occupied pixel is discovered, and the euclidean distance between that occupied pixel and the query (x, y) 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.

B. Ray Marching Distance Transforms

Similar to Bresenham's Line algorithm, ray marching [3] checks points along the line radiating outwards from the query point until an obstacle is encountered. The primary difference is that ray marching makes larger steps along the query ray, avoiding unnecessary memory reads. Beginning at the query point (x, y) the ray marching algorithm proceeds in the theta direction, stepping along the line by the minimum distance between each query point and the nearest obstacle. The algorithm terminates when the query point falls on an obstacle in the map. A precomputed euclidean distance transform of the map provides the distance between each query point and the nearest obstacle in any direction.

Ray marching is on average much faster than Bresenhams line, but edge cases exist in which the performance is equivalent. For example, a query ray which travels closely parallel to a wall will make small steps between each query point for the whole length of the ray. As a result, the performance of ray marching has a long tail distribution

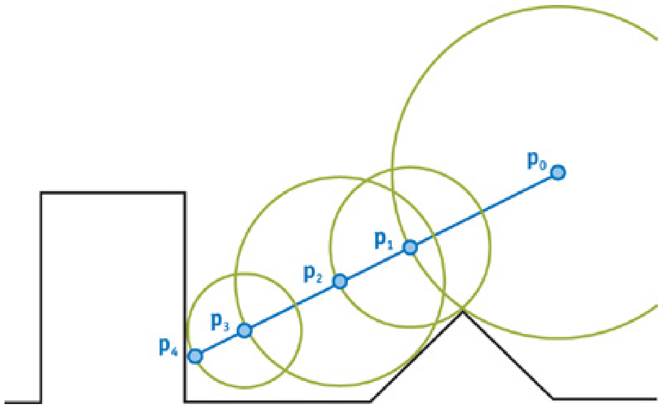

Fig. 1. Visualization of ray marching starting at p_0 towards p_4. Green circle around each query point represents the distance to the nearest obstacle from that point. Blue dots represent the query points, labeled in order of execution. From [4].

which can be problematic for near real time algorithms. Though highly scene dependent, on average ray marching runtime is roughly logarithmic in distance between obstacles, though in the worst case it is linear in map size.

C. Lookup Table

As previously described, a common acceleration technique for two dimensional ray casting is to simply 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 this 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 is small for x and y, it can be significant for theta. As the ray moves further away from the query point along the theta direction, angular discretization error accumulates. For queries (x, y, θ) discretized into (x_d, y_d, θ_d), the distance between the end of the ray $(x_d, y_d, _d)$ and its projection onto the line implied by (x, y, θ) 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 recursive Bayes filter since error is expected and directly modeled.

One method to improve accuracy of queries which fall between discrete states is to ray cast the neighboring discrete states and interpolate between them, rather than simply rounding to the nearest discrete state. 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 and therefore any increase in accuracy with respect to the stored map is effectively negated.

III. APPROACH

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. 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 (CDDT) described here.

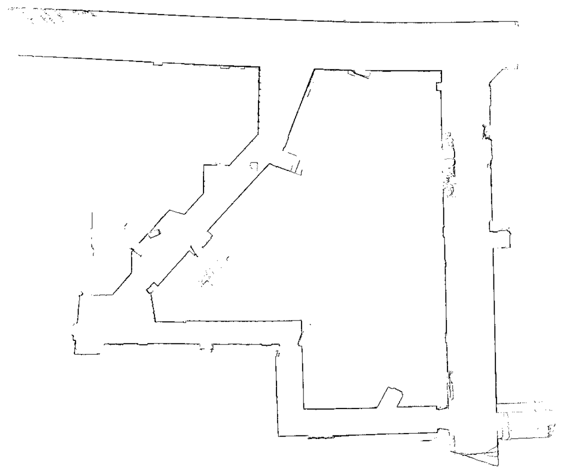

Fig. 2. Occupancy grid map of the Stata basement loop. 1200x1200

The euclidean distance transform (EDT) of an occupancy grid map stores the distance to the nearest obstacle in any direction for each point (x, y) in the map. In contrast, a directional distance transform (DDT) stores the distance to the nearest obstacle in a particular direction for each point (x, y) in the map. The key difference between a single slice of the LUT and a directional distance transform is the way in which it is computed. To compute the LUT slice, ray casting is performed in the θ_d direction from every (x_d, y_d) in the map using another algorithm such as Bresenhams line. In contrast, to compute the DDT, the obstacles in the map are rotated about the center of the map by $-\theta_d$ and the values of each column in the DDT may be directly computed as a sawtooth function which goes to zero whenever an obstacle is encountered in that column. Thus, each column in the DDT may be characterized as a sawtooth function with uneven periodicity. While it is true that the rotation of the occupancy grid introduces small errors as compared to ray casting for each cell, as previously discussed small errors do not significantly impact particle filter localization performance due to the use of a probabilistic error model. In practice, errors induced by map rotation are dominated by those inherently present in distance sensors.

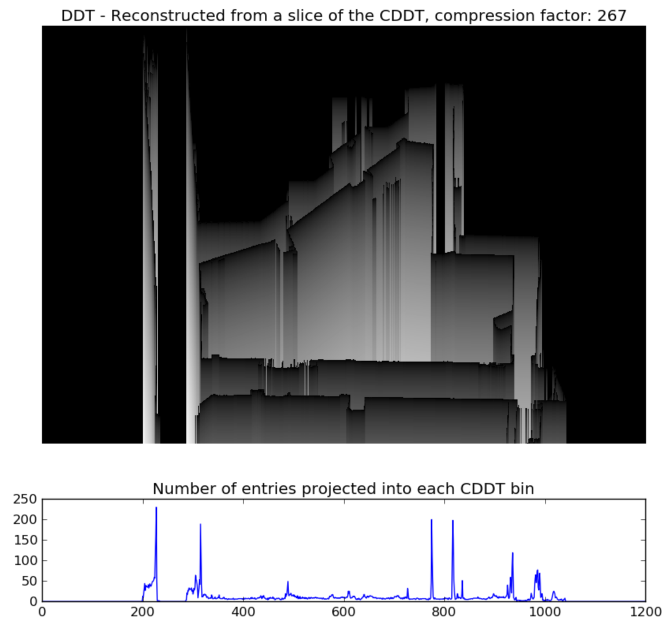

Fig. 3. A single slice of the DDT generated from the CDDT data structure for $\theta = 0$. The value of each pixel represents the distance to the nearest obstacle in the upwards direction with respect to the image above. Each column may be thought of as a sawtooth wave, and the points where the wave is zero are projected into the CDDT bin for that column for later indexing. The blue plot indicates the number of zero points projected into each CDDT bin. The compression factor of 267 is computed as two times the size of the corresponding LUT slice divided by the size of this slice in the CDDT. The factor of two is due to the use of rotational symmetry to reduce the number of slices maintained in the CDDT with respect to the LUT.

The conversion from a DDT to a CDDT is performed by converting the sawtooth response of each column into a set of points in which the sawtooth wave goes to zero. This conversion is lossless in the sense that each element of the table may be recovered as a function of the nearest zero point in a particular direction. If the set of zero points is sorted, then the magnitude of the saw tooth wave at any point in the original DDT may be quickly discovered by performing binary search for the nearest zero point in that column and finding the distance between the query point and that zero point. This losslessness and fast indexing allows for ray cast queries to be quickly resolved without storing the entire DDT, but rather by storing the sorted lists of zero points. For sake of efficiency, the full DDT need not be computed, but rather the CDDT may be built directly by projecting each obstacle into the coordinate space of each theta slice and storing a list of y-coordinates for each discrete x-coordinate. Once all scene geometry is projected into the table, each list of zero points is sorted to facilitate indexing.

Once the CDDT is constructed, querying the data structure to resolve ray casts involves

1) Projecting the query point into the coordinate space of the slice which corresponds to the discretized theta
2) Finding the nearest zero point which has a y-coordinate greater than or equal to the query point in the projected coordinate space

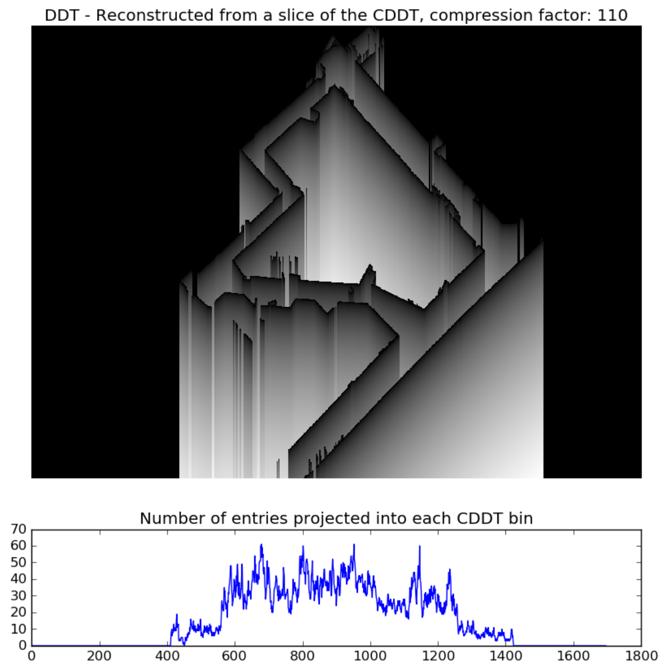

Fig. 4. Another slice of the DDT generated from the CDDT data structure for $\theta = 2.385$.

3) Return the distance between the query point and the nearest zero point in the projected coordinate space, which is simply the difference between y-coordinates

Binary search is theoretically optimal for finding zero points near to query states, in our implementation we conditionally use linear search when the number of zero points is below a certain constant (we use 64) as it has a superior memory access pattern in such cases. 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. *Rotational Symmetry*

A side effect of extracting the zero points from each column of the DDT is the introduction of rotational symmetry. For any given theta, the zero points accumulated by projecting scene geometry into the requisite coordinate space are mirror images of those accumulated by projecting scene geometry into the theta+pi direction. Therefore, one need only compute the CDDT for the range of discrete thetas from $[0, \pi)$ and the DDT for the range $[0, 2\pi)$ may be inferred, resulting in factor of two further reduction in memory usage.

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 in a particular (x, y, θ), the ray cast for $(x, y, \theta + \pi)$ may be resolved with a single additional memory read. Once the index i of the nearest zero point in the theta direction z_θ is discovered, the index of the nearest zero point in the opposite direction is simply i-1. For example, in robots with laser scanners sweeping angles larger than 180, this symmetry can be used to reduce the number of data structure traversals required to compute the sensor model by up to a factor of two.

B. *Incremental Update*

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 contrast, any change to the map would introduce inconsistencies in a precomputed LUT requiring a full re-computation of the data structure. Ray casting every state in a three dimensional grid is highly computational, ruling out the possibility of using a precomputed LUT with dynamic maps such as in SLAM algorithms. Similarly, ray marching operates on a euclidean distance transform which cannot be easily updated in an incremental fashion. While Bresenhams line algorithm is the slowest option evaluated in terms of ray cast performance, it has the benefit that it operates directly on the occupancy grid map and can therefore be used with dynamic maps. For this reason, Bresenhams line is one of the most commonly used algorithms for ray casting in occupancy grids.

To allow incremental update of the CDDT, the sorted vector data structure used in our implementation should be replaced with an alternative which allows for $O(logn)$ insert and delete, such as a randomized skip list. To aid in node deletion, pointers could be maintained from each grid element to the associated leaf node in the randomized skip lists with a constant factor increase in memory usage.

In this case, the cost of inserting or deleting an occupied grid cell would be $O(theta\_discretization * \log(numberoflutbinelements))$. The number of elements in each lut bin tends to be small and is bounded, so this becomes $O(theta\_discretization)$ which is likely not prohibitive for real-time performance.

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 significantly faster than performing Bresenhams line, and therefore recommend that this modification be the subject of future research.

C. *Pruning*

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. 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. This can result in significant memory usage and data structure construction time reductions in dense maps where the number of obstacles adjacent to empty cells is small compared to the total number of obstacles. 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 which may be removed by the morphological operation.

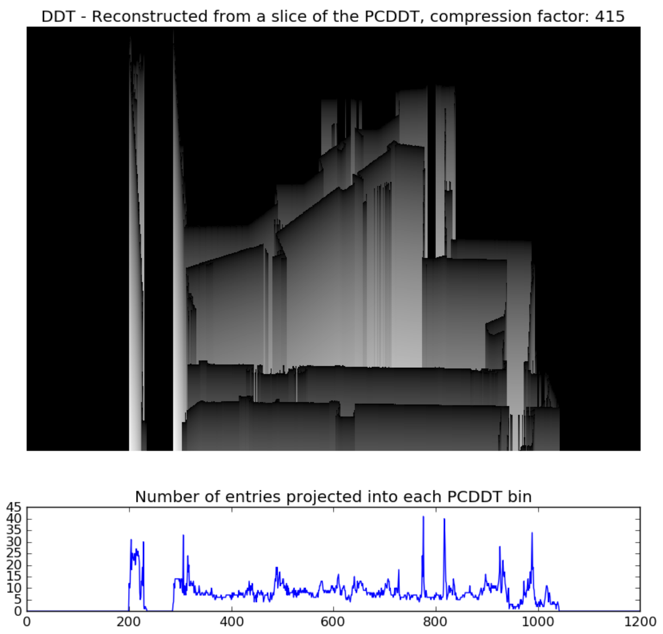

Fig. 5. The same slice of the DDT as 3. Notice that the number of elements projected into each CDDT bin is less peaky since the pruning operation removes most elements of long lines of obstacles. Also note the higher compression factor of 415 for this slice, increased from 267 in the unpruned CDDT.

Additionally, consider a line of obstacles aligned along the Y-axis. Every element in this line will be projected into a single list of zero points for the theta=0 slice of the CDDT. However, the middle elements of the line will never result in a collision. Any ray cast from adjacent states in the $\theta = 0$ direction will return early in the occupancy grid check, and any ray cast from outside the wall will collide with one of the edge-most obstacles in that line. Therefore in the theta=0 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 theta not aligned with an axis. Rather than attempting to analytically determine which obstacles may be discarded, it is simpler to prune the data structure by ray casting from every possible state (as in the LUT data structure) and discard any zero point which is unused.

Pruning does also increase precomputation 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.

IV. EXPERIMENTS

We have implemented the proposed algorithm in C++, 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.

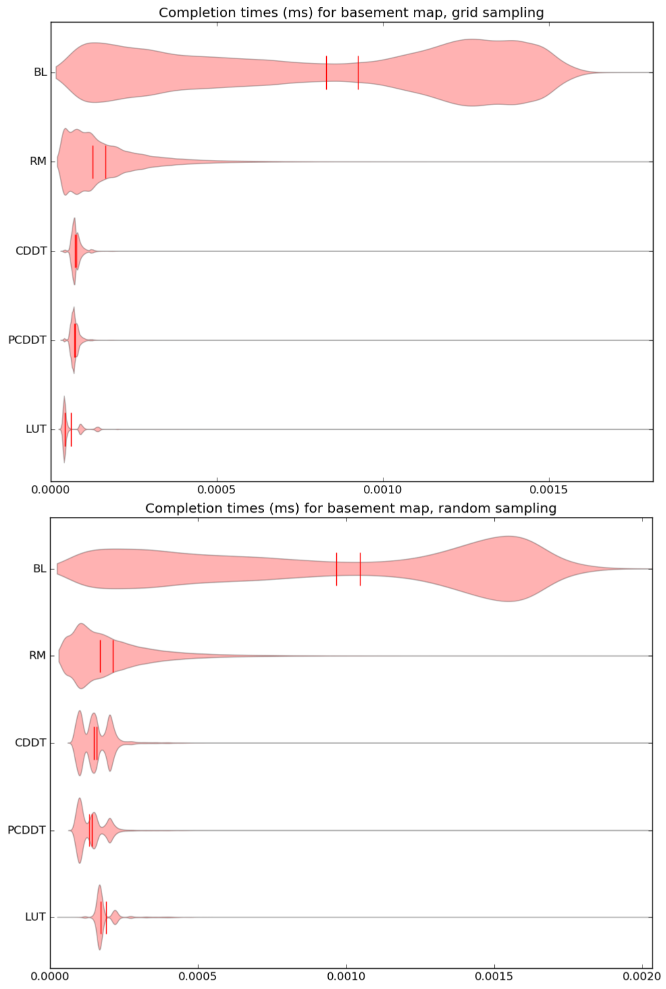

Fig. 6. Violin plots demonstrating histogram of completion time over a large number of queries for each ray cast method. Basement map. X axis shows time in milliseconds, and Y axis shows the number of queries that completed after that amount of time.

A. Synthetic Benchmarks

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.

Basement Map, θ 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

Fig. 7. The construction time and memory footprint of each method for the Basement map.

Synthetic Map, θ 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. The construction time and memory footprint of each method for the Synthetic map.

B. Particle Filter Experiments

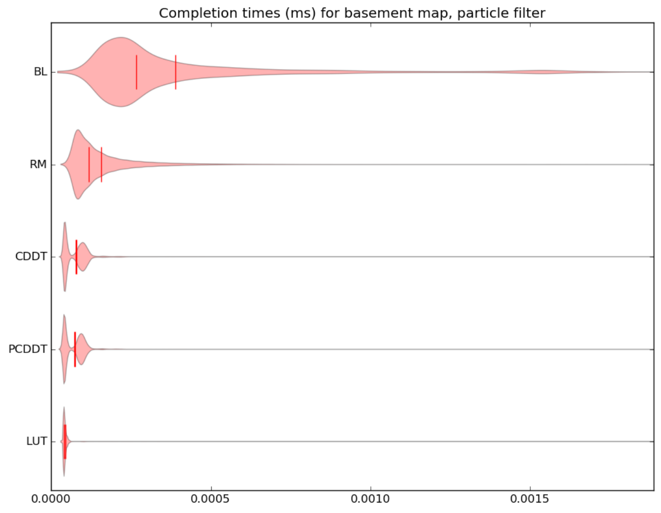

Fig. 9. Violin plots for ray casting completion times during the execution of a particle filter. Basement map. Ray casting is primarily done inside of hallways, reducing the mean ray length and thereby improving Bresenham's Line performance. CDDT and PCDDT exploiting radial symmetry while resolving the sensor model.

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, as well as end to end statistics on the number of particles able to be maintained in real-time. While care was taken in implementing the particle filter, there are undoubtedly many optimizations which could further improve end-to-end performance. Thus, these benchmarks should not be considered an upper bound on performance and should be interpreted comparatively.

Our sensor model is designed for the Hokuyo UST-10LX lidar scanner used aboard the RACECAR platform [cite this?], which features a 270 field of view. Since this FOV is in excess of 180 we exploit radial symmetry discussed in [radial symmetry section] to simultaneously ray cast in the theta and $\theta + 180$ direction when possible. As demonstrated in the below figure, this optimization reduces the number of data structure traversals required by a third as compared to the other methods evaluated, while still resolving the same number of ray casts. As is standard in such applications, we downsample the resolution of the laser scanner to reduce the number of ray casts per sensor model evaluations and to make the probability distribution over the state space less peaked.

V. 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 significantly smaller which 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.

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.

APPENDIX

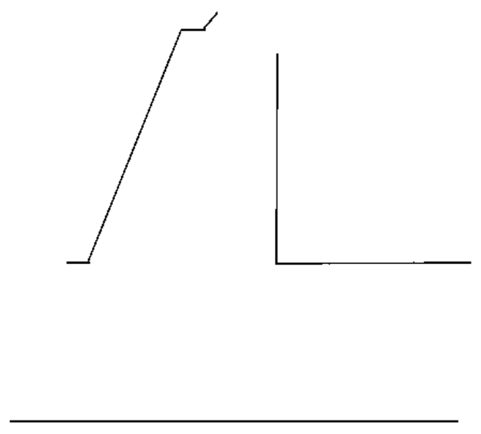

Fig. 10. Synthetic map occupancy grid. 1024x1024

Synthetic Map Ray Cast Benchmarks

Method	Mean	Median	IQR	Speedup
Random Sampling				
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				
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. 11. All times listed in seconds, speedup relative to Bresenham's Line

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

REFERENCES

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

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

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

[4] 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