An Interpolated Dynamic Navigation Function

Roland Philippsen and Roland Siegwart
Autonomous Systems Lab, EPFL-I2S-ASL, Station 9
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
roland.philippsen@gmx.net

***Abstract*— The E* algorithm is a path planning method capable of dynamic replanning and user-configurable path cost interpolation. It calculates a navigation function as a sampling of an underlying smooth goal distance that takes into account a continuous notion of risk that can be controlled in a fine-grained manner. E* results in more appropriate paths during gradient descent. Dynamic replanning means that changes in the environment model can be repaired to avoid the expenses of complete replanning. This helps compensating for the increased computational effort required for interpolation.**

We present the theoretical basis and a working implementation, as well as measurements of the algorithm's precision, topological correctness, and computational effort.

***Index Terms*— Mobile robot path planning, dynamic replanning**

I. Introduction

A. The Need for Smooth Dynamic Planning

During work on planning in highly cluttered dynamic environments such as mass exhibitions [7], [13], the need arose for a path planner that is capable of taking into account a continuous risk measure, defined on regions that are neither static nor known beforehand. The planner must produce topologically correct and smooth paths that trade off collision risk against expected path length. Changes to the environment model are frequent and the planner should be able to efficiently adapt existing plans.

Here, we treat planning in highly cluttered dynamic environments as a weighted region path planning problem [12], [14], but in this context the regions can not be pre-determined because they depend on movement in the environment. A more flexible environment representation such as a grid-based risk map is more appropriate.

B. Planning with Environment Dynamics

Among the published research that incorporates environment dynamics during path planning, we cite some examples of related work [1], [2], [4]–[6], [10]. However, these approaches are not appropriate for an application in highly cluttered dynamic environments. They either rely on information and computational resources that are not available for such unpredictable settings (extending $\mathcal{C}$ to a full-fledged state-time representation [5], a velocity space [4], [10], essentially off-line simulations [2]), or are limited to constant velocity models [6] (however, the latter is expected to adapt rapidly to changes in the motion model). In [1], environment dynamics are treated using worst-case scenarios that take into account the sensor capacities, but it treats all known obstacle information as static during planning.

II. Approach

A. Distance Maps and Wavefront Propagation

Mobile robot path planning approaches can be divided into five classes [11]. Roadmap methods extract a network representation of the environment and then apply graph search to find a path. Exact cell decomposition methods construct non-overlapping regions that cover free space and encode cell connectivity in a graph. Approximate cell decomposition is similar, but cells are of predefined shape (e.g. rectangles) and do not exactly cover free space. Potential field methods differ from the other four in that they treat the robot as a point evolving under the influence of forces that attract it to the goal while pushing it from obstacles. Navigation functions are commonly considered a special case of potential fields. We adopt a different stance which is key to formulating the E* algorithm.

A grid is an approximate cell decomposition. Calculating the navigation function is like graph search: It starts at the goal location(s) and propagates through the grid until a path is found. By gradient descent, the robot monotonically reduces the "height" of its location on the grid just as it decreases the distance to the goal. Thus, we interpret navigation functions as a *sampling of an underlying distance function*.

Purely graph-based planners have drawbacks due to their discontinuous representation of workspace $\mathcal{W}$ or configuration space $\mathcal{C}$. Potential field methods produce smoother paths and can be expressed in terms of sensor readings, but they are flawed by the existence of local minima. Graph-based planners usually require replanning if the underlying environment model changes. This drawback is addressed by the D* algorithm [17], [18], which minimizes the replanning cost by recalculating the path only where necessary.

Imagine a continuous contour to sweep outward from the goal throughout the environment (see figure 1), and record when it crosses each cell. The crossing time divided by the speed yields a distance. The main insight comes from turning the problem around: Consider the crossing times at the nodes as samples of an underlying continuous navigation function, instead of extending a discretely defined distance function into the continuous domain.

In order to use this approach for path planning, the propagation speed is made to depend on position. By

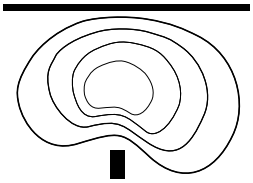

Fig. 1. Continuous domain wavefront formulation. A contour sweeps outward from the goal throughout the environment, taking into account obstacle information.

adding a time axis, the surface traced by the evolving curve becomes a navigation function. The gradient method [9], as well as [3] and [19] take similar stances.

Such continuous formulations have been treated in fields such as fluid mechanics or computer vision. The large body of work on Level Set Methods [15] provides a theoretical foundation for robustly interpolating the crossing time.

B. The Fast Marching Level Set Method

As the Level Set Method (LSM) is not commonly known in the field of mobile robotics, this section serves as introduction. It also introduces the concepts of *upwind property* and the *Fast Marching Method* necessary for understanding this paper. This section is based on chapters 1, 2, and 9 of [16].

The so-called Lagrangian formulation to describing an evolving curve implies parameterizing the curve, deriving its normal vector, and moving discrete points of the curve along this normal. This approach presents serious drawbacks [16]. A solution lies in using a Eulerian formulation, which adds a dimension to the problem and then treats the wavefront as the intersection between a graph[1] and the zero-level of the additional dimension. This is illustrated in figure 2 and equation (1).

$$\begin{cases} \Gamma(t): & \text{closed } (N-1)\text{D surface} \\ \Phi(\vec{x},t): & \mathbb{R}^N \to \mathbb{R} \\ t_0: & \Phi(\vec{x},t=0) = \pm d(\vec{x},\Gamma(t=0)) \\ \Rightarrow & \Gamma(t) = \{\vec{x} \mid \Phi(\vec{x},t)=0\} \end{cases} \tag{1}$$

where Γ denotes the wavefront, N is the supporting dimension, Φ is the graph that is intersected with the zero level to yield Γ, the line labelled t_0 indicates that Φ is initialized to the signed distance from the initial wavefront, and the last line formalizes the way in which Γ and Φ are related.

The advantage of adding the extra dimension is that topology changes can now occur without special treatment, and that numerically stable methods are available for solving the differential equation that describes the front's evolution. This is illustrated in the inset of figure 2 and equation (2).

$$\frac{\partial \Phi}{\partial t} + F|\nabla \Phi| = 0 \tag{2}$$

[1]In this section, *graph* is used in the sense of a function's values throughout its domain, as opposed to the notion of nodes connected by edges.

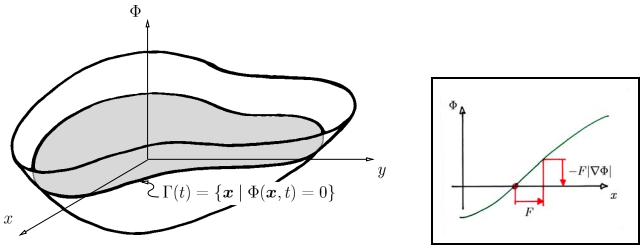

Fig. 2. In the Eulerian perspective, the wavefront is interpreted as the intersection between a graph and the zero-level of an additional dimension Φ. The inset shows the one-dimensional case of the Eulerian formulation to clarify how equation (2) describes the wavefront's evolution: In order to make the intersection move towards the right with speed F, the whole curve Φ has to move downwards with speed $F|\nabla\Phi|$.

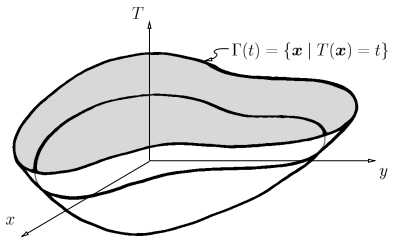

Fig. 3. The Eikonal case leads to a simplified formulation of the Level Set Method that can be solved very efficiently. Notation has changed from Φ to T.

The Level Set Method can be summarized as follows: Convert the initial wavefront $\Gamma(t=0)$ into a graph $\Phi(\vec{x}, t=0)$ by taking the signed distance from $\vec{x}$ to the initial front; repeatedly solve equation (2) using a fixed timestep; determine the front's evolution $\Gamma(t)$ by intersecting $\Phi(t)$ with the zero level. This requires a discrete approximation of the gradient operator ∇, as well as some other heuristics to make it efficient and stable.

The *Eikonal* case is applicable when the propagation speed is always positive (or negative) and depends on position only, i.e. the path planning problem we want to solve. The *Fast Marching Method* can then be applied: It treats Φ as a crossing-time map and $\Gamma(t)$ is now by intersecting Φ with the level of height t. In order to stress this change of interpretation, a notational change is introduced: Φ becomes T. Figure 3 and equation (3) illustrate the Eikonal case.

$$\begin{cases} F = F(\vec{x}) > 0 \\ \Gamma(t) = \{\vec{x} \mid T(\vec{x}) = t\} \\ |\nabla T| F = 1 \end{cases} \tag{3}$$

where F denotes the propagation speed, the wavefront Γ is the intersection between the crossing-time map T and a given instant t, and the simplified differential equation that has to be solved is given in the last line.

In the Eikonal case, T can be built outward starting at $T=0$. This is due to the *upwind property*: A given location is traversed only once by the wavefront. Given that the LSM is calculated on a grid, a discrete notation is introduced. Equation (4) is obtained by applying a first-order gradient approximation to (3).

$$\max(D_{ij}^{-x}T_{ij}^{n+1},0)^2+\ldots \\ +\min(D_{ij}^{+y}T_{ij}^{n+1},0)^2=\frac{1}{F_{ij}^2} \quad (4)$$

where D_{ij}^{-x} denotes the finite difference along negative x at grid point (i,j), T_{ij}^{n+1} is the crossing time at (i,j) for the next step, and F_{ij} is the propagation speed at (i,j).

III. Overview of the E* Framework

A. Abstractions

Several abstractions make E* generic. The *interpolation kernel* makes E* independent of the choice of approximation scheme and *wavefront propagation* is incorporated as a generic *event*-based process. The environment and navigation function is stored in a *grid* of *cells*. E* relies on other processes to keep the environment model up to date, which use a high-level interface to communicate with it[2]. Environmental information is stored in a cell's *meta information*, which is an encoding of the *traversal risk*. Whereas the risk is normalized to $[0,1]$, the meta information is a kernel-dependent mapping of the risk.

B. Backpointers

An important aspect of the LSM is the upwind property. In D*, the upwind property is traced using *backpointers* needed for dynamic replanning: They store on which neighbor a cell's path cost depends, such that all descendants of a location can be visited if that location's environmental information changes. E* extends backpointers to an ordered set. See section V-C for more details on this seemingly straightforward extension.

C. Generic Interpolation Kernels

An interpolation kernel is a function that calculates a given cell's crossing time T_i. Cells c_i belong to a given domain C, each c_i has a set of neighbors N_i and stores environment properties as meta information F_i (5). The cell's propagator set $P_i \subseteq N_i$ and meta information F_i influence the kernel (6). Each cell has a set of backpointers $B_i \subseteq P_i$. The generic interpolation kernel is denoted k.

$$c_i \in C,\ N_i \subset C,\ c_i \notin N_i,\ c_1 \in N_2 \Leftrightarrow c_2 \in N_1 \quad (5)$$

$$F_i \geq 0,\ T_i \geq 0,\ \{T_i, B_i\} = k(F_i, P \subseteq N_i) \quad (6)$$

The remainder of this paper treats interpolating the wavefront between *two* cells. As a consequence, a special case of the above formulation is used: $P = \{c_1, c_2\}$, $\{T_i, B_i\} = k(F_i, c_1, c_2)$.

At first sight, the propagators are the same as the backpointers. However, the kernel might fail to provide valid interpolation for certain combinations, and the use of *fallback* or degenerate solutions is required in these cases, and this information has to be passed to E* such that the backpointers reflect the actual dependency between cells (see section V-C). This is why the kernel provides not only the updated value, but also backpointer information needed to trace propagation direction during dynamic replanning.

[2]Describing this high-level interface is outside the scope of this paper, but note that it provides a convenient way of using E* as a global planner. Details can be found in [13].

D. The Wavefront as Event Queue

An *event* encapsulates the elemental propagation step. Events are stored in a queue that ensures upwind propagation order. A *queue key* is assigned to each event, and the queue is sorted by ascending key. Three types of events are used to implement generic wavefront propagation: `LowerEvent` for path cost decreases, `RaiseEvent` for path cost increases, and `RetryEvent` for attempting to decrease a path cost after a wake of `RaiseEvents` has swept an area.

The queue key corresponds to the upper bound on the optimal path cost at which the event must be triggered in order to respect the upwind property. The upper bound on optimal path costs is a value maintained by the wavefront, all cells with values at or below this bound are valid (they can not be influenced by further event propagations).

IV. Two Kernels

We present two interpolation kernels which will be used in the evaluation of E* in section VI.

A. Graph Distance: NF1 Kernel

The simplest kernel is one that uses only one propagator and does not interpolate at all. This is useful to quantify the effects of interpolation. Update equation (7) and environment representation (8) are fairly straightforward for this kernel.

$$\begin{aligned} T_0 &= \min_{c\in N_0}(T_c + h + F_0) \\ B_0 &= \arg\min_{c\in N_0}(T_c + h + F_0) \end{aligned} \quad (7)$$

$$F_i = \begin{cases} 0 & \Leftarrow c_i \in \text{free space} \\ \infty & \Leftarrow c_i \in \text{C-space obstacle} \end{cases} \quad (8)$$

where T_0 is the value after propagation, h is the grid resolution, F_0 is the meta information of the cell which is being updated, and N_0 is the set of its neighbors. Here, meta information is treated as the additional cost of going from a cell to the one that is being updated. A risk of 0 corresponds to $F_i = 0$, a risk of 1 implies $F_i = \infty$.

B. Gradient Approximation: LSM Kernel

Gradient approximation refers to an implementation of the first-order upwind interpolation scheme for Fast Marching Methods presented in [8], resulting in (9).

$$\max(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0)^2 + \\ \max(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0)^2 = 1/F_{ij}^2 \quad (9)$$

where D_{ij}^{-x} is the finite difference operator along negative x at the grid point (i,j), and F_{ij} is the (known) propagation

speed at (i, j). T corresponds to the navigation function that is to be calculated.

Developing $D_{ij}^{\pm\{x,y\}}T$ leads to a quadratic equation with coefficients that take values based on a switch on the sign and magnitude of the finite difference operators. A geometric interpretation of (9) is useful to determine the propagator set that will yield the optimal solution *prior* to interpolating. The inset in figure 4 shows the situation, following the development in [8]. The cell in the center is being updated. Interpolation implies using up to two neighbors, which need to lie on different axes. Without loss of generality, it can be assumed that the two neighbors leading to the best interpolation are **A** and **C**, and that $T_\mathrm{A} \leq T_\mathrm{C}$. The update equation becomes (10).

$$\begin{aligned}&(T - T_\mathrm{A})^2 + (T - T_\mathrm{C})^2 = h^2/F^2 \\ &\quad\Leftrightarrow \begin{cases} T = t_\mathrm{A} = t_\mathrm{C} \\ (t_\mathrm{A} - T_\mathrm{A})^2 + (t_\mathrm{C} - T_\mathrm{C})^2 = h^2/F^2 \end{cases}\end{aligned} \tag{10}$$

where T is to be determined, T_A and T_C are the values of the best neighbors, h is the distance between two neighbors, and F is the propagation speed at (i, j).

The novel geometrical interpretation is based on introducing two parameters t_C and t_A that are interpreted as the axes of a Cartesian coordinate frame. The solutions for (10) are found at the intersections between the diagonal $t_\mathrm{A} = t_\mathrm{C}$ and a circle of radius h/F centered at $(T_\mathrm{C}, T_\mathrm{A})$.

The switch expressions surrounding $D_{ij}^{\pm\{x,y\}}T$ in (9) lead to constraints that need to be added to (10): Either it has a real solution T with $T > T_\mathrm{C}$, or a real solution to the degenerate form (11) with $T_\mathrm{A} < T \leq T_\mathrm{C}$. The degenerate (fallback) solution is equivalent to finding the intersection between a horizontal line $t_\mathrm{A} = T_\mathrm{A} + h/F$ and the diagonal $t_\mathrm{A} = t_\mathrm{C}$.

$$(T - T_\mathrm{A})^2 = \frac{h^2}{F^2} \;\Leftrightarrow\; \begin{cases} T = t_\mathrm{A} = t_\mathrm{C} \\ t_\mathrm{A} = T_\mathrm{A} + h/F \end{cases} \tag{11}$$

Figure 4 shows the overall geometrical interpretation for a given $(T_\mathrm{C}, T_\mathrm{A})$. Equation (10) has to be solved only if the point $(T_\mathrm{C}, T_\mathrm{A} + h/F)$ lies above $t_\mathrm{C} = t_\mathrm{A}$ (i.e. the *interpolating* curve in figure 4), and that only the higher of the two intersections has to be found. The final equation is (12).

$$\begin{aligned}T = &\begin{cases} T_\mathrm{A} + h/F & \Leftarrow\; T_\mathrm{C} - T_\mathrm{A} \geq h/F \\ \frac{1}{2}\left(-\beta + \sqrt{\beta^2 - 4\gamma}\right) & \text{otherwise} \end{cases} \\ \text{where} &\begin{cases} \beta = -\left(T_\mathrm{A} + T_\mathrm{C}\right) \\ \gamma = \frac{1}{2}\left(T_\mathrm{A}^2 + T_\mathrm{C}^2 - h^2/F^2\right) \end{cases}\end{aligned} \tag{12}$$

recall that $T_\mathrm{A} \leq T_\mathrm{C}$ and note that $F \to 0 \;\Rightarrow\; T \to \infty$. Also note that cells on the border of the grid might not have neighbors of type **A** and **C**, in which case the fallback solution is used.

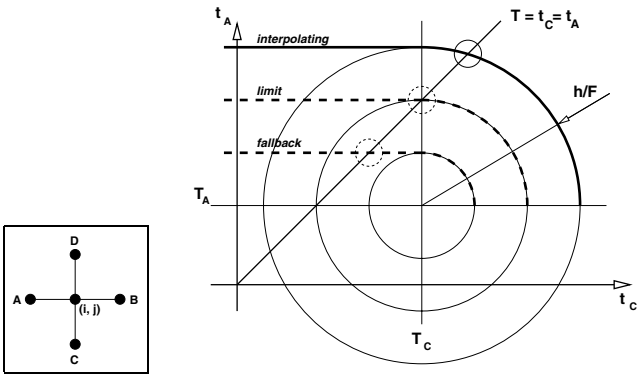

Fig. 4. Geometric interpretation of LSM interpolation, the inset shows the cell neighborhood. Equations (10) and (11) can be read as finding the intersection between the line $t_\mathrm{C} = t_\mathrm{A}$ and the curve labelled *interpolating* (thick solid line). Two dashed curves illustrate how the interpolation behaves when h/F becomes smaller (*limit* and *fallback* curves). The small solid circle indicates the intersection that serves as solution for the interpolating case, and the small dashed circles show the same for the limit and fallback cases.

The LSM kernel maps zero risk to the maximum propagation speed (we use $F_{ij,\max} = 1$), and the maximum risk implies $F_{ij} = 0$.

V. Implementation Details

A. Propagating an Event

`LowerEvents` are sent to all neighbors of a cell whose value decreases, e.g. when a previously blocked passage is discovered to be open. The value *after* the decrease is used as queue key: The lower event was triggered because the highest known optimal path cost reached its target cell, lowering that cell's value implies the same decrease for the path cost bound. Propagating a lower event means calculating the best possible path cost estimate and updating the backpointers accordingly. Then, lower events are sent to all neighbors whose value lies above the current one.

`RaiseEvents` are created when a cell's value increases, e.g. due to a previously undetected obstacle across the planned path. It is sent to all neighbors that have a backpointer to the cell, using the value *before* increase as queue key: It indicates that path costs higher than the cell's old value are now non-optimal, which is required such that the neighbors with backpointers to the just-updated cell get propagated next. When a raise event is propagated, the destination's value is set to infinity[3] and its backpointers set to null. Then, raise events are sent to all concerned neighbors in order to propagate the information upward. Finally, a `RetryEvent` is triggered on the cell that has just been increased, with a queue key determined to allow subsequent path cost decreases as soon as possible.

A `RetryEvent` is the same as a `LowerEvent`, except for its higher priority (see next section). It acts as a sort of rear guard trailing behind wakes of `RaiseEvents` to prevent backpointer loops and a "back wash" phenomenon

[3]The implementation of ∞ is a finite value much higher than the maximum meaningful accumulated path cost.

(a duplicate raise wake going in the opposite direction of the original one).

B. Resolving Event Conflicts

During insertion of events into the queue, attention has to be paid to not overwrite existing ones. Events win by priority, or by their queue key in a tie. Priorities reflect the importance of events: `RetryEvent` > `RaiseEvent` > `LowerEvent`. Raise events are considered more important than lower events, because missing an existing shortcut is less critical than trying to go through a region that is known to be of high risk. Also note that raise events trigger retry events after they have been propagated: Retry events have the same effect as lower events, in other words a lower event is only delayed by the higher priorities of raise and retry events. Overriding raise by retry events is consistent because the latter are only triggered after a raise event has been propagated (the cell in question has an infinite value anyways, it cannot be raised further).

C. Ensuring Backpointer Consistency

There is a fundamental difference between `RaiseEvent` propagation in E* and RAISE state calculations in D*. The latter does not set the node's path cost to infinity, but calculates the usual path cost propagation. This is consistent with the single backpointers used in D*, but interpolation and the multiple backpointers needed to trace cell dependencies raise the following problem in E*: Suppose a cell receives a raise event from one of its neighbors. If we now recalculate the interpolation, this would likely result in a change of backpointers because the previously good propagator is now at a higher path cost. However, the backpointers are needed to propagate a path cost increase to *all* descendants of a location. Changing a backpointer while this chain has not been completed violates the upwind property.

These issues are addressed by setting a cell's value to infinity when a raise event is propagated, and passing on the raise wake to the *unmodified* backpointers. Now that the cell is at infinity, the backpointers become useless (the cell cannot be raised further in any case), which is why they are set to null to avoid creating spurious raise events that would result in no change of the navigation function. In any case, after a raise event, a cell will be subject to a retry event which applies the interpolation and sets new backpointers.

VI. Evaluation and Performance

In order to verify that E* yields useful results, it is necessary to test the consistency of the framework and its ability to incorporate various interpolation kernels. We compare E* with ground truth distances in completely known environments to verify that the distance to the goal is appropriate and evaluate the precision of different kernels. Then we evaluate the consistency of dynamic replanning: Does it produce the same results as reinitializing and replanning the whole grid? Dynamic replanning is supposed to make it more efficient to change the environment model after a navigation function has been calculated. How much work can actually be saved?

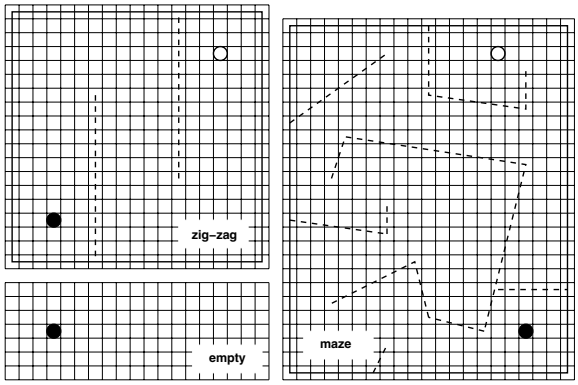

Fig. 5. Simulation setup for evaluating E*– solid lines denote a-priori known obstacles, dotted lines indicate obstacles that need to be discovered by the robot during its movement. The empty circle is the start, and the solid circle the goal. The empty environment is used for determining in which way the initial goal radius influences the convergence towards the true Euclidean distance. The zig-zag and maze environments are used to determine the gain from dynamic replanning and to compare the path smoothness produced by different interpolation methods.
Actual cell sizes vary in the experiments. The grid resolution shown here is very coarse in order to illustrate how the grid is placed onto the environment.

The first suite of tests (ground truth comparison) is run in environments that are empty or have a zig-zag hallway (left of figure 5). Consistency and performance of dynamic replanning are measured by simulating a robot's movement through an environment with unknown obstacles that are discovered when they enter the sensor range of the vehicle (the "zig-zag" and "maze" maps in figure 5).

The goal region is a circle, all cells within this region are initialized to the Euclidean distance to the goal point. All cells outside the goal are initialized to infinity, except the cells just outside the border of the goal which are assigned lower events. Using larger goal radii increases the number of initial cells and gives the interpolation a better starting condition. In the extreme case, only one cell is in the goal region, and the kernels are forced to start out with fallback solutions.

Note that for all these experiments, obstacle information is binary: A cell is either in free space (risk=0) or occupied (risk=1). Also, obstacles are not grown to the robot radius (the robot is considered a point). These simplifications are acceptable for evaluating the general characteristics of E*. A more elaborate planning approach that uses continuous risks, realistic robot sizes, and buffer zones around obstacles is presented in [13].

A. Precision

The relative error of the propagation result is measured to investigate how the kernel, the resolution, and the goal radius influence the relative error (13).

$$e_{\mathrm{c}} = \frac{T_{\mathrm{c}} - d_{\mathrm{c}}}{d_{\mathrm{c}}} \tag{13}$$

where T_{c} is the value of the navigation function at cell c

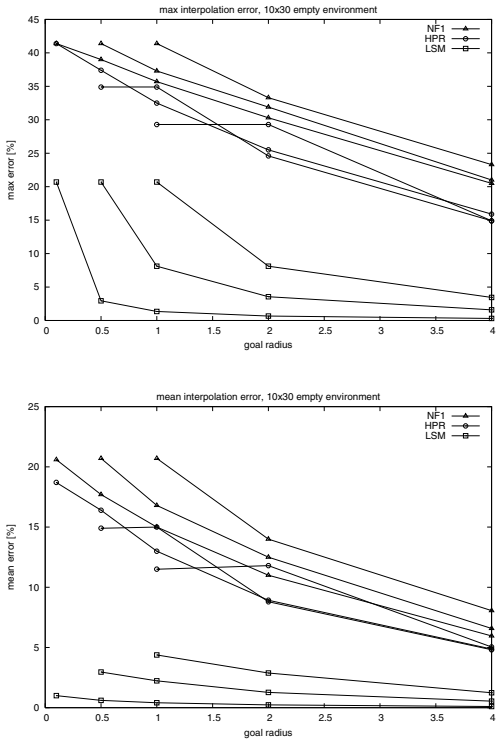

Fig. 6. These plots summarize the relative error between propagation result and true distance in an empty environment of 10×30. The top graph shows the maximum error e_c in function of the interpolation method and the goal radius. The bottom graph shows the mean error. It can be seen that LSM performs better than HPR (another kernel not presented in this paper), which performs slightly better than NF1. The three lines per interpolation method correspond to the three cell sizes: 1 at the top, 0.5 in the middle, and 0.1 at the bottom. Note the case where the cell size equals the goal radius, it illustrates the maximum error because the interpolation can not take advantage of a smoothly initialized goal region.

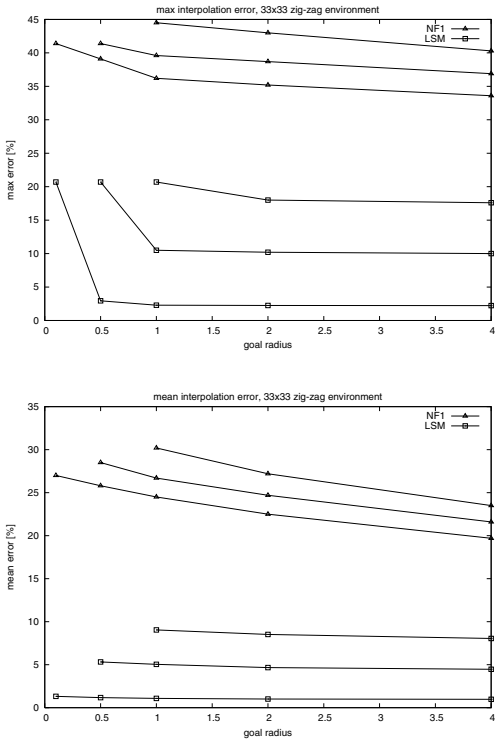

Fig. 7. Relative error of E* in a 33×33 zig-zag environment. As in figure 6, the top graph shows the maximum error and the bottom graph shows the mean error. The overall heightened error with respect to the empty environment is due to the difference between ground truth calculation (which can use the exact endings of the interior walls) and the propagation (which is forced to go through the first non-occupied cell near the end of each wall).

and d_c is the true distance from c to the goal. Figure 6 shows the maximum and mean values of e_c.

None of the kernels underestimates the distance to the goal[4]. All improve e_c when increasing the ratio of goal radius over cell size. Note that the first run in each series of a given cell size is initialized using a single goal cell and thus indicates the effects of fallback solutions. Running these ground truth comparisons in the zig-zag environment indicates that the wavefront is capable of "going around corners". The results are given in figure 7.

B. Computational Effort

How much work can be saved by dynamic replanning? This depends on the environment and how many changes to the meta information occur before the robot reaches the goal. We thus simulate a simple point robot following the negated gradient with bounded acceleration and speed. The robot is equipped with a sensor of limited range. When a previously unknown obstacle is detected, propagation is performed until the highest known optimal path cost lies above the value at the robot's position. Computational complexity is measured by counting the number of events that are propagated until the robot can resume its movement. Simulations are run with and without interpolation. In the latter case, the method requires fewer events because there is only one backpointer. The LSM kernel requires up to two backpointers, so raise events are propagated to up to twice as many descendants than for the NF1 kernel.

Table I compares the performance with and without interpolation. The gain is calculated without taking into account the initial planning. The meanings of the line labels in these tables are: **cell size** is the grid spacing h e.g. in (10); **N° dyn. propagations** is the event propagation count over the whole run when using dynamic replanning; **N° replan prop.** is the event count over the whole run when using complete replanning; **gain** is the relative reduction in the number of events when using dynamic replanning. It is calculated as $(n_{\text{complete}} - n_{\text{dynamic}})/n_{\text{complete}}$.

The cost of interpolation improvement stems from two aspects: An increase of computational complexity due to the higher number of backpointers, and an increase in the duration of each operation due to the more elaborate calculations required for interpolation.

Table II compares the propagation counts. The theoretical increase is twofold (twice as many backpointers), however the values observed in the zig-zag and maze environments lie between a 25% and 62% increase. $\rho = n_{\text{dynamic}}/n_{\text{complete}}$ is a normalized measure of the number of

[4]At least not to within an error of 10^{-14} which is considered to be due to numerical effects

NF1 kernel in zig-zag map			
cell size	0.67	0.37	0.20
N° dyn. propagations	1'242	3'926	13'627
N° replan prop.	2'209	7'279	26'226
gain	**43.8** %	**46.1** %	**48.9** %

LSM kernel in zig-zag map			
cell size	0.67	0.37	0.20
N° dyn. propagations	1'759	6'417	20'260
N° replan prop.	2'243	7'337	26'192
gain	**21.6** %	**12.5** %	**22.6** %

NF1 kernel in maze map			
cell size	0.71	0.38	0.20
N° dyn. propagations	3'521	11'890	43'836
N° replan prop.	6'881	25'463	95'075
gain	**48.8** %	**53.3** %	**53.9** %

LSM kernel in maze map			
cell size	0.71	0.38	0.20
N° dyn. propagations	5'464	18'887	74'880
N° replan prop.	8'565	26'880	116'271
gain	**36.2** %	**29.7** %	**35.6** %

TABLE I

E* PERFORMANCE WITH AND WITHOUT INTERPOLATION ON A 20×20 ZIG-ZAG AND A 20×25 MAZE MAP. THE GAIN FROM DYNAMIC OVER COMPLETE REPLANNING IS SLIGHTLY WORSE WHEN INTERPOLATING. THE DEPENDENCY ON CELL SIZE IS NOT SIGNIFICANT

Relative computational complexity / zig-zag			
cell size	0.71	0.38	0.20
relative burden ρ(NF1)	0.562	0.539	0.520
relative burden ρ(LSM)	0.784	0.875	0.774
ρ(LSM)/ρ(NF1)	1.39	1.62	1.49

Relative computational complexity / maze			
cell size	0.71	0.38	0.20
relative burden ρ(NF1)	0.512	0.467	0.461
relative burden ρ(LSM)	0.638	0.703	0.644
ρ(LSM)/ρ(NF1)	1.25	1.50	1.40

TABLE II

E* COMPUTATIONAL COMPLEXITY WITH AND WITHOUT INTERPOLATION. THESE NUMBERS ARE PROPAGATION COUNTS THAT INDICATE THE RELATIVE COMPLEXITY OF INTERPOLATION.

Relative and absolute operation cost					
		System A		System B	
		dbg	opt	dbg	opt
	overhead [ns]	33.0	31.3	11.3	10.9
	N° calls	513'939	513'971	513'939	513'971
NF1	kernel [ns]	33.7	31.6	12.8	11.0
	optimum [ns]	225	181	77.2	67.0
LSM	kernel [ns]	46.3	34.6	15.4	12.1
	optimum [ns]	273	185	89.8	67.1
ratio	**kernel**	**1.37**	**1.09**	**1.21**	**1.10**
	optimum	**1.21**	**1.03**	**1.16**	**1.00**

TABLE III

E* OPERATION COST WITH AND WITHOUT INTERPOLATION IN MAZE ENVIRONMENT (20×25, CELL SIZE 0.2). THE OPERATIONS OF THE LSM KERNEL ARE UP TO 37% MORE EXPENSIVE THAN THE NON-INTERPOLATING ONES.

dynamic replanning events. $\rho(\text{LSM})/\rho(\text{NF1})$ indicates how much wider raise events spread when using interpolation.

Table III presents operation delays. The times for single kernel calculations and finding the optimum (lowest) interpolation for a cell were measured under desktop system load. System A is a 466MHz Intel Celeron running linux-2.6.7, system B is a 1.8GHz Intel Pentium 4 running linux-2.4.22. The measurements were performed with debug (dbg) and optimized (opt) executables produced with the `-g` and `-O3` flags of GCC-3.3.

The propagation count depends on the environment due to the backpointers' dependency on events which in turn depend on when the robot discovers which parts of the environment, whereas the operation delay is concerned with the operation of the kernels and depends on the platforms and optimizations used to execute the program. The complexity increases by up to 62% in the studied settings. The operation cost for a single calculation of the interpolation kernel is up to 37% higher[5].

C. Path Smoothness

Figure 8 shows the paths produced with and without interpolation. On the left, the preference for displacements along coordinate axes and the two diagonals illustrates the grid-based distance measure of NF1, whereas the paths produced using LSM are very close to the line-of-sight towards the edge of a known obstacle or the goal.

Figure 9 illustrates a situation where a passage opens and provides a shortcut to the goal. Then, an obstacle appears that makes navigating the passage more difficult (however, the robot could still fit through). In this case the path switches back to the detour because the accumulated risk of the shortcut is too high. Note that this sequence of navigation functions has been created using a more elaborate approach [13] that uses a user-defined risk transition zone around obstacles to take into account the robot's radius as well as a security distance. This can be seen in the lower right of figure 9, which shows the risk map (color scaled cells) and the state of the wavefront when the last replanning cycle stopped (small dots that go form arcs of circles).

[5]Overall run times are also influenced by the complexity of insertion and removal operations on the event queue, which is $O(n \log n)$ if balanced binary trees are used in the implementation. Such storage overhead applies to all methods that rely on ordered propagation.

VII. CONCLUSION AND OUTLOOK

The E* algorithm allows to calculate and update smooth navigation functions that approximate true distance much better than other grid or graph based methods – by an order of magnitude or more given the right conditions. The additional computational complexity required for this achievement is a factor of two in the theoretical worst case, but experiments suggest a factor of approximately 1.2 to 1.6 in practice. Each of the propagation steps becomes more elaborate as well, this amounts to a factor below 1.4 in the case of the robust LSM interpolation when compared with the NF1 kernel that mimics D*.

In addition to providing an interpolated navigation functions with dynamic replanning, E* is independent of interpolation details and can thus be used to evaluate different kernels in terms of their quality and computational costs.

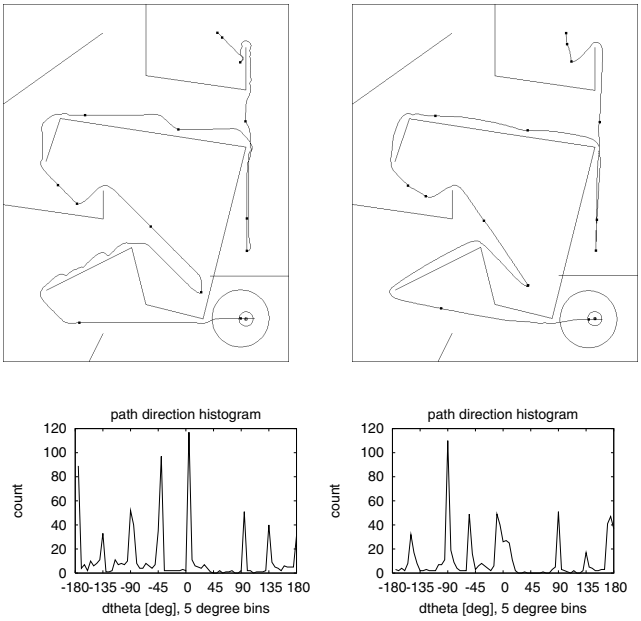

Fig. 8. Example paths from the simulations, the robot starts in the upper right without any knowledge of the environment and proceeds to reach the goal in the lower right, discovering obstacles that come into its sensor range (illustrated by the large circle). The top row shows the final paths: On the left, no interpolation was used and the path shows grid effects. On the right, interpolation has been used to achieve a better distance map. The bottom row shows histograms of the path directions: On the left, you can see that the NF1 prefers directions along multiples of $\pi/4$, whereas there is less preference for these angles in LSM. Note that the environment presents sections where the best path direction lies close to a multiple of $\pi/4$, the peaks in the LSM histogram illustrate this (note in particular the peak which is to the left of -45°).

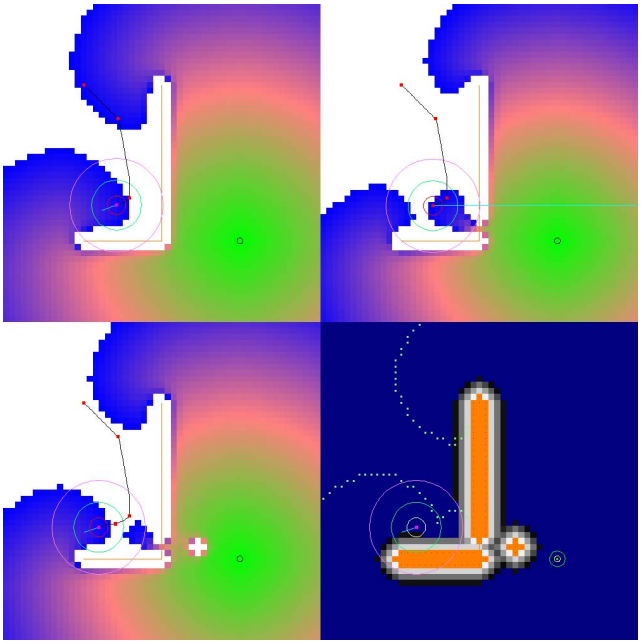

Fig. 9. Obstacle removal and addition with LSM: After the robot has discovered the L-shaped obstacle, it plans a path around it (top left). The top right shows the navigation function after freeing a passage in the obstacle, the optimal path switches topology by going through this opening. Then, an obstacle was added behind the passage, but such that the robot could pass (bottom right shows the risk map). However, the optimal path switches back to the old topology, because the new obstacles causes a higher accumulated risk: The robot takes a detour to avoid the dangerous zone (bottom left).

Generalizing E* to higher dimensions and interpolation orders is possible: Event propagation relies on neighborhood information which can be defined independently of dimension. Extending to non-grid representation is feasible as well: The only part of this work which depends on the grid nature is the LSM interpolation, and this has already been applied to triangulated domains in [8].

Acknowledgment

The authors would like to thank Kurt Konolige for his feedback and pointing out relevant publications.

References

[1] R. Alami, T. Simon, and K. Madhava Krishna. On the influence of sensor capacities and environment dynamics onto collision-free motion plans. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[2] Enrique J. Bernabeu, Josep Tornero, and Masayoshi Tomizuka. Collision prediction and avoidance amidst moving objects for trajectory planning applications. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2001.

[3] Michael S. Branicky and Ravi Hebbar. Fast Marching for Hybrid Control. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, 1999.

[4] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using the relative velocity paradigm. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1993.

[5] Thierry Fraichard. Trajectory planning in a dynamic workspace: a 'state-time space' approach. *Advanced Robotics*, 13(1):75–94, 1999.

[6] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255, 2002.

[7] B. Jensen, R. Philippsen, and R. Siegwart. Motion detection and path planning in dynamic environments. In *Workshop Proceedings Reasoning with Uncertainty in Robotics, International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[8] R. Kimmel and J.A. Sethian. Computing geodesic paths on manifolds. *Proc. Natl. Acad. Sci. USA*, 95(15):8431–8435, July 1998.

[9] Kurt Konolige. A gradient method for realtime robot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2000.

[10] Frederic Large, Sepanta Sekhavat, Zvi Shiller, and Christian Laugier. Towards real-time global motion planning in a dynamic environment using the NLVO concept. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[11] J.-C. Latombe. *Robot motion planning*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1991.

[12] J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: Finding shortest paths trhough a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.

[13] Roland Philippsen. *Motion Planning and Obstacle Avoidance for Mobile Robots in Highly Cluttered Dynamic Environments*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[14] N. C. Rowe and R. S. Alexander. Finding optimal-path maps for path planning across weighted regions. *International Journal of Robotics Research*, 19(2):83–95, 2000.

[15] J. A. Sethian. Evolution, implementation, and application of level set and fast marching methods for advancing fronts. *Journal of Computational Physics*, (169):503–555, 2001.

[16] J.A. Sethian. *Level Set Methods – Evolving interfaces in geometry, fluid mechanics, computer vision, and materials science*. Cambridge University Press, 1996.

[17] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1994.

[18] Anthony Stentz. The focussed D^* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[19] John N. Tsitsiklis. Efficient Algorithms for Globally Optimal Trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, September 1995.