**2014 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS 2014)
September 14-18, 2014, Chicago, IL, USA**

Layered Costmaps for Context-Sensitive Navigation

David V. Lu, Dave Hershberger, and William D. Smart

Abstract— **Many navigation systems, including the ubiquitous ROS navigation stack, perform path-planning on a single costmap, in which the majority of information is stored in a single grid. This approach is quite successful at generating collision-free paths of minimal length, but it can struggle in dynamic, people-filled environments when the values in the costmap expand beyond occupied or free space.**

We have created and implemented a new method called *layered costmaps*, which work by separating the processing of costmap data into semantically-separated layers. Each layer tracks one type of obstacle or constraint, and then modifies a master costmap which is used for the path planning. We show how the algorithm can be integrated with the open-source ROS navigation stack, and how our approach is easier to fine-tune to specific environmental contexts than the existing monolithic one. Our design also results in faster path planning in practical use, and exhibits a cleaner separation of concerns that the original architecture. The new algorithm also makes it possible to represent complex cost values in order to create navigation behavior for a wide range of contexts.

I. Introduction

Navigation algorithms have become increasingly sophisticated over the decades. They can process large amounts of sensor data to keep track of the locations of obstacles and free space with great accuracy. Combined with the right path planners, they can navigate robots around their environments with great skill. However, many of these navigation algorithms suffer from the same problem: the algorithms optimize based on the single constraint of finding efficient collision-free paths.

Such an algorithm is fine for many use cases, or for navigation in the abstract, if all that matters is getting from point A to point B. It is not sufficient for other use cases. For robots moving in dynamic environments populated with people, more complex constraints need to be integrated into the optimization. Moving from one point to another is just one part of a larger context. It is not enough that a robot moves around an obstacle just to avoid a collision; the robot must treat that obstacle differently because of what it is semantically. For example, driving a few centimeters away from a table is perfectly fine in most cases. However, driving that closely to a person is socially undesirable. Yet, if the navigation algorithm treats all sensed obstacles equally, there is no way for the path planner to be able to choose one path over the other.

Lu is a PhD candidate in the Dept. of Computer Science at Washington University, St. Louis, Missouri, 63130. davidlu@wustl.edu

Hershberger contributed to this work while he was a research scientist at Willow Garage, Inc. Menlo Park, CA hersh@gmail.com

Smart is faculty in the Department of Mechanical Engineering, Oregon State University. bill.smart@oregonstate.edu

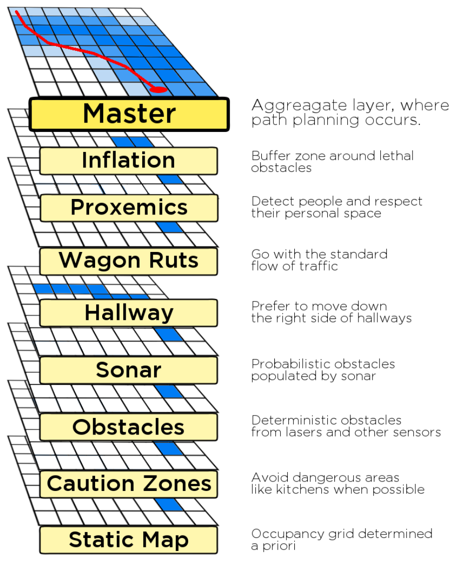

Master
Aggreagate layer, where path planning occurs.
Inflation
Buffer zone around lethal obstacles
Proxemics
Detect people and respect their personal space
Wagon Ruts
Go with the standard flow of traffic
Hallway
Prefer to move down the right side of hallways
Sonar
Probabilistic obstacles populated by sonar
Obstacles
Deterministic obstacles from lasers and other sensors
Caution Zones
Avoid dangerous areas like kitchens when possible
Static Map
Occupancy grid determined a priori

Fig. 1. A stack of costmap layers, showcasing the different contextual behaviors achievable with the layered costmap approach.

There are many additional scenarios, beyond respecting people's personal space, where choosing the shortest collision-free path may not be optimal. Given information about where people often are, a longer path that avoids probable obstacles may be preferable. The robot must also consider the utility of entering potentially hazardous areas, such as kitchens, which are valid paths, but they come with a cost. Even simple factors like driving on the right side of a hallway will need to be considered. Which path the robot takes will depend on having additional information about the larger contexts.

The information about the environment that the path planners use is stored in a costmap. In a traditional costmap, all of the data is stored in the singular grid of values, in what we term a *monolithic costmap*. The monolithic costmap has been the prevailing technique because of its simplicity, in that there is only one place to read values from and write values to. One result of this is that a great deal of semantic information about the values in the costmap is lost, which makes proper maintenance of the costmap from cycle to cycle more difficult.

In this paper, we introduce our solution for incorporating the additional semantic information into costmaps, with a new approach called *layered costmaps*. Using the ROS Navigation framework as a starting point, we show that the layered costmaps replicate the functionality of the previous navigation algorithm, while adding the flexibility to handle more contexts. Fig. 1 shows a possible configuration of the layered costmap with different types of layers. We will discuss the algorithm and data structure, and where they have improved the previous approach. Then we will examine the different layers that can be added to the costmap, both old and new, and the environmental contexts that they integrate.

II. Related Work

The focus of this work is on grid-based representations used for path planning. The immediate ancestor of modern costmaps is the occupancy grid, developed by Moravec and others at CMU in the 1980s [1, 2]. The semantics of the contained values is straightforward; each cell's value is the probability that there is an obstacle present, and thus the update process is a straightforward application of Bayes rule. Konolige [3] and Thrun [4] improved the probability model to better localize obstacles.

The grid-based costmap approach (where the grid values are not probabilities but costs), has proven useful especially when the location of the obstacles is easier to pin down (i.e not using sonar). In the past, these approaches have primarily focused on binary costmaps, where the cell is either occupied or free space. Now more complex costs are being added to the costmap, resulting in trickier semantics for the values in the costmap. These non-lethal costs, with values between the occupied and free, typically represent soft constraints. Autonomous vehicles used such values to optimizing for driving on the correct side of the street and other preferential driving behaviors[5]. Gerkey and Agrawal [6] represented different types of terrain and their traversability with different costs in the costmap. The soft constraints are also used for human-robot-interaction-based constraints. The costmap system by Sisbot et al. [7] took into account peoples' personal space and fields of vision, as did Kirby et al. [8] who also modeled social behaviors like passing on the right. Even more complicated cost methods for people-aware navigation were developed by Svenstrup et al. [9] and Scandolo and Fraichard [10].

III. The Monolithic Costmap

The monolithic costmap, with all of the data stored in a single grid of values forms the basis of most costmap implementations being used, including the ubiquitous ROS [11] Navigation Stack. In this paper, we focus on the ROS Navigation algorithm and implementation since it is a widely used and runs on dozens of robot hardwares[1]. It uses a monolithic costmap for global planning, and another for local planning.

The monolithic costmaps have proven effective at calculating the minimal-length collision-free paths. Writing initial values into the costmap is straight-forward, but with the limited storage space, the update process is problematic, limiting the types of achievable functionality and on costmap's efficiency and extensibility. The main weaknesses of the monolithic costmap approach are as follows.

[1]`wiki.ros.org/navigation/RobotsUsingNavStack`

1) Limited Information During Update Step: One major limitation of monolithic costmaps is that most of the information in the costmap is stored in one location. Consider the relatively simple example of a conflict between the sensor data and the values already in the global costmap. The sensor data indicates that a certain area is clear, while the costmap indicates there is an obstacle. The correct method for updating the costmap depends on the origin of the data and additional semantic information. One scenario might be that the previous values indicated a prior position for a person who has now moved. Then, the correct behavior may be to overwrite the lethal values in the costmap with the new free values, allowing the robot to pass through the newly vacated space. However, an equally valid scenario is that the values in the costmap originate from the static map, which was created to include obstacles that cannot be seen by the sensors, such as glass walls. In that case, the lethal values should stay in the costmap.

It is impossible to differentiate these two cases in a monolithic costmap, as both present as lethal values in the costmap. Any semantic data for what the values in the costmap represent is stripped from the data as soon as it is reduced to a single value in the costmap.

This is also problematic for properly handling three-dimensional obstacle data. The original developers of the ROS implementation encountered this problem when they used three-dimensional sensors like a tilting laser range finder. If the obstacle data is stored only in the monolithic costmap, obstacles at different heights could be inappropriately removed by clearing observations. Thus, they introduced voxel grids to keep track of the additional information[12]. The solution works for extending the monolithic costmap's functionality in this one use case, but does not generalize.

The limited information becomes more problematic as the number of data sources and types for the costmap increases. Consider when there are multiple non-lethal data sources, each with an individual semantic meaning. If the values are added together in the monolithic costmap, a change to one of the values will result in each of the individual values needing to be recalculated.

2) Fixed Update Areas: The lack of semantic information in the monolithic costmap also makes it difficult to tell how long any particular cost value has been in the costmap. Hence, if the updated area needs post-processing or to be published to some external source, there is no established way to determine the scope of the most recent updates. An ineffective way to deal with this problem is to conservatively estimate a swath of the map that covers the entire area that *could* have been updated, which is what the ROS implementation does. In practice this means updating a roughly 6m x 6m square around the robot, regardless of how much of that space was actually updated.

3) Ad-hoc Update Process: The lack of an established paradigm for maintaining and updating a costmap results in implementations that take an ad-hoc approach. This method has worked thus far due to the relatively small number of data sources used in practice, but it becomes infeasible as the number of sources increases. In order to ensure that the data is combined in the correct way, every data source needs to be aware of every other data source.

Even in the prior work that define useful algorithms for calculating costs, the process that they use to actually integrate them with their full costmap is usually opaque. Without the information about how precisely costmaps are updated, accurately replicating results becomes impossible.

4) Semantically Fixed Interpretation: In addition to limiting the amount of information that costmaps contain, monolithic costmaps also constrain the types of information that can be used. The monolithic costmap also only affords a single interpretation of the values in the costmap. The original occupancy grid definition of costmaps used a probabilistic interpretation. Alternatively, the value could represent some cost/penalty for being at a certain location. With the monolithic costmap, it is ambiguous how to combine a probabilistic data source with a cost-based one.

The ROS costmap implementation has additional problems since the only type of information it accepts is binary obstacle data, i.e. where there are definitely obstacles or there is definitely free space. Adding non-lethal costs does not fit into its monolithic framework. With just one data-type in the costmap, the information is semantically fixed.

IV. Layered Costmaps

A. Data Structure and Update Algorithm

To counteract the limitations introduced in the previous section, we devised the layered costmap. The data structure still contains a two-dimensional grid of costs that is used for path planning. The key difference is how the values of this *master costmap* are populated. Instead of storing data directly in the grid, the layered costmap maintains an ordered list of layers, each of which tracks the data related to a specific functionality. The data for each of the layers is then accumulated into the master costmap, which takes two passes through the ordered list of layers.

In the first pass, the `updateBounds` method, each layer is polled to determine how much of the costmap it needs to update. The layers are iterated over, in order, providing each layer with the bounding box that the previous layers need to update (initially an empty box). Each layer can expand the bounding box as necessary. This first pass results in a bounding box that determines how much of the master costmap needs to be updated. During the second pass, the `updateValues` method is called, during which each successive layer will update the values within the bounding box's area of the master costmap. Fig. 2 illustrates the update algorithm using a set of layers that replicate the behavior of a basic monolithic costmap.

Some layers will maintain their own version of the costmap for caching results. This is one of the primary ways the data structure maintains semantic information about the data. For example, an obstacles layer keeps a private costmap of the same size as the master costmap to store the results of all previous ray-tracing and marking steps. Since the values in the private costmap are only accessible to the particular layer, the information stored within cannot be lost by another data source writing over it. This in turn minimizes how frequently the costmap must recalculate values that had previously been overwritten.

Other layers do not require that much data to be kept from cycle to cycle and will update the master costmap with their data on each turn, or will simply operate on the data that other layers have already written into the master costmap.

The example in Fig. 2 shows how the previous ad hoc approach used to generate the costmap can be refined into a neat, well-defined process. A more precise explanation of how each layer changes the master costmap is in section VI.

B. Benefits

The layered costmap approach specifically addresses the limitations of the monolithic costmap.

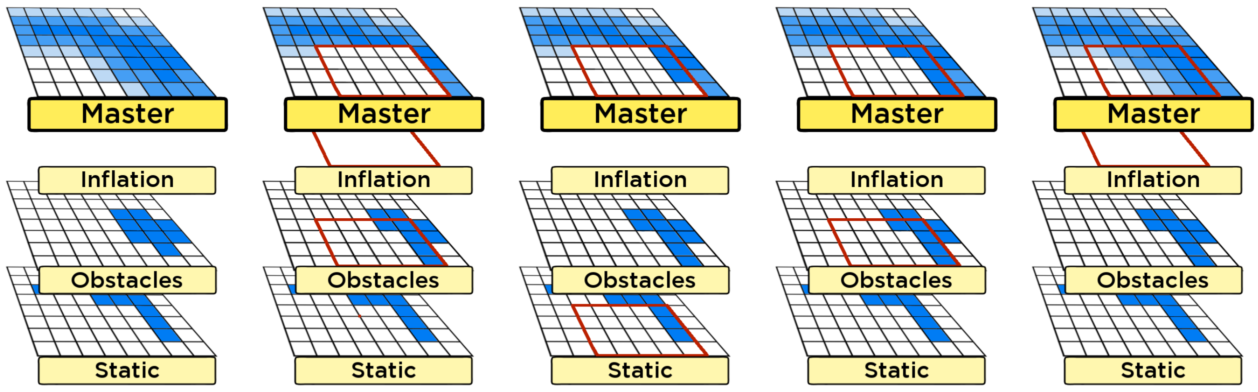
1) Clearer Update Step: Different types of costmap information are added to separate layers in the layered costmap approach, making the update step more clearly delineated. If the desired behavior included the ability to treat static obstacles, sensed laser obstacles and sensed sonar obstacles differently, storing those obstacles in their own layers simplifies the bookkeeping substantially. Each layer only needs to keep the information consistent with other information of the same type.

The layered costmap also eliminates contention between the competing costmap information sources. Each layer only needs to be updated as new information of that type comes in. If a layer remains largely static, it does not need to be recalculated each time another layer updates some subarea. The static layer merely needs to update into the master costmap and the update can move on to the next layer.

This clearer separation of concerns also makes the individual components of the costmap easier to tune. Initial users can introduce one layer at a time and debug each in turn.

2) Dynamic Update Areas: As opposed to the fixed or unknown regions that are updated on each round of updates in the monolithic costmaps, by virtue of the `updateBounds` pass through the layers, the layered costmap only updates the region of the map that the individual layers deem necessary. This gives the costmap extra stability, guaranteeing that only values within the bounding box are updated. Furthermore, it can potentially be more efficient by updating smaller amounts of the map.

3) Ordered Update Process: As opposed to the undefined order in which elements in the monolithic costmap were updated, the layered costmap has an explicit ordering. In our example, it is clear that the inflation layer will inflate values from both the obstacles and static layers by virtue of the inflation layer coming after the other two in the ordered list. Furthermore, the interactions between layers are explicitly specified. Each costmap can be configured to combine the previous value and the layer's value as a maximum, minimum or some other mathematical function of the two.



(a) Initial Costmap Values (b) UpdateBounds (c) UpdateValues: Static (d) UpdateValues: Obstacles (e) UpdateValues: Inflation

Fig. 2. Update Algorithm - In (a), the layered costmap has three layers and the master costmap. The obstacles and static layers maintain their own copies of the grid, while the inflation layer does not. To update the costmap, the algorithm first calls the `updateBounds` method (b) on each layer, starting with the first layer in the ordered list, shown on the bottom. To determine the new bounds, the obstacles layer updates its own costmap with new sensor data. The result is a bounding box that contains all the areas that each layer needs to update. Next, each layer in turn updates the master costmap in the bounding box using the `updateValues` method, starting with the static layer (c), followed by the obstacles layer (d) and the inflation layer (e).

4) Flexible Configurations: Finally, and most importantly, the capabilities of the layered costmap approach are endless. The layers needed to implement an equivalent set of behaviors to the previous implementation are only the beginning. As many layers as the robot operator desires can be added to the layered costmap. The result is that the individual layers can implement arbitrarily complex logic for updating the costmap, expanding the costmap's semantic possibilities. Each of the layers can also have its own independent representations of the data, such that probabilistic occupancy grids can exist in their own layers alongside cost-based layers.

V. Comparisons

A. Implementation Specifics

Although the algorithm and data structure for layered costmaps are system-agnostic, due to the ubiquity of the platform, we focused on implementing the system to work with the ROS Navigation stack in order to demonstrate the capabilities of the approach. The layered costmap implementation keeps the `costmap_2d` API mostly in tact and like the rest of the navigation code, is implemented in C++, as are each of the layers.

Implementing a layer is quite easy. First, a new class must be created which extends the `costmap_2d::Layer` class. This means implementing the `initialize` function (where the layer can independently subscribe to any data sources in the ROS ecosystem), the `updateBounds` function and the `updateCosts` function. Independently compiled layers can be plugged in to the layered costmap with simple run-time parameter changes.

Whereas previously the costmap class had special cases to deal with whether there was a static map or not, or whether to track the obstacles in three dimensions, these cases are instead handled by configuring the global and local costmaps with different layers.

The two implementations of costmaps were run through repeated simulated trials in Gazebo scenarios. The robot employed was the PR2, as was used in the initial benchmarking of ROS navigation[12]. After hundreds of simulations, we found no noticeable difference between the paths generated by the two implementations, with regard to path length, time to completion and relation to obstacles.

B. Timing Comparison

One of the most critical statistics we analyzed was the average runtime of the costmap update cycle. Due to the speeds at which the local planning needs to run and adjust to new obstacles, the update process must be quite quick. The standard we aimed to achieve was an update frequency of at least 5 Hz (as had been used by the previous implementation), capping the individual cycle runtime at 0.2 seconds. The monolithic costmap was able to exceed that by an order of magnitude or two, depending on the specifics of the environment and the system running the costmap. The layered costmap implementation also depends on these variables. Ultimately, the layered implementation runs faster in certain scenarios and slower in a few corner cases.

In our simulations, the average update times for the global costmap were 0.00166 and 0.00236 seconds for the monolithic and layered implementations respectively, and for the local costmap, the update times were 0.00493 and 0.00463 seconds respectively. Using a one-sided t-test, we found no significant difference in the average local update time. The global update time is significantly slower with the layered costmap ($p<0.001$). However, we determined that this was the result of a sparse simulated environment. In those simulations, the robot was placed in a completely open environment except for a single, relatively small obstacle between the robot and its goal. Thus, the robot's laser readings extend to their maximal distances in most directions, resulting in a large area that the layered implementation needs to update.

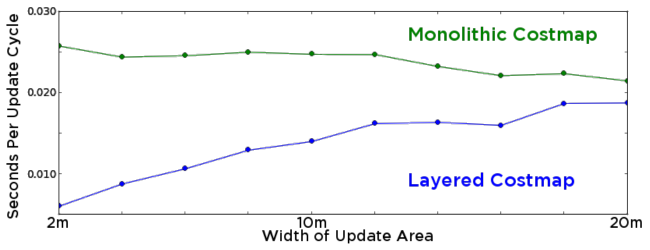


Fig. 3. Update Time vs. Update Area - Since the size of the updated area stays constant with the monolithic implementation, the timing stays roughly constant. However, the updated area varies with the layered costmap implementation, so the timing changes as well.

This area needs to be updated by each layer, slowing down the overall update speed.

We also simulated the robot in more cluttered environments, in which the robot was surrounded on all sides by walls at set distances away. With walls very close to the robot, the update area is much smaller. As seen in Fig. 3, in this scenario, the layered costmap is faster. As the updated area grows, the monolithic implementation's update time stays roughly constant, whereas the layered costmap's update time grows to match the increasing number of cells to update. Given that the costmap system is designed for working in cluttered, fast-changing environments, the layered costmap's speed in those environments are more relevant.

C. *Navigating the Real World*

In addition to our thorough simulated tests, we also tested the layered costmap on the PR2 platform in the real environments. The tests were primarily performed in the office environment at Willow Garage. Using the layers to mimic the monolithic costmap structure, the PR2 was able to successfully replicate all the path planning behavior of the previous implementation. However, the more exciting results occurred when we modified the layers to get behavior that was impossible with the monolithic costmaps.

First, by separating the static and obstacle layers, we controlled whether the obstacle layer had the power to overwrite the static map. As mentioned in section III, the monolithic costmap navigation could improperly clear parts of the static map, leading to the robot planning a path that moved through a solid wall. By only allowing the obstacle layer to ray-trace and clear the sensed obstacles (and not those in the static map), the wall never was cleared from the master costmap, eliminating the embarrassing wall-charging behavior.

The introduction of new layers also enables new previously impossible behavior. The motivating use case behind our investigations of the costmap was to create socially-aware robot navigation similar to the works cited in section II. We successfully integrated such a layer into the PR2's path planning[2]. The details of the new layer and other layers we created are elaborated upon in the following section.

[2]As seen in this video: youtube.com/watch?v=Pzx0yyEcfgI

VI. The Layers

Beyond the functionality that allows the layered costmap to replicate other costmaps, its main virtue is the ability to easily integrate additional layers which will be treated in the same way as the other elements of the costmap. These additional layers give the costmap the ability to represent information from many varied contexts and generate motion that reacts appropriately to those contexts.

A. *Standard Layers*

Static Map Layer: In order to perform global planning, the robot needs a map that reaches beyond its sensors to know where walls and other static obstacles are. The static map can be generated with a SLAM algorithm a priori or can be created from an architectural diagram.

When the layer receives the map, the updateBounds method will need to return a bounding box covering the entire map. However, on subsequent iterations, since it is static after all, the bounding box will not increase in size. In practice, the static map has been the bottom layer of the global costmap, and thus it copies its values into the master costmap directly, since no other layers will have written into the master before it.

If the robot is running SLAM while using the generated map for navigation, the layered costmap approach allows the static map layer to update without losing information in the other layers. In monolithic costmaps, the entire costmap would be overwritten.

Obstacles Layer: This layer collects data from high accuracy sensors such as lasers and RGB-D cameras and places it in a two dimensional grid. The space between the sensor and the sensor reading is marked as free, and the sensor reading's location is marked as occupied. During the updateBounds portion of each cycle, new sensor data is placed into the layer's costmap, and the bounding box expands to fit it.

The precise method that combines the obstacles layer's values with those already in the costmap can vary, depending on the desired level of trust for the sensor data. Previously, the default behavior was to overwrite the static map data with the sensor data. This was most effective in scenarios where the static map may be inaccurate, and is still available in the layered approach. However, if the static map is more trustworthy, then the layer can be configured to only add lethal obstacles to the master costmap.

Voxels Layer: This layer has the same function as the Obstacles Layer, but tracks the sensor data in three-dimensions. The three dimensional voxel grid, introduced in Marder-Eppstein et al. [12] allows for more intelligent clearing of obstacles to reflect the multiple heights at which they can be seen.

Inflation Layer: As discussed earlier, the inflation process inserts a buffer zone around each lethal obstacle. Locations where the robot would definitely be in collision are marked with a lethal cost, and the immediately surrounding areas have a small non-lethal cost. These values ensure the robot does not collide with lethal obstacles, and prefers not to get too close. The updateBounds step increases the previous bounding box to ensure that new lethal obstacles will be inflated, and that old lethal obstacles outside the previous bounding box that could inflate into the bounding box are inflated as well. The updateValues step operates directly on the master costmap, without storing a local copy.

B. New Functionality

Sonar Layer: Monolithic costmaps are capable of handling sonar data, but layered costmaps increase the options for how to deal with it. Dedicating a layer to sonar readings can avoid problems with glass walls being cleared out by laser observations. Furthermore, we can also use this layer to treat sonar data differently than the high accuracy obstacles layer. The sonar layer we built implements a probabilistic sonar model and updates the costmap using Bayesian logic. We can then set a cutoff probability in which we only write data that we are relatively sure about into the master costmap. Note that this approach allows us to maintain the semantic meanings of the probabilities without having to directly combine them with the costs.

Caution Zones Layer: This layer gives us the ability to specify areas of the robot environment with greater detail than free/occupied. Despite being free of obstacles, most robots will want to avoid navigating into stairwells leading down. Or perhaps the robot should never navigate into a particular person's office. There are numerous scenarios where operators will want to restrict where the robot can safely drive, despite appearing navigable. One technique seen in practice for these restrictions is to mark obstacles on the static map. This technique can work, but removes information from that map that might be needed for other applications, such as AMCL. This layer also affords us the ability to mark zones that are not necessarily forbidden, but not desirable. Adding a non-lethal cost to a kitchen can ensure the robot does not drive near hazardous liquids unless there is no other option. These zones can also be used for areas where it would be socially less acceptable to be, such as the space between a person and an object they are interacting with, like a TV, as seen in Ferguson and Likhachev [5].

Claustrophobic Layer: The inflation layer adds a small buffer around lethal obstacles, but the claustrophobic layer adds a larger buffer to increase the relative cost of driving close to obstacles. As a result, the robot would prefer to move in wide open spaces as far from obstacles as possible, thus maximizing the clearance to any sensed obstacles. This layer would be useful for scenarios with more uncertainty about the exact location of the robot relative to obstacles and the odds or costs of driving into an obstacle are high.

C. Human-Robot Interaction Layers

As seen in the Section II, one of the primary motivations for adding more complex costs to the costmap is for modeling constraints introduced by human-robot interaction.

Proxemic Layer: There has been a steady rise in the study of the spatial relations between people and robots, as well as methods for ensuring robots do not violate the expected relations. The most common way this is done is by adding Gaussian distributions, or mixtures of multiple Gaussian distributions, to costmaps, as in Kirby et al. [8]. These adjustments create areas around detected people that makes paths passing closer to people more costly, respecting their proxemic concerns.

We created a proxemic layer which implements Kirby's mixture of Gaussians model. Using the location and velocity of detected people (extracted from laser scans of the person's legs), the layer writes the Gaussian values for each person into the layer's private costmap, which are then added into the master costmap. The values generated are scaled according to two different parameters, the amplitude and the variance. In general, as you increase these parameters, the optimal path moves further from the person. However, as discussed in [13], there is a limit to how high these parameters can be changed before the optimal path changes to be the shortest path. This means that tuning the parameters in an attempt to get the robot to travel further away, the opposite happens, which is socially suboptimal. The results from [13] were replicated using fully simulated paths in the Gazebo simulator.

We have also begun to implement layers based on the more complex proxemic models mentioned in Section II. Some of the models assume more information than a person's location and orientation, and our layer implementations assume they are paired with robust enough sensor capabilities to detect things like head and body pose.

Hallway Layer: In some cultures, there is the custom of walking on the right side of pathways, much in the same way drivers in many countries stay to the right side of the road. We implemented a layer that determines whether the robot is in a hallway and dynamically will increase the cost on the left side of the hallway to have the robot prefer the right side. We used a similar model in a recent user study [14] where the layer changed costs to make the robot to prefer navigating on the opposite side of the corridor as the closest person to it (which was often the right side). The addition of this layer was shown to not only effectively move the robot to one side of the hallway, but also to make the person behave more effectively during the interaction.

Wagon Ruts Layer: If the robot aims to avoid being socially invasive and minimize unexpected obstacles, one effective strategy could include mimicking human traffic patterns. This layer can decrease the cost of paths that people have traveled on, resulting in the robot's optimal path to follow them as well. You could also reverse the polarity of the costs, and increase the value in areas where people often are in order to minimize social disruption.

VII. Discussion

In this paper, we have discussed the benefits of the new layered costmap model over the previous monolithic model. Due to its efficiency and extensibility, an implementation of it has been adopted as the default navigation algorithm for the all released versions of ROS starting with Hydro, the source code for which can be found at github.com/ros-planning/navigation. All of the code for the additional layers can found linked to from wiki.ros.org/costmap_2d. Furthermore, based on the plugin-based layer structure, we hope that the new developments in creating additional costmap rules will be implemented as layers and tested within the ROS navigation framework, allowing for more open exchange of algorithmic behavior and more accessible comparisons between them.

The layered costmap and the associated layers open up the possibility for a wide range of additional robot behaviors. As more layers are assimilated into the planning algorithms, the robots will become more aware of different facets of their environment, and take those contexts into account while navigating. The current state of the practice is just to ignore the additional contexts, or tackle them one at a time. While the layered costmap does enable the contexts to be integrated, we predict that the future challenge will be to find a way to dynamically manage the collections of layers in order to ensure the right contexts are prioritized at the right times. Proxemic behavior dictates that personal space should be respected, but precisely how much less efficient the robot's path should be as a result is an open question. Half of the problem is designing costmap layers such that the mathematically optimal path is the desired distance away. The other half is a social question with unclear answers, of how to balance the needs of robots against the needs of people. While we can offer no concrete answers to such a question, we believe that having a highly customizable data structure for customizing robot behavior will make answering such a question much easier.

References

[1] L. Matthies and A. Elfes, "Integration of sonar and stereo range data using a grid-based representation," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1988, pp. 727–733.

[2] H. P. Moravec, "Sensor fusion in certainty grids for mobile robots," *AI magazine*, vol. 9, no. 2, pp. 61–74, 1988.

[3] K. Konolige, "Improved occupancy grids for map building," *Autonomous Robots*, vol. 4, no. 4, pp. 351–367, 1997.

[4] S. Thrun, "Learning occupancy grids with forward models," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2001, pp. 1676–1681.

[5] D. Ferguson and M. Likhachev, "Efficiently using cost maps for planning complex maneuvers," in *Proceedings of the ICRA 2008 Workshop on Planning With Costmaps*, 2008.

[6] B. P. Gerkey and M. Agrawal, "Break on through: Tunnel-based exploration to learn about outdoor terrain," in *ICRA Workshop on Path Planning on Costmaps, May 2008*, Pasadena, California, 2008.

[7] E. Sisbot, L. Marin-Urias, R. Alami, and T. Simeon, "A human aware mobile robot motion planner," *IEEE Transactions on Robotics*, vol. 23, no. 5, pp. 874–883, 2007.

[8] R. Kirby, R. Simmons, and J. Forlizzi, "COMPANION: A Constraint-Optimizing Method for Person-Acceptable Navigation," in *Proceedings of the 18th IEEE Symposium on Robot and Human Interactive Communication (Ro-Man)*, Toyama, Japan, 2009, pp. 607–612.

[9] M. Svenstrup, S. Tranberg, H. Andersen, and T. Bak, "Pose estimation and adaptive robot behaviour for human-robot interaction," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009, pp. 3571–3576.

[10] L. Scandolo and T. Fraichard, "An anthropomorphic navigation scheme for dynamic scenarios," in *Proceedings of the IEEE Internation Conference on Robtoics and Automation (ICRA)*, 2011, pp. 809–814.

[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, Kobe, Japan, 2009.

[12] E. Marder-Eppstein, E. Berger, T. Foote, B. P. Gerkey, and K. Konolige, "The Office Marathon: Robust Navigation in an Indoor Office Environment," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Anchorage, Alaska, 2010.

[13] D. V. Lu, D. B. Allan, and W. D. Smart, "Tuning Cost Functions for Social Navigation," in *Proceedings of the International Conference on Social Robotics (ICSR)*, 2013.

[14] D. V. Lu and W. D. Smart, "Towards More Efficient Navigation for Robots and Humans," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.