Optimal Path Planning and the Fast Marching Method

J. J. Clark
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Mathematics Commons, and the Statistics and Probability Commons

Recommended Citation
https://digitalcommons.usu.edu/gradreports/1259

This Report is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Plan B and other Reports by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.
Optimal Path Planning and the Fast Marching Method

J. J. Clark

August, 2002

Abstract

The problem of determining an optimal path for an object moving through some obstacle space presents several nontrivial subproblems. The foremost being the computational complexity that is involved and how to best deal with the associated large data volume. For example, a non-symmetric object moving in three dimensions possesses six degrees of freedom. This can lead to a computational grid that may easily be on the order of $10^{12}$. Furthermore, for every point in the computational domain, several complex calculations must be performed. These include performing tests to determine if the object and obstacles intersect, and numerically solving the eikonal equation in multiple dimensions. The latter is accomplished via the Fast Marching Method (FMM), which this report outlines. At the heart of all of these problems is the way in which the configuration of the object is best represented. Thus, due to these and other complications, it is crucial that efficient algorithms are developed and the best possible representations are used to make path planning problems solvable.
Contents

1 Introduction ................................................. 4

2 Problem Overview ........................................... 4
   2.1 Basic Path Planning .................................. 5
   2.2 Configuration Space in \(\mathbb{R}^2\) .................. 5
   2.3 Intersection Tests .................................. 6
   2.4 The Eikonal Equation ................................ 6
      2.4.1 Level Set Methods .............................. 6
   2.5 Determining the Optimal Path ...................... 7
   2.6 A Demonstration in \(\mathbb{R}^2\) ...................... 8

3 The Fast Marching Method ................................ 10
   3.1 The Algorithm ...................................... 10
   3.2 Remarks ............................................ 11

4 The Discretized Eikonal Equation ......................... 12

5 Configurations in \(\mathbb{R}^3\) ................................ 13
   5.1 Rotations in Space ................................ 13
   5.2 Formal Definition .................................. 14
   5.3 Representing the Configuration Space .......... 14
      5.3.1 Parameterizing SO(3) .......................... 14
      5.3.2 Exponential Coordinates ....................... 15
      5.3.3 Quaternions ................................... 16
   5.4 Computational Grids ................................ 16
      5.4.1 A Unit Disk Grid ............................ 16
      5.4.2 The Fat Cube .................................. 19

6 Conclusions and Future Work ............................... 21

A Main FMM Code .............................................. 23
   A.1 Function to compute travel times ................. 24
   A.2 Subroutines ....................................... 25

B Code To Compute Optimal Path ............................. 27
   B.1 Compute gradient ................................. 29

C FMM Code on Unit Disk .................................... 30
   C.1 Function to compute travel times ................. 32
   C.2 Subroutines for disk function ................... 34

D Code to Perform Map to Unit Disk ....................... 36

2
List of Figures

1. Three line obstacles in the region \(0 \leq x, y \leq 10\). ........................................... 9
2. Contour plot of the travel time function. ................................................................. 9
3. Contour plot of the travel time function along with the optimal path from \((1, 1)\) to \((9, 1)\). ................................................................. 10
4. Patched grid of the unit disk. The point \(A\) is associated with grid regions 0, 3, and 4. ................................................................. 17
5. Closer view of the grid point \(A\) along with six possible neighbors labelled \(n_i\), \(i = 1, \ldots, 6\). ................................................................. 18
6. A rectangular version of the unit disk grid shown in Figure 4. ............................... 19
7. Contour plot of \(T(x, y)\) on a rectangular version of the unit disk grid. ............... 20
8. Density plot of \(T(x, y)\) on the unit disk. Times were computed on the rectangular version and then mapped onto the unit disk. Lighter regions correspond to increasing time. ......................................................... 20
1 Introduction

The determination of some path between two points is a problem that comes up in a variety of contexts. For example, the route one takes to work requires a lower level of planning when compared to the more sophisticated motion of a robotic welding arm maneuvering about the piece(s) being welded. Clearly, the latter example can become quite complex, depending on several factors. As a result, it becomes necessary to efficiently determine the movements (i.e., paths) that the robot can make in order to minimize certain costs (e.g., time). This leads us to the goal of this report. We will address the problem of optimal path planning with a mathematical statement of the problem, and provide subsequent efficient solution procedures along with possible further research to be done.

Before we begin, it is helpful to mention what is meant by an “object”. A classic example of path planning is the piano mover’s problem, which is simply the question of whether or not a piano (a non-symmetric object) can be moved from one place to another. Thus in this case, the object is simply a piano. For the most part, our discussion will be based on ideas from robotics, so the terms robot and object are often used interchangeably. It is also important to point out that this paper will in no way provide a thorough treatment of the mathematics associated with robotics, which can be found in [5] and [7]. However, the ideas that are presented should give the reader a good idea of the issues associated with such problems.

In order to obtain a firm handle on the complex issues associated with path planning problems, we will first investigate the general path planning problem in Section 2, where a statement of the problem will be given. In addition, a simple example to demonstrate the procedure will be provided. This will help introduce the various components of path planning problems, including a discussion of the eikonal equation and how this can be solved using the Fast Marching Method that is discussed in Section 3. Here, the algorithm will be given along with a discussion of the subtleties associated with its application. In Section 4 the necessary tools needed to solve the discretized eikonal equation will be developed, where the idea of upwinding will be introduced and how the Fast Marching Method can be applied. The concept of a configuration space will be investigated in Section 5. This includes some of the possible ways to represent such a space, and leads us to the consideration of possible parameterizations for the group of proper rotations, denoted $SO(3)$. Utilization of $SO(3)$ for representing a rotating object is a standard tool in the field of robotic manipulation. After properly introducing $SO(3)$, we turn to the problem of laying a grid on this group in Section 5.4. Finally, we close with some concluding remarks in Section 6.

2 Problem Overview

As is typically the case, much insight can be gained by first considering a “simpler” problem. As a result, this section will frequently use the case of path planning in $\mathbb{R}^2$ as a means of clarifying certain concepts, while attempting to keep the presentation in a general form. Furthermore, path planning in $\mathbb{R}^3$ is considerably more involved, so this will provide us with an opportunity to introduce the basic ideas, and save the details of the problem in $\mathbb{R}^3$ for later sections.
2.1 Basic Path Planning

The following statement is a slight modification of what Latombe [5] refers to as the “basic motion planning problem”. Given an object $R$ and obstacle space $B$, determine a continuous path $\lambda$ that is parameterized by $t \in [0, 1]$ for $R$ that starts at some initial configuration $\lambda(0) = x_0$ and ends at a final configuration $\lambda(1) = x_1$ which satisfies the no interference condition given by

$$T[R; \lambda(t)] \cap B = \emptyset \quad \forall t \in [0, 1],$$

(2.1)

where $T[R; \lambda(t)]$ denotes the translation and rotation of the object $R$ by the amount specified by $\lambda(t)$. In other words, we need to determine some path between two points such that the object does not attempt to “pass through” an obstacle.

2.2 Configuration Space in $\mathbb{R}^2$

Intuitively, an object’s configuration simply tells us about the object’s location and orientation. More precisely, it is the tensor product of ordinary Euclidean space and the group of proper rotations. For example, in $\mathbb{R}^2$ the space domain is usually taken to be the unit square, so that the configuration space $C$ is given by $C = [0, 1] \times [0, 1] \times [0, 2\pi]$ with points $x \in C$ of the form $x = (x, y, \theta)$. Thus, the path planning problem in $\mathbb{R}^2$ leads to a 3-dimensional configuration space (3 degrees of freedom).

**Remark 1.** Referring back to our path $\lambda$, we have that $\lambda$ is in fact a continuous map into the configuration space $C$, (that is, $\lambda: [0, 1] \rightarrow C$).

**Remark 2.** For an object moving in $\mathbb{R}^3$, we have $C = \mathbb{R}^3 \times SO(3)$, where $SO(3)$ is the group of proper rotations in $\mathbb{R}^3$. Thus, an object moving in 3-dimensions contains 6-degrees of freedom (6-DOF).

**Remark 3.** One important fact that will become more apparent when we discuss the configuration space for an object moving in $\mathbb{R}^3$ (see Section 5) is that $C$ gives us a way to simplify our problem. That is, a robot can now be represented as a point $x \in C$, so that planning the motion of some odd-shaped object can be transformed into planning the motion of a point. Obviously, this is a powerful tool.

2.3 Intersection Tests

A significant amount of the work involved with any sort of path planning is the determination of when the object and obstacles collide. In terms of our configuration space, this means that for every point $x \in C$ we must determine if the object and obstacles intersect. Furthermore, the data volume for the 6-DOF problem is often quite large\(^1\), so it is necessary that these intersection tests be performed in an efficient manner.

A method that has proven to work well in $\mathbb{R}^2$ when the object $R$ and obstacles $B$ can be represented as the union of convex polyhedra is a linear programming approach developed by Grandine at Boeing (see [3] for the ideas behind this method). One other possibility proposed by Epton through private communication is based on using a voxelized\(^2\) characteristic function for representing $R$ and $B$, so that the well-developed theory (and efficiency) of the Fast Fourier Transform could be utilized.

\(^{1}\) $C$ may be on the order of $10^{12}$ points.

\(^{2}\) Volumetric pixels—can be thought of as a 3-d version of pixels.
In the process of performing the intersection tests, it will be necessary to define what can be thought of as a velocity function associated with each point in the configuration space. That is, let a point \( x \in C \) be denoted as feasible if it satisfies \( T[R; x] \cap B = \emptyset \), and non-feasible otherwise. Then, define the velocity function \( c(x) \) by

\[
  c(x) = \begin{cases} 
    1 & \text{if } x \text{ is feasible}, \\
    0 & \text{otherwise}.
  \end{cases}
\]

(2.2)

The reason for calling this a velocity function will become more clear in later sections. For now it is sufficient to think of \( c \) as signalling (1 for yes, 0 for no) whether a point is part of an obstacle or not.

### 2.4 The Eikonal Equation

Consider the cost function \( s(x) = 1/c(x) \), where \( c \) is defined in equation (2.2). That is, the cost associated with a feasible point \( x \) will be given unit cost, while non-feasible points have essentially infinite cost. The optimal path \( \lambda(t) \) with \( t \in [0, 1] \) can then be determined by minimizing the integral

\[
  \int_{x_0}^{x_1} s(\lambda(t)) dt,
\]

(2.3)

where \( t \) parameterizes the arc-length of the path so that \( ||\lambda_t|| = 1 \). In other words, we wish to determine a path that minimizes the time it takes to travel between two points. Thus, if \( T(x) \) denotes the minimal time required to travel from the point \( x_0 \) to some other point \( x \), then we have

\[
  T(x) = \min_\lambda \int_{x_0}^{x} s(\lambda(t)) dt.
\]

(2.4)

Furthermore, the set of all points \( x \in \mathbb{R}^n \) that are \( t \) units from \( x_0 \) are given by the curve \( T(x) = t \), and since minimal cost paths will always be orthogonal to the level curves of \( T \) it follows that \( ||\nabla T(x)|| = s(x) \). That is, the path \( \lambda(t) \) that satisfies (2.4) will always be orthogonal to the gradient of \( T \), so that the length of \( \nabla T \) at a point \( x \) will be equal to the cost \( s \) associated with that point. This leads to the eikonal equation given by

\[
  ||\nabla T(x)|| = s(x), \quad T(x_0) = 0.
\]

(2.5)

Solving (2.5) yields travel times from the point \( x_0 \) to all points in the domain of the problem.

#### 2.4.1 Level Set Methods

A more general formulation of the eikonal equation can be obtained via Level Set Methods, which are developed in Osher and Sethian [8]. The underlying theme behind these methods is the modelling of a propagating front, and this is precisely the way in which we can view the evolution of our time function.

By way of derivation, consider a simple example given by Sethian [12]. Suppose we have an initial surface \( S_0 \) given by the unit circle for which \( T(x, y) = 0 \) on \( S_0 \), and we wish to compute how long it will take to reach points outside of the circle. Clearly, if we simply compute a series of concentric circles (i.e., propagate the initial circle outwards), then the
distance from a point \((x, y)\) to \(S_0\) will correspond to travel times. That is, at any future time \(t > 0\), the location of the front \(S\) is given by the set of points

\[ S(t) = \{(x, y) : T(x, y) = t\}, \tag{2.6} \]

where \(T(x, y) = (x^2 + y^2)^{1/2} - 1\) is the travel time function. Equation (2.6) is referred to as a level set equation, since it is an equation for some level curve \(t\) of the function \(T(x, y)\).

The details of these ideas can be found in [13], where more general cases are considered that take into account complicated fronts moving with varying speeds that can depend on several factors (e.g., local curvature and total oscillation).

For our purposes it is only necessary to assume that the front moves along its normal vector field with constant speed. Then, differentiating both sides of \(T(x) = t\) with respect to \(t\) yields

\[ \nabla T \cdot x'(t) = 1. \tag{2.7} \]

The condition that the front moves in a direction normal\(^3\) to itself with speed \(v(x)\) means that

\[ x'(t) = \frac{\nabla T}{\|\nabla T\|} v(x), \tag{2.8} \]

where \(\|\cdot\|\) denotes the usual Euclidian norm. Substituting (2.8) into (2.7) leads to the eikonal equation given by

\[ \|\nabla T(x)\| v(x) = 1, \tag{2.9} \]

with the initial condition that \(T\) is set equal to zero on some initial set of points.

In terms of path planning, we have that the velocity is given by \(v(x) = c(x)\), where \(c\) is defined in (2.2). Thus, the front is allowed to move with unit speed when a point is feasible, and travel times for non-feasible points will be essentially infinite.

It is constructive to consider what is going on from a physical standpoint. If we consider again the simple propagating circle example, then we can imagine that the concentric circles represent a wave moving outward in water. That is, solving the eikonal equation when the initial set is just a single point, is conceptually similar to dropping a pebble in the water. Furthermore, this is just an application of Huygen’s principle, which as we’ll see, is the driving force behind the Fast Marching Method.

### 2.5 Determining the Optimal Path

Once the eikonal equation \(\|\nabla T(x)\| = 1/c(x)\) has been solved (see Section 3) for all points \(x\) in the configuration space \(C\), the optimal path from some initial position and orientation \(x_0 \in C\) to a final configuration \(x_1 \in C\) can be determined by working backwards. That is, the ordinary differential equation

\[ \frac{dx}{dt} = -\nabla T(x), \quad x(0) = x_1, \tag{2.10} \]

is solved until the destination \(x_0\) is reached. This can be seen by considering the function \(T\) as being some potential with \(x_0\) at the bottom. That is, \(x_0\) is a global minimum of the function \(T\), so \(-\nabla T\) gives us the optimal direction in which to follow in order to arrive at \(x_0\).

\(^3\)The unit normal vector \(n\) is given by \(n = \nabla T/\|\nabla T\|\).
2.6 A Demonstration in $\mathbb{R}^2$

Based on our formulation, there are three major components for solving the path planning problem:

1. Given the obstacle space and the object, perform the intersection tests for all points $x \in \mathcal{C}$. In the process, define the velocity function $c(x)$ given by (2.2).

2. Compute travel times $T(x)$ for all points $x \in \mathcal{C}$ by solving the eikonal equation (2.9) via the Fast Marching Method.

3. Solve equation (2.10) for the optimal path.

As an example to demonstrate the process, consider the simple obstacle space given in Figure 1, which contains three 1-dimensional obstacles (lines) in the region $0 \leq x, y \leq 10$. First, let the object be a point, so there is no need to perform intersection tests. Thus, if we take the initial position to be at $(1,1)$, then the first step is to solve the eikonal equation of the form

$$
\|\nabla T(x,y)\| = s(x,y), \quad T(1,1) = 0,
$$

where the function $s = 1/c(x,y)$ is given by

$$
s(x,y) = \begin{cases} 
1 & \text{if } (x,y) \notin B \\
\infty & \text{otherwise}
\end{cases}
$$

Here, the set $B$ includes all of the points where obstacles are located. In addition, for practical computations, $s$ is set equal to some large relative value when obstacles are present (e.g., $s(x,y) = 10^{10} \forall (x,y) \in B$). A contour plot of the travel time function $T(x,y)$, which was computed using the Fast Marching Method (see Section 3), is shown in Figure 2. As can be seen, the obstacles appear wider, but this is a result of the coarse grid that is used.

Once $T(x,y)$ is determined, the optimal path starting at $(1,1)$ to some destination, say $(9,1)$, can be found by solving equation (2.10). However, the way in which the function $s(x,y) = 1/c(x,y)$ was defined, (2.10) will be very stiff. That is, along the obstacles, travel times can vary dramatically, resulting in large gradients. In order to circumvent this problem, we can essentially normalize, and instead solve the equation

$$
\frac{dx}{\mu T_x} = \frac{dy}{\mu T_y}, \quad x(0) = x_1,
$$

until the point $x_0$ is reached.

In order to see that the solution of this equation does in fact produce the desired path, let us rewrite the system in differential form as

$$
dx = \mu T_x(x,y) \ dt, \\
dy = \mu T_y(x,y) \ dt,
$$

where $\mu = 1/\|T(x,y)\|$. Solving both equations in (2.14) for $dt$, and setting the results equal yields

$$
\frac{dx}{\mu T_x} = \frac{dy}{\mu T_y}
$$
Figure 1: Three line obstacles in the region $0 \leq x, y \leq 10$.

Figure 2: Contour plot of the travel time function.
Figure 3: Contour plot of the travel time function along with the optimal path from (1, 1) to (9, 1).

Thus, assuming \( \nabla T \) is smooth enough and not close to 0, the solution is independent of \( \mu \). In fact, this shows that we can scale by any power of \( \|T(x, y)\| \) in our equation.

Referring back to our current example, equation (2.13) takes the form

\[
\begin{bmatrix}
x(t) \\
y(t)
\end{bmatrix}' = \frac{-\nabla T(x, y)}{\|\nabla T(x, y)\|}, \quad (x(0), y(0)) = (9, 1),
\]

which is integrated until the point (1, 1) is reached. This was done using a first order Heun’s method with bilinear interpolation of \( \nabla T(x, y) \) to produce the path shown in Figure 3.

3 The Fast Marching Method

As previously noted, it is vital that efficient numerical schemes be used in order for path planning problems to be solvable from a computational standpoint. Fortunately, the Fast Marching Method, which was developed independently by Tsitsiklis [14] and Sethian [11] in 1995-96, provide us with exactly that: An \( O(N \log(N)) \) algorithm, where \( N \) is the number of grid points. Details of the FMM will not be presented here—(see [1]). However, we will outline the basic algorithm for the two-dimensional problem, and investigate some of the details of applying the FMM.

3.1 The Algorithm

A statement of the algorithm is given first, with some explanations to follow.

Begin by laying down a rectangular grid on the spatial domain, and set \( T_{i,j} = 0 \) on an initial set of grid points \( A_0 \). Here, \( i = 1, \ldots, m \) and \( j = 1, \ldots, n \) denote the location in the
computational domain and \( m \) and \( n \) denote the total number of grid points in the \( x \) and \( y \) directions, respectively. Then, after \( N \) times through the algorithm, we have the following sets:

- \( A_N \) = the set of grid points where travel times are settled.
- \( R_N \) = the set of grid points adjacent to \( A_N \), but not in \( A_N \).
- \( F_N \) = everything else.

A loop of the algorithm then looks like:

1. For all grid points \((i, j) \in R_N\), (re)compute candidate travel times \( T_{i,j}^{\text{cand}} \) by solving the discretized eikonal equation.

2. Remove \((i_{N+1}, j_{N+1}) \in R_N\) with minimum travel time and form the set

   \[ A_{N+1} = A_N \cup \{(i_{N+1}, j_{N+1})\} \]

3. Add all of the neighbors of \((i_{N+1}, j_{N+1})\) that are not currently in \( A_N \) to form the set

   \[ R_{N+1} = R_N \cup \{(i, j) : (i, j) \notin A_N \text{ and } (i, j) \text{ is a neighbor of } (i_{N+1}, j_{N+1})\} \]

4. Repeat until all points in the domain have settled travel times.

### 3.2 Remarks

Let us now make some clarifying comments:

1. For path planning problems, the initial set \( A_0 \) is usually taken to be the initial position, so that \( A_0 = \{(i_0, j_0)\} \), where \((i_0, j_0)\) is the grid point associated with \( x_0 \).

2. By settled travel times, it needs to be made clear that travel times are computed for all points in the neighbor set \( R_N \), but only one of these travel times is kept. This being the point that has the minimum travel time in the set, which is then removed from \( R_N \) and added to \( A_N \) (step 2. of the algorithm). Moreover, this is why points in \( R_N \) are said to have candidate travel times.

3. In general, the grid coordinates of a point in \( R_N \) differ from those of a point in \( A_N \) by 1 in exactly one position.

4. See Section 4 for a discussion on the discretization of the eikonal equation, and how an approximate solution can be computed.

5. It is critical that the point in \( R_N \) with the minimum candidate travel be selected. This ensures that certain monotonicity properties are maintained, which are vital for the algorithm to work effectively. For a more involved discussion of these properties, see Epton and Grandine [1], where precise theorems are given.
6. The algorithm can be made more efficient by utilizing heap-management techniques for determining the minimum time $T_{r \cap n}$ for all $(i, j) \in R_N$. More specifically, this amounts to using a data structure that represents a complete binary tree. See [10] for further details of heap data structures. Details of their application to the FMM are found in [12].

7. Termination of the algorithm will occur when the sets $R_N$ and $F_N$ are empty.

8. Finally, in terms of our path planning problem with obstacles, one might wonder if—or, even how—travel times are computed at points within the obstacles. The answer is yes: Numerically, instead of defining the velocity function (2.2), we define the slowness function $s(x) = \frac{1}{c(x)}$, with $s$ set equal to some large relative value (e.g. $10^{10}$) when $x$ is non-feasible. This ensures that the problem is well-posed, and that travel times are computed for all points in the domain.

Clearly, there are many details that can be further explored, but the general idea of how the algorithm works should be apparent: March the expanding front outwards with increasing time, where each iteration through the loop is an application of Huygen’s principle to determine which grid point the front shall pass through next.

4 The Discretized Eikonal Equation

The Eikonal Equation (2.9) is a nonlinear partial differential equation that must be solved numerically. In two-dimensions it takes the form

$$\left( \frac{\partial T}{\partial x} \right)^2 + \left( \frac{\partial T}{\partial y} \right)^2 = \frac{1}{c(x, y)^2}. \quad (4.1)$$

In order to state a scheme for solving (4.1), we need to define the first order forward and backward difference approximations for the first partial derivatives of $T$. Let $D_{i,j}^{+x}$ denote the forward difference approximation of the $x$ derivative at grid point $(x, y) = (ih, jk)$, where $h$ and $k$ denote the grid spacing in the $x$ and $y$ directions, respectively. Then, the forward difference approximation of $T_x$ can be written as

$$D_{i,j}^{+x}(T) = \frac{T_{i+1,j} - T_{i,j}}{h}. \quad (4.2)$$

Similarly, the backward difference approximation $D_{i,j}^{-x}$ is given by

$$D_{i,j}^{-x}(T) = \frac{T_{i,j} - T_{i-1,j}}{h}. \quad (4.3)$$

Analogous operators can be defined for the $y$ derivative as well. These can now be used to state the first-order upwind discretization of the eikonal equation that is described by Sethian [12]:

$$\max(D_{i,j}^{-x}(T), -D_{i,j}^{+x}(T), 0)^2 + \max(D_{i,j}^{-y}(T), -D_{i,j}^{+y}(T), 0)^2 = \frac{1}{c_{i,j}^2}, \quad (4.4)$$

$^4$Originating in wave optics, Huygen’s principle tells us that we can view an expanding front as a collection of point sources on a moving wavefront, each of which is the source of a new secondary wave. The front is then given by the envelope of these wavelets. This is how our front is able to “bend around” obstacles, as seen in Figure 2.
where \( c_{i,j} = c(ih,jk) \) is the velocity function at the grid point \((x,y) = (ih,jk)\).

There are several factors that lead to the upwinding condition that are beyond the scope of this paper—(see [1] and [13]). However, intuitively we can think of upwinding as a way to ensure that information (in our case, time) "flows" in the appropriate direction. That is, we want our solution to flow (i.e., propagate) away from points that it has already visited in a "downwind" fashion. For an exhaustive study of upwinding in the numerical solution of partial differential equations, in particular hyperbolic equations which are directly related to our equations, refer to [6]. In addition, although this first order scheme is sufficient for our needs\(^5\), it should be noted that information for higher order methods can be found in [13], but these methods are only applicable for structured grids.

Let us now consider how equation (4.4) can be solved. First, note that if all neighboring grid points have settled travel times, then we basically have a quadratic equation for \( T_{i,j} \). This observation leads to an iterative method for computing the solution given by Rouy and Tourin in [9]. Unfortunately, this procedure has a computational complexity on the order of \( N^4 \), where \( N \) is the number of grid points in each direction. Clearly, for large problems, this is not acceptable.

Fortunately, the Fast Marching Method provides us with a more efficient scheme. Due to the way in which (4.4) was constructed based on upwinding, we know that information is flowing one way. Furthermore, Epton and Grandine [1] have proven monotonicity properties that guarantee using (4.4) will produce the desired solution.

5 Configurations in \( \mathbb{R}^3 \)

The idea of a configuration space for the two-dimensional problem was introduced in Section 2.2. In that case, the configuration space \( \mathcal{C} \) was three-dimensional (two spatial coordinates and an angle of rotation). In other words, the problem had 3-degrees of freedom (3-DOF). Now, if we want to solve the three-dimensional path planning problem, then we will have 6-DOF. That is, \( \mathcal{C} \) will now be at least six-dimensional. This follows from the fact that a non-symmetric object moving in \( \mathbb{R}^3 \) contains three possible independent axes of rotation, so that it should require at least three parameters to describe the orientation.

5.1 Rotations in Space

In order to accurately define the general configuration space, we need to first consider the group of proper rotations in space. In general, this can be defined to be the space of rotation matrices \( \Theta \) in \( \mathbb{R}^{n \times n} \) given by

\[
SO(n) = \{ \Theta \in \mathbb{R}^{n \times n} : \Theta^T \Theta = I \text{ and } \det(\Theta) = +1 \}.
\]  

(5.1)

We will only concern ourselves with \( n = 3 \), where \( SO(3) \) is commonly referred to as the rotation group of \( \mathbb{R}^3 \), with the identity matrix \( I \) as the identity element and matrix multiplication as the group operation\(^6\). For a detailed discussion of \( SO(n) \), as well as other

\(^5\)First order schemes are also necessary when dealing with unstructured grids, which will be utilized when we investigate applying the algorithm to objects in \( \mathbb{R}^3 \) that rotate.

\(^6\)For \( n = 2 \), we were able to represent \( SO(2) \) with one parameter \( \theta \).
applications of group theory, see [2]. In addition, Murray et al. [7] discuss \( SO(3) \) in the context of robotic motion.

5.2 Formal Definition

We are now in a position to give a definition of the configuration space in terms of our problem. An object \( R \) moving in \( \mathbb{R}^3 \) that is allowed to rotate can be completely described by its configuration. Then, the set of all configurations of \( R \) is given by

\[
C = \mathbb{R}^3 \times SO(3),
\]

which is the tensor product of the Euclidian space \( \mathbb{R}^3 \) and the group of proper rotations \( SO(3) \).

As mentioned earlier, the real utility of the configuration space is that we can represent an object as a single point. For example, consider the piano mover's problem, and imagine trying to accurately represent the piano. Clearly, this is not an easy task, so what is typically done is to make certain simplifications so that the piano can be reduced to the union of convex polyhedra. Then, when formulating the problem in configuration space, modifications are made to the obstacles\(^7\) in order for the piano to be reduced to a single point \( x \in C \).

5.3 Representing the Configuration Space

We have seen that the configuration space \( C \) gives us a way to describe the location and orientation of an object \( R \). Thus, it seems reasonable that we should be able to represent a particular configuration of \( R \) by the point \( x = (v, \Theta) \), where \( v \in \mathbb{R}^3 \) is the position vector and \( \Theta \in SO(3) \) describes the rotation. However, \( \Theta \) contains nine elements, so that we would possibly need a total of twelve different parameters for every point in \( C \) (i.e., \( x \in \mathbb{R}^{12} \)). Fortunately, the properties of \( SO(3) \) give us conditions that make some of these parameters dependent on others, so that \( C \) is a subset of \( \mathbb{R}^{12} \). This leads one to wonder what the minimum number of parameters that are needed to represent points in \( C \). For example, when \( n = 2 \), we had that \( x = (x, y, \theta) \) with \( \theta \in [0, 2\pi) \) by employing modulo \( 2\pi \) arithmetic. That is, we were able to represent \( SO(2) \) with a single parameter by restricting the values of \( \theta \), so it seems reasonable that we should be able to obtain a similar reduced parameterization of \( SO(3) \).

5.3.1 Parameterizing \( SO(3) \)

It should be clear that the problem of determining the best possible representation of the configuration space is more a problem of finding a good parameterization of \( SO(3) \). Hence, let us consider some possibilities.

As noted, there are three axes of rotation for a 3-dimensional object, so it seems reasonable that we should be able to describe the orientation (i.e., represent \( SO(3) \)) with just three parameters. The most straightforward method for accomplishing this leads to the

\(^7\)The obstacles essentially "grow". That is, the shape of the obstacles will change when they are mapped into the configuration space—see [5].
classic Euler angles \((\alpha, \beta, \gamma)\), which describe the sequence of rotations of some coordinate frame relative to another. Unfortunately, the Euler angles suffer from singularities—that is, they are not injective. Furthermore, the problem can not be remedied by simply restricting the angle values like we did with \(\theta\) in the \(\mathbb{R}^2\) case.

It turns out that all possible 3-parameter representations of \(SO(3)\) are doomed to suffer from singularities. In fact, Murray et al. [7] point out that this is a topological fact. Thus, we are lead to look for the next best possibility: A 4-dimensional representation of \(SO(3)\) that does not contain singularities.

### 5.3.2 Exponential Coordinates

Consider again the rotation of an object in \(\mathbb{R}^3\), which can be described by some \(\Theta \in SO(3)\). If \(\phi \in \mathbb{R}\) measures the angle of rotation about a unit vector \(u \in \mathbb{R}^3\) which specifies the axis of rotation, then we should be able to relate \(\Theta\) with \(\phi\) and \(u\). That is, Euler’s Theorem (see [7], p. 30) tells us that every rotation of an object specified by an angle \(\phi\) and axis \(u\) corresponds to some \(\Theta \in SO(3)\), so it should be the case that we can write \(\Theta\) as a function of \(\phi\) and \(u\).

In order to obtain such a relationship, we need to first define the skew-symmetric matrix representation of a vector \(x \in \mathbb{R}^3\) to be

\[
\text{Skew}(x) = \hat{x} = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \tag{5.3}
\]

The set of all \(3 \times 3\) skew-symmetric matrices \(S\) is itself a vector space denoted by \(so(3)\). A proper definition of this space is

\[
so(3) = \{ S \in \mathbb{R}^{3 \times 3} : S^T = -S \}. \tag{5.4}
\]

Then, given \(\phi \hat{u} \in so(3)\) with \(||u|| = 1\), and \(\phi \in \mathbb{R}\), the exponential of \(\phi \hat{u}\) is given by

\[
\exp(\phi \hat{u}) = I + \phi \hat{u} + \frac{\phi^2}{2!} \hat{u}^2 + \ldots. \tag{5.5}
\]

Finally, it can be shown that \(\exp(\phi \hat{u}) \in SO(3)\) so that \(\Theta = \exp(\phi \hat{u})\) is in fact a proper rotation, giving us our desired relationship. Furthermore, if we are given \(\Theta \in SO(3)\), then there exists \(u\) and \(\phi\) with \(||u|| = 1\) such that \(\Theta = \exp(\phi \hat{u})\). This useful fact is proven by Murray et al. [7], where they refer to the vector \(\phi u\) as the the exponential coordinates of the rotation \(\Theta \in SO(3)\).

From a computational standpoint, it should be clear that the infinite series given by (5.5) is of little use. However, we have for \(\hat{x} \in so(3)\), the following recursive relations:

\[
\hat{x}^2 = xx^T - ||x||^2 I, \\
\hat{x}^3 = -||x||^2 \hat{x}, \tag{5.6}
\]

from which higher powers can be calculated. Then (5.6) can be used with \(x = \phi u\) and \(||u|| = 1\) to show that

\[
\exp(\phi \hat{u}) = I + \sin(\phi) \hat{u} + (1 - \cos(\phi)) \hat{u}^2, \tag{5.7}
\]

which is commonly referred to as Rodrigues’ formula.
5.3.3 Quaternions

A global parameterization of $SO(3)$ can be obtained using the set of unit quaternions. In general, an element $\tilde{q}$ of the set of quaternions $Q$ is given by $\tilde{q} = (q_0, q)$, where $q_0 \in \mathbb{R}$ and $q \in \mathbb{R}^3$. The unit quaternions $\tilde{q} \in Q$ are simply those that satisfy $\|\tilde{q}\| = 1$.

The real utility in quaternions is provided by the fact that $Q$ forms a group with quaternion multiplication as the group operation. Furthermore, it can be shown that the group structure of quaternions is directly related to the group $SO(3)$. That is, given $\Theta = \exp(\varphi \mathbf{u}) \in SO(3)$, define the quaternion $\tilde{q}$ by

$$\tilde{q} = (\cos(\varphi/2), \sin(\varphi/2) \mathbf{u}),$$

from which it is easy to verify that $\|\tilde{q}\| = 1$. Thus, $\tilde{q}$ lives on the unit sphere in $\mathbb{R}^4$.

If we are given a unit quaternion $\tilde{q} = (q_0, q)$, equation (5.8) can be used to determine the rotation via the relationships

$$\varphi = 2 \cos^{-1}(q_0),$$

and

$$\mathbf{u} = \begin{cases} \frac{q}{\sin(\varphi/2)} & \text{if } \varphi \neq 0, \\ 0 & \text{otherwise}. \end{cases}$$

5.4 Computational Grids

The way in which the computational grid is defined will directly translate into whether or not the Fast Marching Method can be effectively utilized. This amounts to determining the best way to lay a grid on $SO(3)$, so that the exploration of configuration space can be performed in an efficient manner. In particular, the way in which we have described how the FMM solves the discretized eikonal equation requires that the grid be rectangular. Furthermore, this facilitates the determination of neighboring grid points, which, in a rectangular grid, is performed twice for each degree of freedom at every grid point.

5.4.1 A Unit Disk Grid

Based on the discussion of a parameterization of $SO(3)$, it is apparent that the quaternions provide the best possible representation. However, this amounts to using points on the unit sphere (denoted $S^3$) in $\mathbb{R}^4$, which is clearly difficult to visualize. Instead, let us first consider what a grid may look like on the unit disk, which represents an “equitorial slice” of $SO(3)$. This can be thought of as three dimensional rotations with one component held fixed.

One possible coarse grid for the unit disk is given in Figure 4. Unfortunately a single grid is no longer possible, so five sub-grids are used. The central rectangular grid is denoted grid 0, with four surrounding non-rectangular grids 1 through 4 that give us our first hint at difficulties. This is due to the fact that our rectangular formulation of the Fast Marching Method will no longer work throughout the entire grid. Thus, we must turn to the considerably more complex theory of applying the FMM to unstructured grids, which is developed in Epton and Grandine [1].

---

8The “arrow” above a character will be used to denote quaternions.
In order to gain a better understanding of the difficulties associated with applying the Fast Marching Method to a patched, non-rectangular grid, consider the point \( A \) in Figure 4 which is in fact associated with 3 grids (grids 0, 3, and 4). Now, in order to formulate the discretized eikonal equation, we need to consider all possible ways in which information can arrive at \( A \) through neighboring grid points. However, two of the grids that contain \( A \) are non-rectangular so that we can no longer simply "look" in opposite directions to determine neighbors, since the grid is no longer isotropic. Moreover, since this is an interior point in two dimensions, it seems that there should be at least four neighbors, but it is not at all exactly clear what the neighbor set of \( A \) consists of.

If we denote neighbors of the grid point \( A \) as those being "adjacent" to \( A \), then we could have up to six neighbors for this example. Figure 5 shows a close-up view of our previous grid about the point \( A \), along with adjacent points labelled \( n_i, i = 1, \ldots, 6 \). First, we can reduce our neighbor set by one, since the diagonal point \( n_2 \) was not considered for the structured grid case, so there is no need to include it now. However, do the points \( n_4 \) and \( n_5 \) need to be included? The answer, initially given by Kimmel and Sethian [4] and developed fully in Epton and Grandine [1], is to require a grid that yields triangles with no obtuse angles. Justification for this acute triangulation of the mesh is a direct result of the upwind nature of solution procedure, but a rigorous proof is beyond the scope of this paper (see [1]).

One important concept that needs to be made clear is the acute triangulation is implemented at a "local level". Referring back to our example, we have that \( n_4 \) is a neighbor of \( A \) when determining the neighbor set of \( A \), but \( A \) is not included in the neighbor set of \( n_4 \). This is simply due to the fact that the angle between \( n_3 \) and \( n_5 \) with \( n_4 \) as a vertex is non-obtuse. Furthermore, Epton and Grandine [1] point out that this is a necessity when solving the eikonal equation on a curved surface, since a global acute triangulation is not

Figure 4: Patched grid of the unit disk. The point \( A \) is associated with grid regions 0, 3, and 4.
Let us now consider one possible approach for solving the eikonal equation on the unit disk. This can be accomplished by performing a linear mapping $L$ of the unit disk grid into a "rectangular version", where we can apply the FMM to solve the eikonal equation. In other words, we have $L : S \rightarrow [-1, 1] \times [-1, 1]$, where $S = \{(x, y) : x^2 + y^2 \leq 1\}$ denotes the unit disk. Afterwards, the inverse mapping $L^{-1}$ back onto the disk can be performed. This is possible only if we map the velocity function $c(x, y)$ as well, in order for the solution to be equivalent to what one would get without performing the mapping.

Figure 6 shows a rectangular version of our previous unit disk grid from Figure 4. As can be seen, this rectangular grid of the unit disk maintains the overall structure in that it contains 5 sub-grids with each sub-grid containing the same number of points, but now there is some redundancy in some of the grid points. That is, there are grid points along the edges of the outer sub-grids that are equivalent to points on the edge of different sub-grids, so special care needs to be taken at these points. For example, grid points along the left edge of grid 4 are equivalent to the grid points along the bottom edge of grid 3. To see this, consider the equivalent nodes labelled $e_3^4$ and $e_3^4$ in Figure 6, that are associated with grids 3 and 4, respectively. We say that these nodes are equivalent since the travel time will be the same at both points. Furthermore, it is not difficult to see that these grid points share neighbors.

Based on the rectangular structure of grid in Figure 6, it seems reasonable that we can implement the Fast Marching Method for this grid with slight modifications for points that are equivalent. For example, if the point $e_3^4$ from Figure 6 is the point with the most recently settled travel time (that is, $e_3^4$ was the point in $R_N$ with the minimum travel time so it is now in $A_{N+1}$), then $e_3^4$ needs to be added to the alive set $A_{N+1}$ as well. In addition, the neighbor set $R_{N+1}$ will now include the neighbors of both $e_3^4$ and $e_3^4$.

Figure 5: Closer view of the grid point $A$ along with six possible neighbors labelled $n_i$, $i = 1, \ldots, 6$. 

![Grid Point A and Neighbors](image-url)
In order to demonstrate what the contours would look like on the rectangular version of the unit disk, let us take uniform unit speed (i.e., \( c(x, y) = 1 \)) and start at an initial point in grid 4 (see Figure 7). This helps to demonstrate how the outer sub-grids are in fact “connected”, since the contours that reach the end of an outer sub-grid essentially pass directly into the adjacent sub-grid.

Figure 8 shows a density plot of \( T(x, y) \) on the unit disk. This is the result of computing the travel times on the rectangular version of the unit disk (Figure 7), and then mapping the results onto the unit disk.

### 5.4.2 The Fat Cube

The extension of the unit disk grid to three dimensions leads to what is commonly referred to as the fat cube grid of the unit ball. The fat cube is composed of an interior cube (corresponding to grid 0 of the unit disk grid) surrounded by six caps, which are 3-dimensional versions of regions 1-4 of the unit disk grid. Clearly this addition of one dimension will further complicate the issue of determining neighbor sets. However, the acute triangle condition in two dimensions can be extended, where now we require that the grid contains no simplices with obtuse angles.

As noted, the unit quaternion parameterization of \( SO(3) \) corresponds to points on the unit sphere \( S^3 \) in \( \mathbb{R}^4 \). A grid of \( S^3 \) can be obtained by extending the fat cube grid of \( S^2 \), which leads to the fat hypercube. Clearly, this fat hypercube is difficult to visualize, so that the details of applying the Fast Marching Method need to be further investigated. However, much of the necessary theory is developed in [1], where the ideas of solving the curved surface eikonal equation can be utilized for the unstructured patched grid on \( SO(3) \).
Figure 7: Contour plot of $T(x, y)$ on a rectangular version of the unit disk grid.

Figure 8: Density plot of $T(x, y)$ on the unit disk. Times were computed on the rectangular version and then mapped onto the unit disk. Lighter regions correspond to increasing time.
6 Conclusions and Future Work

The procedure for determining an optimal path has been outlined, where several different sub-problems have been addressed. The most significant of which is the gridding of $SO(3)$, which still requires further research.

In addition, the Fast Marching Method, an efficient numerical scheme for solving the eikonal equation, has been discussed. Solving the eikonal equation amounts to the determination of distances, so any method for performing this is quite useful. Readers interested in the eikonal equation, in particular a curved surface eikonal equation (which computes distances on surfaces), should refer to Epton and Grandine [1]. In addition, this reference also includes a solid theoretical framework in which to apply the FMM on both structured and unstructured grids.

The list of applications where Fast Marching Methods prove useful is impressive. For a discussion of these, along with references containing background information, see Sethian [13].

References


A Main FMM Code

function u = fmm(A, s, h, ngrid, skip)

% function to solve the eikonal equation ||grad(u)|| = s based
% on the Fast Marching Method algorithm.
% Input:
% A = alive set
% s = velocity function
% h = step size for each dimension (ndim-vector)
% ngrid = number of grid points for each dimension (ndim-vector)
% skip = grid skip factors for each dimension ((ndim+1)-vector)
% Output:
% u = travel times for all grid points (vector)
%
% JJ Clark, 12.01

its = 1;
ndim = length(ngrid); % # of dimensions
N = skip(ndim+1); % total number of grid points
itmax = N*N;
u = zeros(N,1); % initialize travel times
C = A; % initialize close set equal to alive set
% initialize all travel times at far points (not in A or C) to -infinity
u(find(C == 0)) = realmax;

% iterate until all grid points have settled travel times
while max(u) == realmax & its < itmax
% find close node with smallest travel time
Cinds = find(C);
minval = min(u(Cinds));
inds = find(u(Cinds) == minval);
index = Cinds(inds(1));

% add to alive set and remove from close
A(index) = 1;
C(index) = 0;

% determine neighbors and update travel times
xind = index2ij(index,grid);
for i=1:ndim
  il = nextlo(index,xind(i),ngrid(i),skip(i));
  if il > 0 & A(il) ~= 1
    u(il) = tcand(il,u,A,s,h,ngrid,skip);
    C(il) = 1;
  end
end
A.1 Function to compute travel times

function tc = tcand(index,u,A,s,h,ngrid,skip)

% function to compute candidate travel time based on fast
% marching methods. Called from fmm.m
% Input:
% index = current grid index
% u = travel times
% A = alive set
% s = velocity function
% h = grid spacing vector
% ngrid = # of grid points in each dimension
% skip = grid skip factor
% Output:
% tc = candidate travel time
%
% JJ Clark, 12.01

ndim = length(ngrid); % number of dimensions
a = 0; b = 0; c = 0;

if A(index) == 1
    tc = u(index);
    return;
end
xind = index2ij(index,ngrid);
for i=1:ndim
    nbr = realmax;
    % determine neighbor with smallest travel time
    il = nextlo(index,xind(i),ngrid(i),skip(i));
    ih = nexthi(index,xind(i),ngrid(i),skip(i));
    if il > 0 & A(il) == 1
        nbr = min(nbr,u(il));
    end
    if ih > 0 & A(ih) == 1
        u(ih) = tcand(ih,u,A,s,h,ngrid,skip);
        C(ih) = 1;
    end
end
its = its+1;
end
if ih > 0 & A(ih) == 1
    nbr = min(nbr, u(ih));
end
if nbr < realmax
    hh = h(i)*h(i);
    a = a + 1/hh;
    b = b + 2*nbr/hh;
    c = c + nbr^2/hh;
end
c = c - s(index);
if a ~= 0
    tc = (b + sqrt(b*b - 4.0*a*c)) / (2.0*a);
else
    tc = realmax;
end
tc = min(tc, u(index));

A.2 Subroutines

function index = ij2index(ij, skip)

% function to convert coordinate indices (i,j) to global index
% Input:
%   ij = coord. indices
%   skip = grid skip factor
% Output:
%   index = global index

ndim = length(ij);
ind = ij(1);
for k=2:ndim
    index = index + (ij(k)-1)*skip(k);
end

function ij = index2ij(index, ngrid)

% function to convert global index to coordinate indices (i,j)
% Input:
%   index = global index
%   ngrid = # of grid points in each direction
% Output:

index = index - 1;
ndim = length(ngrid);
ij = zeros(ndim,1);
for k=1:ndim
    ij(k) = mod(index, abs(ngrid(k)));  
    index = floor(index/abs(ngrid(k))); 
end
ij = ij + 1;

function ans = nexthi(index, icrd, ngrid, skip)

    if icrd ~= abs(ngrid)
        ans = index + skip;
        return;
    end
    if ngrid > 0
        ans = -1;
        return;
    end
    ans = index + (ngrid + 1) * skip;

function ans = nextlo(index, icrd, ngrid, skip)

    if icrd ~= abs(ngrid)
        ans = index + skip;
        return;
    end
    if ngrid > 0
        ans = -1;
        return;
    end
    ans = index + (ngrid + 1) * skip;
if icrd = 1
    ans = index - skip;
    return;
end
if ngrid > 0
    ans = -1;
    return;
end
ans = index - (ngrid + 1) * skip;

B  Code To Compute Optimal Path

function xpath = path2d(start, dest, xh, u, h, ngrid, skip)

    % function to determine the optimal path based on travel times by
    % solving the ode
    %
    %   dx   -grad(u)
    %   dt   |grad(u)|
    %
    % The travel time function u is computed with the fmm.m function.
    % Input:
    %   start = x0 (given by index)
    %   dest = destination point (index)
    %   xh = path step size
    %   u = travel times (stored as a vector)
    %   h = grid step sizes (ndim-vector)
    %   ngrid = # of grid points in each direction (ndim-vector)
    %   skip = grid skip factor ((ndim+1)-vector)
    % Output:
    %   xpath = matrix containing path coordinates
    %
    % JJ Clark, 12.01

    ndim = length(ngrid)  % # of dimensions
    hpath = xh * ones(ndim, 1);  % path step size (vector)
    itmax = length(u)^2;
    eta = 1e-4;  % error control parameter

    % determine x coord's of starting position
    xind = index2ij(start, ngrid);
    x = (xind - 1) .* h;
% determine indices and x coord's of destination
xfind = index2ij(dest, ndim, ngrid);
xf = (xfind - 1) .* h;

% record starting coord's in path matrix and initialize index
xpath(:, 1) = x;
index = start;
its = 1;

% integrate until we've reached the destination
while index ~= dest & its < itmax
    % check to see if we are in the "start cell"
    if norm(xf - x) < norm(h)
        xpath(:, its) = xf;
        return
    end
    xtild = x;
    grad = grad2d(u, x, h, ngrid, skip);
    k1 = - hpath .* grad;
    xtemp = x + k1;
    g(:, its) = grad;
    grad = grad2d(u, xtemp, h, ngrid, skip);
    k2 = - hpath .* grad;
    x = x + k1/2 + k2/2;
    end

    diff = norm(xtild - x);
    if diff ~= 0
        hpath = min([3/2, (3 * eta / (4 * diff))^(1/3)]) * hpath;
    end

% determine indices and index of x coord's
xind = floor(x ./ h) + 1;
index = ij2index(xind, skip);
its = its + 1;
xpath(:, its) = x;
28
B.1 Compute gradient

function ans = grad2d(u, x, h, ngrid, skip)

% function to compute grad(u)/|grad(u)| at x using bilinear
% interpolation. Function is called by path2d.m that solves the
% optimal path problem in 2-dimensions.
% Input:
%  u = travel time function
%  x = point to evaluate at
%  h = step sizes for each direction (vector)
%  ngrid = # of grid points in each direction (vector)
%  skip = grid skip factor (vector)
% Output:
%  ans = grad(u)/|grad(u)|

ndim = length(x); % # of dimensions
xind = floor(x./h) + 1; % indices of x
ind = ij2index(xind,skip); % index of x
orig = (xind - 1).*h; % the local 'origin' of x (indices)
newx = (x - orig)./h; % transformed coord's of x based on orig
grad = zeros(ndim,1); % grad(u)
gt = grad; % interp coefficient

% compute gradient at corners of cell
for j=0:1
    for i=0:1
        cindex = ind + i*skip(1) + j*skip(2); % corner index
        vals = abs(1-i-newx(1)) * abs(1-j-newx(2)); % interp values
        for k=1:ndim
            back = 0;
            forw = 0;
            cind = index2ij(cindex, ngrid); % corner indices
            iy = nextlo(cindex, cind(k), ngrid(k), skip(k));
            if iy == -1
                back = (u(cindex) - u(iy)) / h(k); % backward diff approx.
            end
            iy = nexthi(cindex, cind(k), ngrid(k), skip(k));
            if iy == -1
                forw = (u(iy) - u(cindex)) / h(k); % forward diff approx
            end
            gt(k) = max([back, -forw, 0]); % interp coefficient
        end
    end
end
if gt(k) == -forw
    gt(k) = forw;
end
end
grad = grad + vals*gt;
end
ans = grad/norm(grad);

C  FMM Code on Unit Disk

function u = fmmdisk(A, s, h, ngrid, skip, kk, n)
% function to solve the eikonal equation ||grad(u)|| = f on a
% rectangular version of the unit disk grid via the Fast Marching
% Method. Calls equivnfz.m to compute the equivalent nodes and
% no-fly-zone nodes (see this m-file for the grid representation).
% Input:
%  A  alive set
%  s  velocity function
%  h  step size for each dimension (vector)
%  ngrid = number of grid points for each dimension (vector)
%  skip = grid skip factor
%  kk  # of grid points along 'short' side of outer grids
%  N  # of grid points along 'long' side and size of interior grid
% Output:
%  u  travel times for all grid points (vector)
%
% JJ Clark, 1.29.02

its = 1;
ndim = length(ngrid);  % # of dimensions
N = skip(ndim+1);  % total number of grid points
itmax = N*N;  % maximum # of iterations
u = zeros(N,1);  % initialize travel times
C = A;  % initialize close set equal to alive set

% initialize all travel times at far points (not in A or C) to -infinity
u(find(C == 0)) = realmax;

% determine equivalence relations and no-fly-zones
[eq,nfz] = equivnfz(kk,n);

% determine neighbor index(es)
Cinds = find(C);

% iterate until all grid points have settled travel times
% (i.e., no more neighbors)
while ~isempty(Cinds) & its < itmax
    % find close node with smallest travel time
    minval = min(u(Cinds));
    inds = find(u(Cinds) == minval);
    index = Cinds(inds(1));

    % add to alive set and remove from close
    A(index) = 1;
    C(index) = 0;

    % determine if there is an equivalent node
    eqindex = findeq(index,eq);
    if eqindex ~= 0
        A(eqindex) = 1;
        C(eqindex) = 0;
    end

% determine neighbors and update travel times
xind = index2ij(index,ngrid);
for i=1:ndim
    % determine next lower index
    il = nextlo(index,xind(i),ngrid(i),skip(i));
    % check for end of grid, not already alive, and a no-fly-zone
    if il > 0 & A(il) ~= 1 & isempty(find(nfz == il))
        u(il) = tcanndisk(il,u,A,s,h,ngrid,skip,eq,nfz);
        C(il) = 1;
    end
    % check for equivalent node
    eqindex = findeq(il,eq);
    if eqindex ~= 0
        u(eqindex) = u(il);
        C(eqindex) = 1;
    end
end

% determine next higher index
ih = nexthi(index,xind(i),ngrid(i),skip(i));
% check for possible path
if ih > 0 & A(ih) ~= 1 & isempty(find(nfz == ih))
    u(ih) = tcanndisk(ih,u,A,s,h,ngrid,skip,eq,nfz);
    C(ih) = 1;
    % check for equivalent node
    eqindex = findeq(ih,eq);
if eqindex ̸= 0
    u(eqindex) = u(ih);
    C(eqindex) = 1;
end
end
end
Cinds = find(C);      % indices of close points
its = its+1;
end

C.1 Function to compute travel times

function tc = tcanddisk(index,u,A,s,h,ngrid,skip,eq,nfz)

% function to compute candidate travel time based on fast
% marching methods for the eikonal equation on the unit disk.
% Called from fmmdisk.m
% Input:
% index= current grid index
% u = travel times
% A = alive set
% s = velocity function
% h = grid spacing vector
% ngrid = # of grid points in each dimension
% skip = grid skip factor
% eq = equivalent node storage
% nfz = no-fly-zone index #'s
% Output:
% tc = candidate travel time
%
% JJ Clark, 1.29.02

ndim = length(ngrid);  % number of dimensions
a = 0;
b = 0;
c = 0;

if A(index) == 1  % return if already alive and settled
    tc = u(index);
    return;
end
if ~isempty(find(nfz == index))  % illegal no-fly-zone index
    tc = realmax;
    return;

end
end
xind = index2ij(index,ngrid);

for i=1:ndim
    nbr = realmax;
    % determine neighbor with smallest travel time
    il = nextlo(index,xind(i),ngrid(i),skip(i));
    ih = nextthi(index,xind(i),ngrid(i),skip(i));
    if il > 0 & A(il) == 1
        nbr = min(nbr,u(il));
    end
    if ih > 0 & A(ih) == 1
        nbr = min(nbr,u(ih));
    end
    if nbr < realmax  % solve quadratic
        hh = h(i)*h(i);
        a = a + 1/hh;
        b = b + 2*nbr/hh;
        c = c + nbr^2/hh;
    end
end
C = C - s(index);
if a == 0
    tc = (b + sqrt(b*b - 4.0*a*c)) / (2.0*a);
else
    tc = realmax;
end
tc = min(tc, u(index));

% compute travel time for equivalent index
eqindex = findeq(index,eq);
if eqindex ~= 0
    xind = index2ij(eqindex,ngrid);
    a = 0;
    b = 0;
    c = 0;
    for i=1:ndim
        nbr = realmax;
        % determine neighbor with smallest travel time
        il = nextlo(eqindex,xind(i),ngrid(i),skip(i));
        ih = nextthi(eqindex,xind(i),ngrid(i),skip(i));
        if il > 0 & A(il) == 1
            nbr = min(nbr,u(il));
        end
        if ih > 0 & A(ih) == 1
nbr = min(nbr,u(ih));
end
if nbr < realmax % solve quadratic
    hh = h(i)*h(i);
    a = a + 1/hh;
    b = b + 2*nbr/hh;
    c = c + nbr^-2/hh;
end
end
c = c - s(eqindex);
if a ~= 0
    tceq = (b + sqrt(b*b - 4.0*a*c)) / (2.0*a);
else
    tceq = realmax;
end
tc = min(tc, u(index));
end
% take minimum of possible times
tc = min(tc, u(index));

C.2 Subroutines for disk function

function [equiv,nfz] = equivnfz(kk,N)

% function to determine equivalent grid points and no-fly-zone
% grid points for applying the FMM on the approximate unit
% disk grid. Equivalent nodes occur along the short edges
% of the outer grids, while no-fly-zone grid points are found
% directly adjacent to equivalent grid points that are not
% contained within the grid itself. A global grid is formed
% over the five kk-by-N subgrids which have the form:
%
%       *******
%       *    *
%       *** *** Each outer subgrid is kk-by-N,
%       *    *
%       *** *** so that the total grid size is
%       *    *
%       (N+2kk-2)-by-(N+2kk-2).
%       *    *
%       *******
%
% See the m-file fmmso2.m for further details.
% Input

%
% kk = # of grid points along the 'short' side
% N = # of grid points along the 'long' side
% Output
% equiv = 2-by-? vector containing equiv node indexes
% nfz = 2-by-? vector containing no-fly-zone indexes
%
% JJ Clark, 2.22.02

k = kk-1;
% lower left
for i=1:k
    diff = (kk-i)*(N-1); % difference in equivalent nodes
    ind = (kk-1)*N+i; % horizontal node
    equiv(1,i) = ind; % horizontal
    nfz(1,i) = ind-N; % determines no-fly-zone
    equiv(2,i) = ind-diff; % vertical
    nfz(2,i) = ind-diff-1;
end

% lower right
for i=1:k
    diff = i*N+i; % horizontal
    ind = kk*N-(k-i);
    equiv(1,i+k) = ind; % horizontal
    nfz(1,i+k) = ind-N;
    equiv(2,i+k) = ind-diff; % vertical
    nfz(2,i+k) = ind-diff+1;
end

% upper left
for i=1:k
    diff = (kk-i)*N+kk-i; % horizontal
    ind = N*(N-kk)+i;
    equiv(1,i+2*k) = ind; % horizontal
    nfz(1,i+2*k) = ind+N;
    equiv(2,i+2*k) = ind+diff; % vertical
    nfz(2,i+2*k) = ind+diff-1;
end

% upper right
for i=1:k
    diff = i*(N-1); % horizontal
    ind = N*(N-kk+1)-kk+1+i;
    equiv(1,i+3*k) = ind; % horizontal
    nfz(1,i+3*k) = ind+N;

equiv(2,i+3*k) = ind+diff; % vertical
nfz(2,i+3*k) = ind+diff+1;
end

function eqindex = findeq(index,equiv)

% function to determine if index is equivalent to another grid point.
% Input:
% index
% equiv
% Output:
% eqindex = 0 if no equivalent nodes, or equivalent index o.w.
% JJ
% JJ Clark, 2.22.02

% determine indices in the equiv array
[ieq,jeq] = find(equiv == index);
if isempty(ieq)
    eqindex = 0;
    return;
else
    eqindex = equiv(ceil(2/ieq),jeq);
end

D Code to Perform Map to Unit Disk

% Code to map grid points and travel times from rectangular grid to
% the unit disk grid. Travel times are computed using fmmdisk.m
% Necessary input:
% kk = # of grid points along outer regions (see equivnfz.m)
% h = grid spacing vector
% u = travel time vector
% ngrid = # of grid points in each direction
% Computes:
% ud = array containing coord's on unit disk with associated times
% JJ
% JJ Clark, 4.01.02

a = .7/sqrt(2); % define size of interior square on disk
c = abs((kk-1)*h(1)-1); % define size of interior square on rectangle
ud = [];
% contains coord's with travel times
j = 1;
% loop thru all travel times
for i=1:length(u)
    if u(i) < realmax
        xind = index2ij(i,ngrid);
        xcrd = (xind - 1).*h - 1;
        % if not in no-fly-zone
        % compute indices
        % compute coord's
        % determine which region we are in
        if abs(xcrd(1)) < c+h(1)/2
            if abs(xcrd(2)) < c+h(2)/2 % region 0 - interior square
                igrd = 0;
            elseif xcrd(2) < c % region 1 - top cap
                igrd = 1;
            else % region 4 - bottom cap
                igrd = 4;
            end
        elseif xcrd(1) > c % region 2 - right cap
            igrd = 2;
        else % region 3 - left cap
            igrd = 3;
        end
        xd = rec2disk(xcrd,a,c,igrd);
        ud(:,j) = [xd; u(i)];
        j = j+1;
    end
end

function x = rec2disk(w,a,c,igrd)

% function to map the rectangular version of the unit disk grid into
% the 'proper' unit disk grid.
% The rectangular version lives on [-1,1] x [-1,1],
% with the interior square given by [-c,c] x [-c,c].
% Input:
%   w = coord's on rectangular grid
%   a = interior square of disk grid [-a,a] x [-a,a]
%   c = interior square of rect grid [-c,c] x [-c,c]
%   igrd = grid region id
% Output:
%   x = coord's on unit disk
% JJ Clark, 4.01.02

rt2i = 1/sqrt(2);
xi = w(1);
eta = w(2);
% determine which region and rotate into region 1.
switch igrd
  case 0
    x = (w+c)/(2*c);
    x = 2*a*x-a;
    return;
  case 1
    A = eye(2);
  case 2
    A = [0 -1; 1 0];
  case 3
    A = [0 1; -1 0];
  case 4
    A = [-1 0; 0 -1];
end

w = A*[xi; eta];
% perform rotation
xi = w(1);
eta = w(2);

xi = (xi+c)/(2*c);
eta = (eta-c)/(1-c);
% map [-c,c] --> [0,1]
% map [c,1] --> [0,1]

% define adjustment value for bilinear interp map
c = xi*eta*xtop(1,a) + xi*(1-eta)*xbtm(1,a) ...
  + eta*(1-xi)*xtop(0,a) + (1-xi)*(1-eta)*xbtm(0,a);
% perform bilinear interpolation
x = (1-eta)*xbtm(xi,a) + eta*xtop(xi,a) ...
  + (1-xi)*xlft(eta,a) + xi*xrt(eta,a) - c;
% rotate back to proper place
switch igrd
  case 1
    return;
  case 2
    A = [0 1; -1 0];
    x = A*x;
  case 3
    A = [0 -1; 1 0];
    x = A*x;
  case 4
    A = [-1 0; 0 -1];
    x = A*x;
end

38