Solving Reachable Sets on a Manifold

by

Elizabeth Ann Cross

B.Math., Carleton University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

August 30, 2007

Abstract

First, this thesis explores the implementation of the fast marching method as part of the toolbox of level set methods. This method uses Dijkstra's algorithm to approximate the solution to the non-linear Eikonal equation. Functions for calculating signed distances and extension velocities are also implemented. These functions use the fast marching method in their implementation.

Second, it explores a method for computing reachable sets on a manifold; in other words, the dynamics governing these reachable sets can be described by a Differential Algebraic Equation. It uses level set methods to solve the underlying Hamilton Jacobi equation of the reachable set and it ensures an accurate solution on the manifold by using the closest point method. The closest point method guarantees that the reachable set is perpendicular to the manifold at the points of intersection. Several two and three dimensional toy problems and a real-life power generator problem are explored in order to test the method.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost I would like to thank Ian Mitchell for providing me with excellent guidance and support throughout my thesis. He also kept me organized and on track during the entire thesis process. I would also like to thank Robert Bridson for providing insightful comments about my thesis.

Next I would like to thank my SCL labmates for scientific and not so scientific discussions. These discussions made everyday in the lab an enjoyable day. I would also like to thank my other friends in the department and my friends at UBC.

The courses I took in my first year provided me with a wide variety of scientific computing knowledge. I would like to thank the professors who taught these interesting courses.

Finally I would like to thank my moral support system: Heather, Doug, Angie and Clint. They have always supported me through whatever new adventures I chose to pursue.

Chapter 1

Introduction

The first purpose of this research is to add some functionality to Ian Mitchell's Level Set Toolbox [6], and this functionality is an implementation of the fast marching method. This method uses Dijkstra's algorithm to approximate the solution to the non-linear Eikonal equation. In addition, functions for calculating signed distances and extension velocities are also implemented. The fast marching method is used in the calculation of both signed distances and extension velocities.

The second purpose of this research is to develop a method for solving reachable sets on a manifold. The dynamics governing the reachable set evolution can be described by a Differential Algebraic Equation (DAE), where the manifold is an algebraic constraint and the movement of the reachable set is described by a differential constraint. A common method used to solve reachable sets in a full dimensional space is the level set method. In this case the reachable set is represented by an implicit surface function. This research adapts these level set techniques in order to solve reachable sets on a manifold. This adaptation uses the closest point method to ensure that the implicit surface function representing the reachable set is perpendicular to the manifold at all points of intersection. The function for calculating signed distances is used in the closest point method implementation. Thus, the work in the first part of the thesis is incorporated into second part.

This thesis will describe fast marching methods, signed distance functions and extension velocities. It will then outline the implementation of these methods and show some example problems to demonstrate that the implementation is correct. Next, this thesis will outline the method developed to solve reachable sets on a manifold. It will also demonstrate the method by solving two toy problems. Finally, the thesis will describe a power generator example and demonstrate how this real life DAE can be solved with the previously described methods.

1.1 Implicit Surface Functions

In this thesis, closed surfaces are represented by implicit surface functions, ϕ [7]. An interface with a dimension of $n-1$ is embedded as an isocontour into a function ϕ with an input dimension of n and an output dimension of 1. The zero isocontour, $\phi(x) = 0$, is usually used to represent the interface but any isocontour, $\hat{\phi}(x) = a$, can be used for this purpose. These isocontours are equal

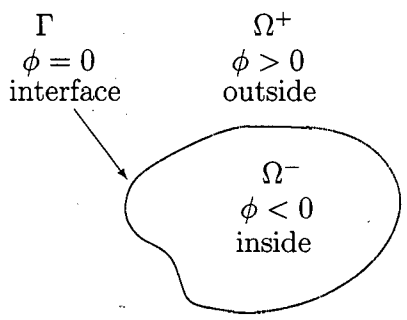

Figure 1.1: Implicit surface representation of an arbitrary curve.

up to a scalar translation, $\phi(x) = \hat{\phi}(x) - a$. This thesis uses the zero isocontour to represent interfaces, so the explicit representation of the interface is defined as $\Gamma = \{x \mid \phi(x) = 0\}$. The zero isocontour divides the domain into two sets. The set inside the interface is defined by $\Omega^- = \{x \mid \phi(x) < 0\}$ and the set outside the interface is defined by $\Omega^+ = \{x \mid \phi(x) > 0\}$.

The majority of the typical geometric tasks for sets and surfaces are easily done with the implicit surface representation. This representation makes it simple to identify which side of the interface a point is on simply by looking at the sign of $\phi(x)$. It is also easy to construct representations by combining several implicit surface representations together. For example, the intersection of the interiors of two interfaces can be represented by taking the maximum value between the two implicit surface representations at each point in the domain, $\phi_3(x) = \max(\phi_1(x), \phi_2(x))$. The outward normal of an implicit surface function is defined as

$$\vec{N} = \frac{\nabla\phi}{|\nabla\phi|}$$

and it is defined over the entire domain except where $|\nabla\phi| = 0$, or where ϕ is not differentiable. This thesis uses finite difference techniques on a cartesian grid to calculate derivatives, such as those in $\nabla\phi$.

Chapter 2

Fast Marching Methods

2.1 Background

2.1.1 Fast Marching Method

The Fast Marching Method (FMM) was originally described in [16] and independently described in [11]. It is a numerical scheme which approximates the solution to the non-linear Eikonal equation [12]. This equation is of the form

$$\begin{aligned} \|\nabla u(x)\| &= c(x) \text{ in } \Omega, \quad c(x) > 0 \\ u &= b(x) \text{ on } \partial\Omega \end{aligned} \tag{2.1}$$

where Ω represents the domain. The boundary is represented by $\partial\Omega$. In the case of the FMM the boundary is the interface and it is fixed throughout the entire calculation. The cost function is $c(x)$ and $b(x)$ is the boundary value function and they are both the given data. The cost function is defined over the whole domain, while the boundary function is only defined on $\partial\Omega$ [12].

Equation (2.1) is defined in a continuous domain. The fast marching method uses a discrete method to solve a continuous problem. This discrete method is outlined in Algorithm 1 and it is called Dijkstra's algorithm [3]. In the algorithm the continuous domain Ω is discretized into a graph, $\mathcal{G}$, consisting of nodes, $\mathcal{G}_n = \{x_i\}$, and edges, $\mathcal{G}_e$. The discretized version of the boundary function, $\partial\Omega$, is the set of nodes $\mathcal{T} \subset \mathcal{G}_n$. Each node in $\mathcal{G}_n$ has a set of neighbors, $\mathcal{N}(x_i)$ associated with it. Each non-boundary node, $x_i \in \mathcal{G}_n \setminus \mathcal{T}$, has a cost, $c(x_i)$, and each boundary node, $x_i \in \mathcal{T}$, has a boundary value, $b(x_i)$. The algorithm first initializes the value, $u(x_i)$, of each node to $+\infty$. It then sets the value of

foreach $x_i \in \mathcal{G}_n \setminus \mathcal{T}$ **do** $u(x_i) = +\infty$
foreach $x_i \in \mathcal{T}$ **do** $u(x_i) = b(x_i)$
$\mathcal{Q} \leftarrow \mathcal{G}_n$
while $\mathcal{Q} \neq \emptyset$ **do**
  $x_i \leftarrow$ `ExtractMin(`$\mathcal{Q}$`)`
  foreach $x_j \in \mathcal{N}(x_i)$ **do**
    $u(x_j) \leftarrow$ `Update(`x_j, $\mathcal{N}(x_j)$, u, c`)`

Algorithm 1: Generic shortest path dynamic programming algorithm. Depending on the choice of update function, this algorithm will produce either Dijkstra's algorithm or the fast marching method.

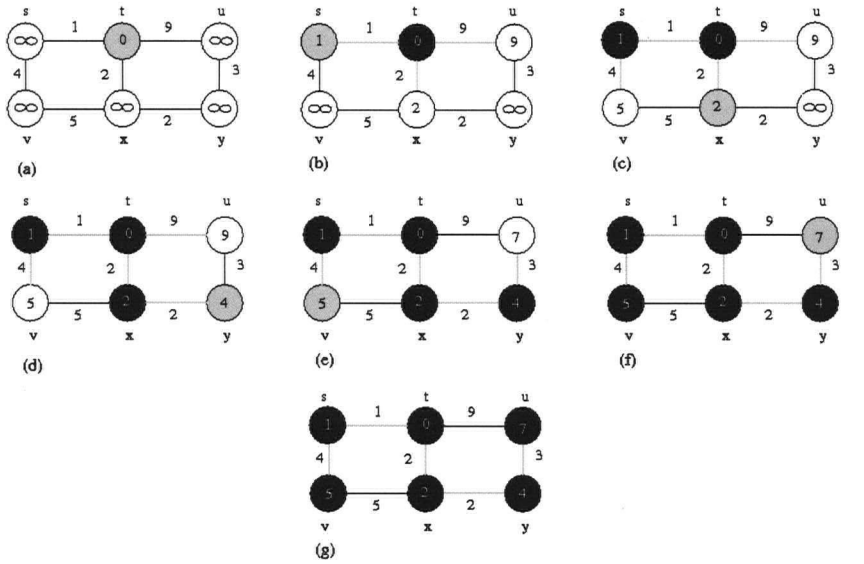

Figure 2.1: Dijkstra's method on a undirected discrete graph.

each boundary node with the data. A heap more generally, a priority queue, is constructed, $\mathcal{Q}$, which contains all the nodes in the graph. Dijkstra's algorithm removes the node with the minimum value and updates all of its neighbors by using the specified update procedure. The algorithm continues to do this until $\mathcal{Q}$ is empty. Thus, a value function is constructed by dynamic programming. It contains the minimum value, $u(x_i)$, at each node, x_i. This minimum value represents the cheapest path from a boundary node to x_i.

Two key factors contribute to the effectiveness of the FMM. First, an upwinding scheme is used to provide an approximation of the viscosity solution for the problem. This type of solution is a particular weak solution which is desirable in this case. Second, the scheme can be computed in a very fast manner by marching the solution outwards from the boundary. Since an upwinding method is used, the value at each node increases as the solution is propagated outwards from boundary nodes.

Dijkstra's algorithm is usually used to solve a single-source shortest-path problem on a undirected graph. In this case, the cost function is defined on the edges, $\mathcal{G}_e$, but in the case of the fast marching method the cost function is defined on the nodes, $\mathcal{G}_n$. Figure 2.1 demonstrates how Dijkstra's algorithm, Algorithm 1, works using its own update procedure as defined in Algorithm 2. In Algorithm 2, $e(x_i, x_j)$ represents the edge between nodes x_i and x_j. If the cost function is defined on the nodes then Dijkstra's method on a discrete graph will never update the value of a node. Thus this example defines the cost function on the edges, $c(e(x_i, x_j))$, to demonstrate how the update function can change a value for a node. In Figure 2.1, **(a)** demonstrates the graph after initialization.

Input: x_j, $\mathcal{N}(x_j)$, u, c
Output: $\min_{x_k \in \mathcal{N}(x_j)}(c(e(x_j, x_k)) + u(x_k))$
Algorithm 2: Dijkstra's update method with costs defined on the edges

Input: x_j, $\mathcal{N}(x_j)$, u, c
Output: $c(x_j) + \min_{x_k \in \mathcal{N}(x_j)} u(x_k)$
Algorithm 3: Dijkstra's update method with costs defined on the nodes

The boundary node is $\mathtt{t} \in \mathcal{T}$ and it has a boundary value, $b(\mathtt{t})$, of 0, and the value, $u(x_i)$, of all other nodes is initialized to $+\infty$. The costs to travel between nodes is defined on the edges. **(b)-(g)** demonstrate the situation after each successive iteration of the while loop. The value of each vertex is displayed inside the node. The black vertices have had their final values assigned and are no longer in the heap $\mathcal{Q}$. The grey vertex in each image currently has the smallest value in the heap and is next to be removed. The grey edges are the current minimum path from the boundary node **t** to all other nodes. The update of a node is demonstrated from **(d)** to **(e)**. Node **u** originally has a value of 9 and its path is **t-u**. Then node **y** is popped from the heap and updates its neighbours, so **u** has a smaller value of 7 and its new path is **t-x-y-u**.

Dijkstra's algorithm is most efficiently implemented by using a min-heap data structure for $\mathcal{Q}$. Each node in the domain is extracted only once from $\mathcal{Q}$, and this means that it takes $O(N)$ operations to traverse all of the N domain nodes. It takes $O(\log N)$ to maintain the heap property because a node is added once, removed once and updated once for each neighbour and each of these operations is $O(\log N)$ complexity in a min heap. Thus, by using a min-heap data structure Dijkstra's algorithm has a complexity of $O(N \log N)$. A backpointer array is maintained in order to determine a node's position in the heap in $O(1)$ time, and this information is needed in order to update the value of nodes in the heap [12].

The chosen update method determines how the lowest value for each node is defined. A one-norm approximation of this value is calculated by using Dijkstra's update method on a Cartesian grid with four neighbours. The fast marching method is a two-norm approximation of this value. Dijkstra's update method with the costs defined on the nodes is described in Algorithm 3, and only this version of Dijkstra's update method in implemented in the Toolbox. The fast marching update method is described in Algorithm 4. This implementation of the fast marching method update procedure was originally described in [4], but Algorithm 4 is updated so that all dimensions have the same grid size, h, instead of a grid size of 1 which was assumed in [4].

2.1.2 Signed Distance Function

A signed distance function is an implicit surface function with the additional property that $\|\nabla u\| = 1$. An implicit surface function, ϕ, can be turned into a signed distance function, u. The signed distance function will have the same zero

Input: x_j, $\mathcal{N}(x_j)$, u, c
for $k \in 1, .., n$ **do**
  $a_k = \min(u(\mathcal{N}_k(x_j)))$, where $\mathcal{N}_k(x_j)$ is the set of x_j's two neighbours along the k coordinate
$a =$ `SortAscending`(a)
$m = n$
while $c(x_j)^2 < \Sigma_{k\in\{1,..,m\}}(h(a_m - a_k))^2$ **do**
  $m = m - 1$
$t = \frac{1}{m}\left(\Sigma_k a_k + \sqrt{m(hc(x_j))^2 + \Sigma_k\Sigma_l a_k a_l - m\Sigma_k a_k^2}\right)$, where Σ indexes are $\{1, .., m\}$
Output: t

Algorithm 4: Fast marching update method

level set as the implicit surface function but it will have a gradient magnitude of one.

The fast marching method can be used to solve for a signed distance function by choosing $c(x) = 1$ and $\Gamma = \partial\Omega$ [1]. The set of boundary nodes, $\mathcal{T}$, and their values, $b(x_i)$, must also be defined in order to run the fast marching method. In the case of the signed distance function, the boundary function is an interface which consists of the zero level set of the implicit surface function. Thus, the boundary nodes are the nodes which are closest to this interface. These nodes can be identified by checking to see if the node has at least one neighbour with an opposite sign. If the signs are opposite then the nodes are on different sides of the interface and thus the node is a boundary node. Once a boundary node is identified its distance, d, from the interface is calculated to be used as its initial value. An example of a boundary node with two neighbours on the other side of the interface can be seen in Figure 2.2. There are several other cases where the neighbours are in other positions; to see these examples consult [1].

In order to calculate the distance from a boundary node, x_0, to the interface, first a distance, s_i, must be calculated for each neighbour x_i of x_0 which is on the other side of the interface. The node x_0 has up to $2n$ neighbours, $x_1, \cdots, x_{2n}$, where n is the number of dimensions of the domain. The distance, s_i, is defined as the distance from x_0 to the intersection point of the interface, $\tilde{x}_i$, on the line connecting the two grid points. The value for s_i is approximated by comparing slopes of ϕ along the line connecting the two grid points. The slope of ϕ for the line segment from x_0 to its neighbour x_i is compared to the slope of ϕ for the line segment from x_0 to the intersection point, $\tilde{x}_i$ in order to solve for s_i. Here is the equation comparing the two slopes

$$\frac{\phi(x_i) - \phi(x_0)}{x_i - x_0} = \frac{0 - \phi(x_0)}{(s_i + x_0) - x_0}$$

and here is the same equation rearranged to solve for s_i

$$s_i = \frac{h\phi(x_0)}{\phi(x_0) - \phi(x_i)}$$

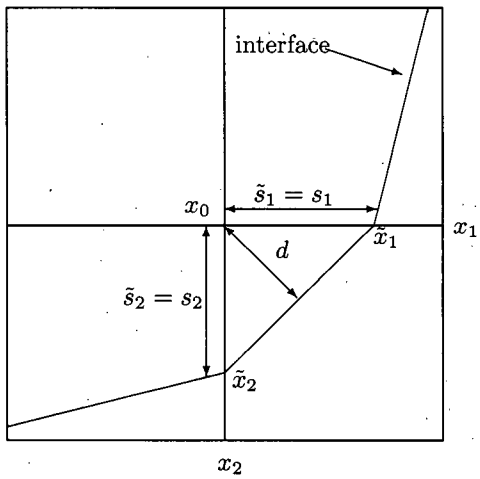

Figure 2.2: Two neighbors of the boundary node x_0 are on the other side of the interface.

where ϕ represents the implicit surface function and h represents the grid spacing.

After all s_i values are calculated the minimum value in each dimension $\tilde{s}_i$ is kept since only the distance to the closest point on the interface is needed. Here is the equation which approximates the gradient of the distance function

$$\left(\frac{d}{\tilde{s}_1}\right)^2 + \left(\frac{d}{\tilde{s}_2}\right)^2 + \cdots + \left(\frac{d}{\tilde{s}_n}\right)^2 = 1$$

where the right side is 1 and the left side represents an upwind approximation to the gradient. Here is the equation rearranged to solve for d

$$d = \frac{\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_n}{\sqrt{(\tilde{s}_2 \cdots \tilde{s}_n)^2 + (\tilde{s}_1 \tilde{s}_3 \cdots \tilde{s}_n)^2 + \cdots + (\tilde{s}_1 \cdots \tilde{s}_{n-1})^2}}$$

repeat the equation for all boundary nodes and then run the FMM.

The fast marching method returns a function u which represents a distance function but not a signed distance function. Fortunately, the sign at each node is already known because the interface moves less than a grid size, h, throughout the calculation so the nodes in the signed distance function have the same sign as those in the initial implicit surface function, ϕ. Thus, each node in the distance function must be multiplied by its sign in ϕ in order to calculated its signed distance value.

2.1.3 Extension Velocity

One of the most fundamental level set methods solves an initial value PDE for motion in the normal direction. The equation for motion in the normal direction is

$$\begin{aligned} \phi_t(t,x) + F(x)\|\nabla\phi(t,x)\| &= 0 \\ \phi(x, t=0) \text{ given} & \end{aligned} \tag{2.2}$$

It is possible that the original speed function, F, is only defined on the interface and needs to be extended onto all level sets. It is also possible that the speed function does not move the level set representation in a nice manner. The effect of the speed function could cause the representation to develop a large or small gradient magnitude, so ϕ becomes very steep, flat or non-monotone. In both cases the level set equation can be restated as

$$\begin{aligned} \phi_t(t,x) + F_{ext}(x)\|\nabla\phi(t,x)\| &= 0 \\ F_{ext} = F \text{ on } \phi(t,x) &= 0 \end{aligned} \tag{2.3}$$

where F_{ext} is the extension velocity. Extension velocity is a common technique used to extend the definition a speed function from the zero level set onto neighboring level sets [1]. This speed function is a parameter for a level set method calculation. The only restriction on the extension velocity is that the value of the extension velocity should be equal to the value of the original speed function at the zero level set. This restriction is set to ensure that the motion of the implicit surface is the same under (2.2) or (2.3).

The stipulation put on the extension velocity in this implementation is

$$\begin{aligned} \nabla F_{ext} \cdot \nabla u &= 0 \\ F_{ext} = F \text{ on } \phi(t,x) &= 0 \end{aligned} \tag{2.4}$$

where F_{ext} is the extension velocity and u is a signed distance function of the implicit surface function, ϕ. The speed function, F_{ext}, can not change in the normal direction of the interface but can only change tangential to the interface. Thus, F_{ext} is constant along the normal direction of the interface, so ϕ's isosurfaces will maintain their original spacing while being evolved by (2.3).

In this implementation, the fast marching method is used to construct an extension velocity and a signed distance function for an initial interface at the same time [1]. The signed distance function is calculated as was described in the previous section, but an additional function is added to the update method in order to calculate the extension velocity.

Firstly, extension velocity values, F_{ext}, must be calculated for boundary nodes. A method for identifying these boundary nodes was described in the previous section, as well as a method for calculating the distance, d, between the boundary node and the interface. There are two ways to calculate the extension

velocity value at boundary nodes. The first way returns the original speed, F, defined at the update node. This method must be used if the original speed is only defined on grid nodes. The second method assumes that a function for the original speed is defined throughout the entire domain or at least continuously near the interface. In this case, a weighted average of speed values is taken for the points $\tilde{x}_i$ which were used to compute d. Here is the extension velocity equation which is equivalent to solving (2.4) locally on a grid cell

$$0 = \left(\frac{F_{ext}(x_0) - F(\tilde{x}_1)}{\tilde{s}_1}, \cdots, \frac{F_{ext}(x_0) - F(\tilde{x}_n)}{\tilde{s}_n}\right) \cdot \left(\frac{d}{\tilde{s}_1}, \cdots, \frac{d}{\tilde{s}_n}\right)$$

where $F_{ext}(x_0)$ is the extension speed for the point x_0 and $\tilde{x}_i$ is the point where the interface intersects with the Cartesian grid lines as illustrated in Figure 2.2. Here is the equation when solved for $F_{ext}(x_0)$

$$F_{ext}(x_0) = \frac{\frac{1}{\tilde{s}_1^2}F(\tilde{x}_1) + \frac{1}{\tilde{s}_2^2}F(\tilde{x}_2) + \cdots + \frac{1}{\tilde{s}_n^2}F(\tilde{x}_n)}{\frac{1}{\tilde{s}_1^2} + \frac{1}{\tilde{s}_2^2} + \cdots + \frac{1}{\tilde{s}_n^2}}$$

Thus, the boundary conditions for (2.4) are defined and next the method for updating extension values for nodes not on the boundary needs to be defined. The extension values are updated at the same time as the distances are updated in the fast marching method. These extension values are calculated by using the same nodes as are used in the update of ϕ. These update nodes, x_i where $i \in \{1, \cdots, n\}$, are defined in Algorithm 4 as the nodes at which the a_i values are chosen when x_0 is being updated. An upwinded gradient for the signed distance value and the extension velocity value is chosen at each node to solve (2.4). Here is the upwind version of (2.4) where gradients are taken assuming that there is a node in every dimension which is used to update ϕ

$$\left(\frac{u(x_1) - u(x_0)}{h}, \cdots, \frac{u(x_n) - u(x_0)}{h}\right) \cdot \left(\frac{F_{ext}(x_1) - F_{ext}(x_0)}{h}, \cdots, \frac{F_{ext}(x_n) - F_{ext}(x_0)}{h}\right) = 0$$

and here is the equation rearranged to solve for $F_{ext}(x_0)$, the extension value for the point x_0,

$$F_{ext}(x_0) = \frac{F_{ext}(x_1)(u(x_1) - u(x_0)) + \cdots + F_{ext}(x_n)(u(x_n) - u(x_0))}{(u(x_1) - u(x_0)) + \cdots + (u(x_n) - u(x_0))}$$

2.2 Implementation

This section discusses some general implementation choices. It also states and describes how to use the three methods which were added to Ian Mitchell's Level Set Toolbox [6].

2.2.1 Implementation Choices

The purpose of this implementation is to add new functionality to Ian Mitchell's Level Set Toolbox [6]. Previously, the toolbox only had functions to solve time-dependent Hamilton-Jacobi PDEs, but this new functionality allows some static Hamilton-Jacobi PDEs to be solved. Since the toolbox is implemented fully in Matlab, the new fast marching method was also implemented in Matlab so that the portability of the toolbox would not be lost. Unfortunately, this implementation was extremely slow and even optimizing the code with help from Matlab's profiler could not speed up the implementation very much. Thus, the code had to be re-implemented in C and called from Matlab through a MEX interface. This greatly increased the speed of execution. An option was to only implement the heap in C, because in Matlab the heap was the slowest part of the code. This option was not chosen because Dijkstra's method is heavily dependent on heap functions. In Algorithm 1 each `ExtractMin` call requires one heap function call and each `Update` call requires up to four heap function calls. Thus, $5N$ heap function calls would be made for a graph with N grid nodes, so making these calls through MEX interfaces would have added too much overhead to program execution.

2.2.2 Dijkstra

This method implements Algorithm 1 and it allows for a specific update function to be chosen.

```
value = dijkstra(grid, bdryData, bdryMask, cost, update_func)
```

The `grid` parameter is a grid structure as defined in Ian Mitchell's Level Set Toolbox [6]. Each dimension in the grid must have the same `grid.dx` value. The `bdryData` parameter is the array defining costs at boundary nodes and the `bdryMask` is a boolean array defining the boundary nodes. The `cost` parameter is an array which defines the cost to travel to a node from one of its neighbors. All three matrices must have the same size as defined in the grid structure. Finally the `update_func` string specifies the desired update function to be used. The fast marching method, Algorithm 4, can be specified by the string `'fmm'`, and Dijkstra's update procedure, Algorithm 3, can be specified by the string `'dijkstra'`.

2.2.3 Signed Distance

This method computes a signed distance function for a given implicit surface function. The method first determines which nodes are boundary nodes and calculates the value at these nodes. The `signedDistance` function then calls the `dijkstra` function and passes in the boundary nodes and boundary values as parameters. The `dijkstra` function computes a signed distance representation of the implicit surface function. The specifics of the method are explained in Section 2.1.2.

```
signed_dist = signedDistance(grid, initFunc)
```

The `grid` parameter has the same constraints as in `dijkstra`. The `initFunc` parameter defines an implicit surface function. This array must have the same dimensions as defined in the grid structure.

2.2.4 Extension Velocity and Signed Distance

This method uses a fast marching method to construct a signed distance function and an extension velocity for an implicit surface function and an initial speed function. The specifics of the method are discussed in Section 2.1.3.

```
[ext_velocity, signed_dist] =
extendVelocity(grid, initFunc, speedOption, speed_func)
```

The `grid` parameter has the same constraints as in `dijkstra`. The `initFunc` parameter defines an implicit surface function. This array must have the same dimensions as defined in the grid structure. The `speedOption` parameter specifies whether the speed at boundary nodes will be calculated at only those nodes (`1`) or whether they will be averaged by speeds from the surrounding interface (`2`). The difference between the two choices is discussed in Section 2.1.3. Finally, the `speed_func` parameter must be defined. This parameter can either be a function handle or an array. If the parameter is a function handle then it refers to a Matlab method which will calculate the speed at a domain node. This function handle has one parameter, `gridPoint`, a vector representing a point in the domain. The prototype for `speed_func` is `speed = speed_func(gridPoint)`. If the parameter is an array then it defines the speed at each grid node and the array must have the same dimensions as defined in the grid structure. In the case when `speedOption` is `1` then both types of `speed_func` parameters can be passed in, otherwise only a function handle can be used as a parameter for `speed_func`.

2.3 Examples

The purpose of this section is to use some example problems to test the implementation of the methods described in Section 2.2.

2.3.1 Shape From Shading

This section describes the function `fmmPublish/examples/shapeFromShading`.

In general, the shape from shading method allows the shape of a surface to be determined from a shaded image of the surface. In this particular case, an Eikonal equation is being solved to determine a viscosity solution for a Lambertian surface illuminated by a single distant light source. The test problems

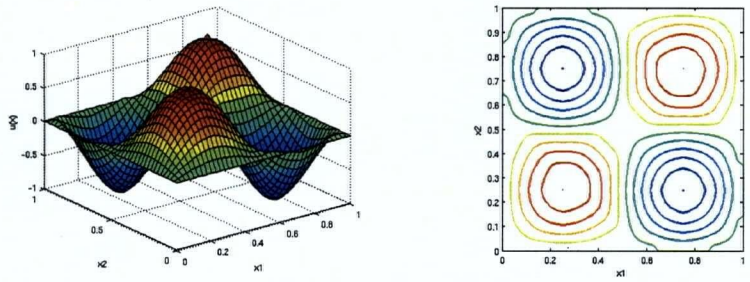

Figure 2.3: Surface (left) and contour (right) plots for the smooth solution of the shape from shading problem. They are generated by the call `shapeFromShading(41, 1, 'fmm')`. The domain of this example is 41 by 41 nodes. The fast marching method is used as the update method to solve the problem.

were given in [8], while their analytical solutions were given in [9]. The Eikonal equation is defined on a two dimensional domain from $[0, +1]^2$.

The cost function is

$$c(x, y) = 2\pi\sqrt{[\cos(2\pi x)\sin(2\pi y)]^2 + [\sin(2\pi x)\cos(2\pi y)]^2}.$$

Boundary nodes are defined on the exterior of the domain as well as on five interior nodes. The interior boundary nodes are defined at the following points in the domain: $\left(\frac{1}{4}, \frac{1}{4}\right), \left(\frac{3}{4}, \frac{3}{4}\right), \left(\frac{1}{4}, \frac{3}{4}\right), \left(\frac{3}{4}, \frac{1}{4}\right), \left(\frac{1}{2}, \frac{1}{2}\right)$. The value of the exterior boundary nodes is always zero. The boundary value at the interior nodes depends on the type of solution desired. A smooth solution defines the values as $[+1, +1, -1, -1, 0]$ respectively, with the resulting analytical solution

$$T(x, y) = \sin(2\pi x)\sin(2\pi y).$$

The continuous but not smooth solution defines the values as $[+1, +1, +1, +1, +2]$ respectively, with the resulting analytical solution

$$T(x,y) = \begin{cases} \max\left(\begin{array}{c} |\sin(2\pi x)\sin(2\pi y)| \\ 1 + \cos(2\pi x)\cos(2\pi y) \end{array}\right) & \text{if} \left\{\begin{array}{c} |x + y - 1| < \frac{1}{2} \\ |x - y| < \frac{1}{2} \end{array}\right\} \\ |\sin(2\pi x)\sin(2\pi y)|, & \text{otherwise} \end{cases}.$$

The smooth solution for the shape from shading problem viewed in Figure 2.3 is generated by the function call below. This function call also generates a continuous but not smooth solution and its results can be viewed in Figure 2.4. The domain for both figures is 41 by 41 nodes and both examples use `'fmm'` update procedure. They both demonstrate the use of `dijkstra`.

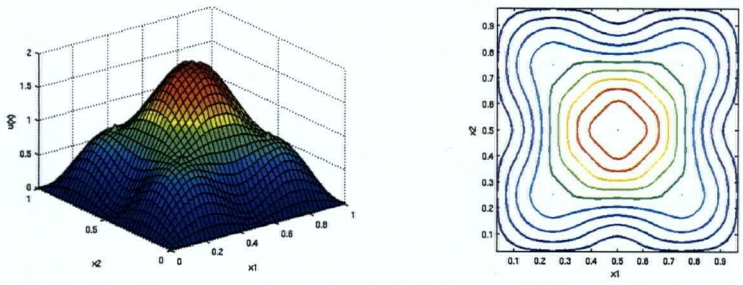

Figure 2.4: Surface (left) and contour (right) plots for the continuous but not smooth solution of the shape from shading problem. They are generated by the call `shapeFromShading(41, 2, 'fmm')`. The domain of this example is 41 by 41 nodes. The fast marching method is used as the update method to solve the problem.

```
[value, g] =
shapeFromShading(numNodes, isSmoothSolution, update_func) :
```

The function above demonstrates the fast marching method by using it to solve the shape from shading problem. The `numNodes` parameter allows the user to specify the number of grid nodes in each dimension. This parameter has a precondition of `numNodes` $= 4m - 1$, where $m > 1$, which is applied to ensure that the interior boundary conditions occur at nodes. The `isSmoothSolution` parameter specifies whether the smooth solution is desired (`true`) or whether the merely continuous one is desired (`false`). Finally the `update_func` parameter specifies the desired update function to be used in method. The fast marching method can be specified by `'fmm'` and Dijkstra's update procedure can be specified by `'dijkstra'`.

2.3.2 Extension Velocity of a Circle

This section describes the function
`fmmPublish/examples/circleExtensionVelocity`.

The purpose of this function is to demonstrate how an extension velocity and a signed distance function can be calculated from an initial function. The initial function chosen in this example is the signed distance from a circle multiplied by a factor of 5. The circle has a radius of 1 and is centered at $[0, 0]$. The initial function is specified by the following equation

$$\phi(x, 0) = 5(\|x\|_2 - 1). \tag{2.5}$$

Thus, the initial function will be a cone with steep walls and a gradient magnitude of 5 as demonstrated in Figure 2.6. The domain of this initial function is

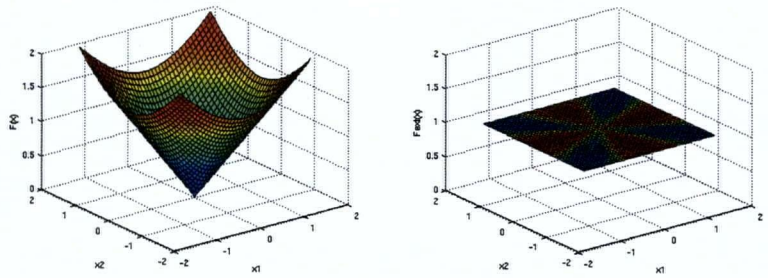

Figure 2.5: Original (left) and extension (right) velocity functions of Equation (2.5). The images are generated by the call `circleExtensionVelocity(41, 2)`.

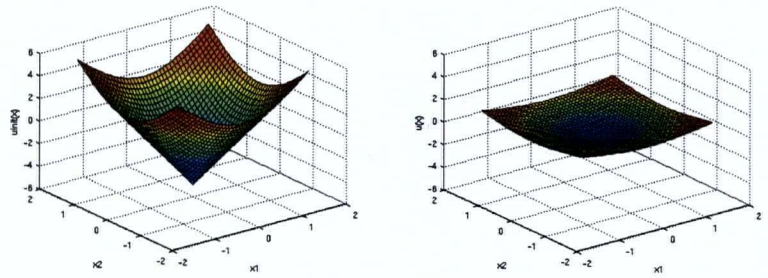

Figure 2.6: Initial (left) and signed distance (right) functions for the implicit surface function in (2.5). The images are generated by the call `circleExtensionVelocity(41, 2)`. Note the vertical scale.

calculated in a 2-dimensional space of $[-1.5, +1.5]$. The original speed at each grid point x_i is the two-norm of that position, $F(x_i) = \|x_i\|$. An extension velocity is not allowed to change normal to the interface and the original speed doesn't change tangential to the interface. The original speed is defined as 1 on the entire interface which is a circle with a radius of 1, so the extension velocity will be a flat plane at 1. The signed distance will be a shallower cone than the initial function and it will have a gradient magnitude of 1.

The extension velocity can be seen in Figure 2.5 and the signed distance for the problem described above can be seen in Figure 2.6. They are generated by the `circleExtensionVelocity` function which demonstrates the use of `extendVelocity`.

```
[ext_velocity, signed_dist, grid] =
circleExtensionVelocity(nodes, speedOption) :
```

The `numNodes` parameter allows the user to specify the number of grid nodes in each dimension. The `speedOption` parameter specifies whether the speed at

boundary nodes will be calculated at only those nodes (1) or whether they will be averaged from speeds from the surrounding interface (2). See Section 2.1.3 for further details about the speed options.

2.3.3 Movement of a Circle in Normal Direction

This section describes the function `fmmPublish/examples/normalCircle`.

The purpose of this function is to demonstrate how to solve a level set equation, for example a time dependent Hamilton-Jacobi equation, while updating its velocity with an extension velocity. The Level Set Toolbox is used to handle the time dependent PDE. This section demonstrates the capability of combining the code from this thesis with the features already in the Toolbox.

In this example, an initial function, ϕ, is five times a signed distance to a circle with radius 1 centered at $[0,0]$: $\phi(x,0) = 5(\|x\|_2 - 1)$. This initial function will be a cone and its walls will have a gradient magnitude of 5. The domain of this initial function is calculated in a 2-dimensional space of $[-1.5, +1.5]^2$. The level set equation, (2.2), moves the initial function in its normal direction. Thus, the surface at the zero level set will be a circle and will expand into a larger circle which is still centered at the origin.

In this case, the initial speed function at each grid point is the two-norm of that position, $F(x) = \|x\|_2$. Thus, without the extension velocity the initial cone will move faster the further away it is from the origin. This will result in the cone becoming more shallow. The result with the extension velocity will be slightly different. An extension velocity is not allowed to change normal to the interface and the original velocity doesn't change tangential to the interface. The original velocity is defined as 1 on the entire interface, so the extension velocity will have a value of 1 throughout the entire domain. Thus, the walls of the cone will move at approximately the same rate and the gradient magnitude of the walls should remain near 5. The bottom of the cone will flatten out.

The different results which the extension velocity causes in the `normalCircle` example can be viewed in Figures 2.7 and 2.8. Both figures demonstrate that without an extension velocity the gradient magnitude decreases with time and with an extension velocity the gradient magnitude remains the same for most nodes.

```
[g, speedStore] = normalCircle(useExtVel, accuracy, numNodes)
```

The method above computes an implicit surface function at four timesteps and at each timestep the contours of the function are plotted. The implicit functions for each timestep are returned in a cell vector called `speedStore`. The grid which represents the domain of the implicit function is also returned. This `grid` parameter is a grid structure as defined in Ian Mitchell's Level Set Toolbox [6]. `useExtVel` is a boolean parameter which is true if an extension velocity will be used and false otherwise. The `accuracy` parameter specifies the degree of accuracy used to solve the level set method. The choices are `'low'`, `'medium'`,

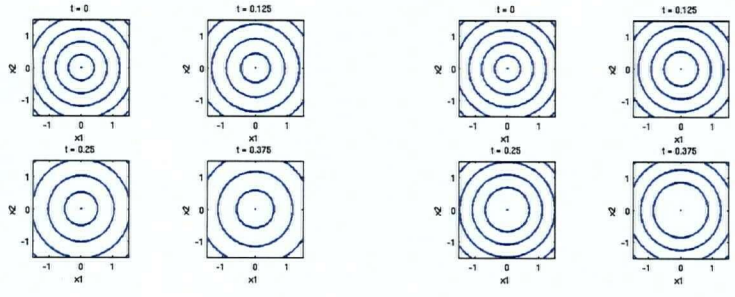

Figure 2.7: Contour plots for a circle moving in its normal direction. The left image demonstrates the level set method run with its regular speed, $F(x) = \|x\|_2$. It is generated by the call `normalCircle(0, 'medium', 41)`. The contours become further apart as the cone shaped surface becomes shallower. The right image demonstrates the level set method with an extension velocity. It is generated by the call `normalCircle(1, 'medium', 41)`. The contours stay the same width apart because the whole surface is moving in its normal direction at the same rate.

`'high'` and `'veryHigh'`. The `numNodes` parameter allows the user to specify the number of grid nodes in each dimension.

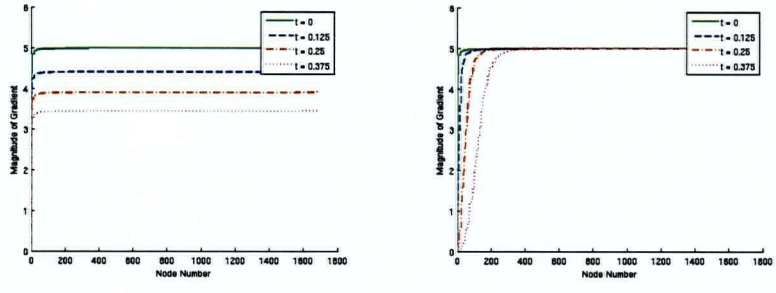

Figure 2.8: Ordered gradient magnitude plots for the surface at different times. In the plots the nodes are sorted by gradient magnitude. The left image demonstrates the level set method run with its regular velocity. It is generated by the call `normalCircle(0, 'medium', 41)`. The most common gradient magnitude decreases as the surface gets shallower. The right image demonstrates the level set method with an extension velocity. It is generated by the call `normalCircle(1, 'medium', 41)`. The most common gradient magnitudes are all 5.

Chapter 3

Reachable Sets on a Manifold

3.1 Background

3.1.1 Differential Algebraic Equations

A DAE is a differential equation which is comprised of algebraic (no derivative is present) and differential (derivatives are present) terms [2]. Here is the general implicit form for a DAE:

$$F(t, y, y') = 0$$

where $\partial F/\partial y'$ may be singular or nonsingular depending on the index of the DAE. The index of a DAE refers to the number of times differentiation must be carried out on the system until an ordinary differential equation is obtained for y. In other words, the number of differentiations required to solve uniquely for y' in terms of y and t [2]. This research only deals with DAEs of index-1, since we need to differentiate the algebraic term in order to specify the velocity field. More specifically this research deals with semi-explicit DAEs of index-1 or in other words an ODE with constraints where the constraints represent a manifold on which the system state evolves. The general form for this type of DAE is

$$\begin{aligned} x' &= f(t, x, z) \\ 0 &= g(t, x, z) \end{aligned} \tag{3.1}$$

where x represents the differential variables and z represents the algebraic variables. In this form it is easy to identify g as the constraint function since its equation does not contain derivatives.

3.1.2 Closest Point Method

The closest point method [10] allows a PDE to be extended off the manifold onto the entire domain. The problem, which is originally only defined on the surface, is embedded onto the whole cartesian grid so that standard level set techniques can be used to solve it. The solution on the manifold can then be obtained by only considering the results along the manifold. This approach is

an alternative to either parameterizing the constraint manifold or triangulating the constraint manifold in order to solve the PDE. The embedding approach is accurate and the PDE is easy to solve numerically once the embedding is complete because there are a lot of well known techniques for solving PDEs on Cartesian grids. Fortunately, the embedding approach used in the closest point method is easy to implement as well.

The embedding approach creates an extended function, which involves replacing surface gradients, defined only on the manifold surface, with standard gradients which are defined on the entire computation domain, $\Re^n$. A requirement for this procedure is that the embedded PDE must be a natural extension of the surface PDE. The simplest way to meet this requirement is for the gradient to be constant along the direction normal to the surface. This means that the gradient will only change in the direction of the surface and no artificial rates of change will be added to the PDE.

The closest point method uses an extended function which is created by composing the original surface function onto a closest point operator. A simple and accurate way to define this closest point operator, $CP(x)$, for a point, x, is to have the operator return the closest point to x on the surface. This operator is used before each timestep in the level set solve to reinitialize the implicit surface function, ϕ, as $\phi(CP(x))$. This will ensure that for a point on the surface, x_0, all points, x_n, along its normal direction will have the same function value, $\phi(x_0) = \phi(x_n)$. Thus, the gradient of $\phi(CP(x))$ will be constant along the normal direction and will only be able to change in directions tangential with the surface.

This closest point operator will create a well defined and smooth function near the surface, but may have discontinuities away from the surface where a point is equidistant from two points on the surface. The greater the local curvature in the manifold the closer the discontinuity will be to the manifold, but the discontinuity should be at least several grid cells away from the manifold in order to achieve accurate results. The closest point method allows for a variety of difficult surfaces to be represented such as open and closed surfaces in any codimension. The codimension of a surface with dimension p in a domain of dimension n is $n - p$. We only study codimension 1 surfaces in this thesis, but the closest point method should work for higher codimensions.

Research in [15] describes a method for calculating the closest point operator. This method uses Voronoi Regions to calculate the operator. Analytic calculation, using standard numerical optimization techniques or using a tree-based algorithm for triangulated surfaces are options suggested in [10] in order to calculate the closest point operator. The research in the thesis solves a constrained non-linear optimization problem in order to determine the closest points.

3.1.3 Reachable Sets

Model checking is a method used to verify or validate complex engineering systems. A common form of model checking is to verify whether a model in evolution can enter into an unsafe state. This form of model checking involves the

computation of a reachable set of which there are two types: forwards reachable sets and backwards reachable sets. Computing forwards reachable sets involves evolving an initial set of states forward in time in order to determine the set of states reached by their trajectories. Computing backwards reachable sets involves declaring a set of target states and then determining the set of start states whose trajectories reach that target set. In [14], a method was developed to solve backwards reachable sets using level set methods. This method expresses the reachable set problem as a Hamilton Jacobi equation. Level set methods are a well known technique for solving the Hamilton Jacobi Equation (HJE), thus they can be used to solve backwards reachable sets.

In order to solve for the backwards reachable set, a continuous system with dynamics, $\dot{x} = f(x)$, is defined in order to move a set of initial conditions. The backwards reachable set is the subset of these initial conditions that are driven into the target set, $\mathcal{T}_0$. The state of the system is defined by x and the target set is defined as a zero sublevel set

$$\mathcal{T}_0 = \{x \in \Re^n \mid \phi_0(x) \leq 0\}$$

of a scalar function $\phi(x, 0) = \phi_0(x)$. The backwards reachable set is defined as $\mathcal{T}(\tau)$ over a finite horizon $\tau < \infty$, the trajectory of the system is defined by $x(\cdot)$ and the state of the trajectory at time τ is defined by $x(\tau)$. Thus, a formal definition of the reachable set, $\mathcal{T}(\tau)$, is the set of $x(0)$ such that $x(\tau) \in \mathcal{T}_0$. It can also be defined in implicit surface notation as a zero sublevel set

$$\mathcal{T}(\tau) = \{x \in \Re^n \mid \phi(x, -\tau) \leq 0\}$$

where $\phi(x, t)$ is the solution to the following HJE which is solved backwards in time from $t = 0$ to $t = -\tau \leq 0$

$$0 = D_t\phi(x, t) + f(x, t) \cdot \nabla\phi(x, t) \tag{3.2}$$

The reachable tube is another object sought from this algorithm. The reach set is the set of states occupied by trajectories at a specific point in time τ; alternatively, the reach tube is the set of states that have been traversed by the trajectories from $t = 0$ to $t = \tau$ [5]. More formally, it is the set of $x(0)$ such that $x(s) \in \mathcal{T}_0$ for some $s \in [0, \tau]$. The following HJE computes the reach tube

$$0 = D_t\phi(x, t) + \min[0, f(x, t) \cdot \nabla\phi(x, t)] \tag{3.3}$$

The minimization with zero in the second term constrains the temporal derivative to a positive sign. This restriction keeps the reachable tube from shrinking as $t \to -\infty$.

This research calculates reachable tubes for the two-dimensional toy problem in Section 3.3.1 and it calculates reachable sets for the three-dimensional toy problem in Section 3.3.2.

3.2 Implementation

3.2.1 Level Set Methods

The implementation is accomplished by using Ian Mitchell's Level Set Toolbox for Matlab [6]. The toolbox uses upwinding schemes to solve HJE which move under a convective field such as (3.2) and (3.3). The solutions displayed in this work will use second order accuracy unless otherwise stated. The convective field is approximated by `termConvection`, the upwinding spatial derivative is approximated by `upwindFirstENO2` and the temporal derivative is approximated by `odeCFL2`. The function `termRestrictUpdate` is used in order to restrict the temporal derivative to a positive sign such as required in (3.3). If the restriction is used then a reach tube is calculated, otherwise a reach set is calculated.

3.2.2 Differential Algebraic Equations

The DAE specifies the velocity field required by `termConvection`. Since the DAE required for this research is an ODE with constraints, the constraint term must be differentiated with respect to t in order to specify a velocity field for all variables. Here is the velocity field, which is derived from differentiating (3.1) and rearranging to solve for z',

$$\frac{dx}{dt} = f(t,x,z)$$
$$\frac{dz}{dt} = \frac{-\frac{\partial g(t,x,z)}{\partial x} f(t,x,z) - \frac{\partial g(t,x,z)}{\partial t}}{\frac{\partial g(t,x,z)}{\partial z}}$$

3.2.3 Closest Point

This implementation of the closest point operator assumes that the constraint surface is represented by an implicit surface function. That implicit surface function is then used to compute a signed distance function using the methods in Chapter 2.1.2.

Algorithm 5 demonstrates how the closest point operator is computed using the signed distance function and the implicit surface function. Table 3.1 describes the parameters used in the closest point operator algorithm, while Tables 3.2 and 3.3 describe the inputs and outputs for the functions used in the closest point operator algorithm.

The algorithm takes in a signed distance representation of the manifold, `signedDist`. It then sorts, `SortAscending`, the nodes in this representation depending on the absolute value of their distance.[1] The nodes are then traversed

[1] A topological sort, where each node occurred after its neighbour of lowest value, would be faster than a strict sort by distance but in practice the optimization routine dominates running time. Thus, the fraction of time saved by using a topological sort is negligible.

Input: `signedDist`, `constraint`
`sorted` = `SortAscending(Abs(signedDist))`
foreach $x_i \in$ `sorted` **do**
  if `IsInterfaceNeigh`(x_i, `signedDist`) **then**
    $\tilde{x}$ = `TravelGradient`(x_i, `signedDist`)
  else
    x_s = `SmallestNeigh`(x_i, `signedDist`)
    $\tilde{x}$ = `closestPoint`(x_s)
  x_{cp} = `CpOptimize`(x_i, $\tilde{x}$, `constraint`)
  `closestPoint`(x_i) = x_{cp}
Output: `closestPoint`

Algorithm 5: Algorithm which computes the closest point operator.

Parameter Name	Type	Description
`signedDist`	Array	Signed distance representation of the manifold
`constraint`	Function handle	Functional representation of the implicit surface function for the manifold
`sorted`	Vector	Sorted list of nodes depending on their distance to the manifold
`closestPoint`	Cell vector with array elements	Closest point on the manifold for each grid node

Table 3.1: Description of parameters used in the closest point operator algorithm.

in their sorted order from the smallest distance to the manifold. If the node is adjacent to the interface, `IsInterfaceNeigh`, then the algorithm takes steps, on the order of Δx, along the gradient until the interface is reached. This step was needed in order to give an accurate initial point $\tilde{x}$ for the optimization problem. To compute the initial guess for nodes which are not adjacent to the interface first the neighbor which is closest to the interface, `SmallestNeigh`, is found. The initial guess becomes this neighbor's closest point, `closestPoint`, which has already been computed. The initial guess, $\tilde{x}$, is used in a nonlinear constrained optimization problem, `CpOptimize`, which finds the closest point on the manifold. Here is the optimization problem:

$$\begin{aligned} &\min \|x_{cp} - x_i\|^2 \\ &\text{subject to } 0 = \texttt{constraint}(x_{cp}) \end{aligned}$$

where x_{cp} is the closest point on the interface for node x_i and `constraint` (x) is the function representing the manifold. The constrained optimization problem is solved using Matlab's `fmincon` function.

The closest point function is composed with ϕ to reinitialize ϕ as an em-

Function Name	Inputs
`SortAscending`	A List of real numbers
`IsInterfaceNeigh`	A grid node, `signedDist`
`TravelGradient`	A grid node, `signedDist`
`SmallestNeigh`	A grid node, `signedDist`
`CpOptimize`	A grid node, an initial guess for position of closest point, `constraint`

Table 3.2: Description of functions and their input parameters used in the closest point operator algorithm.

Function Name	Outputs
`SortAscending`	List sorted in ascending order
`IsInterfaceNeigh`	True if the node is beside the interface, false otherwise
`TravelGradient`	Estimated position of closest point on manifold along the gradient
`SmallestNeigh`	The neighboring node with the smallest `signedDist` value
`CpOptimize`	Closest point on the manifold to the grid node

Table 3.3: Description of functions and their output parameters used in the closest point operator algorithm.

bedded surface. The closest point function returns a point, $x_0 = CP(x_n)$, on the manifold, but this point is not necessarily a grid node. Thus, the function value of $\phi(x_0)$ must be interpolated in order to reinitialize ϕ at each node. The method used for this interpolation calculation is Matlab's `interpn` with cubic accuracy.

3.3 Examples

The validity of the method developed in this research is demonstrated in some simple toy problems. Matlab's DAE solver can be used to generate sample trajectories for semi-explicit DAEs of index-1. A trajectory's position at specific times can be calculated from these trajectories and compared to the implicit surface calculated by the level set implementation. These problems are easy to conceptualize and visualize thus the results of our level set implementation can be clearly grasped.

All examples are calculated on a Linux box with a Trison Pentium 4 3GHZ processor, 1G RAM and 80G Harddrive. The software used was Suse linux 10.1, Kernel version 2.6.16.27-0.9-smp, Matlab 7.3.0.298 (R2006b) and GCC version 4.1.0.

3.3.1 Two-dimensional DAE Toy Problem

The DAE for the two-dimensional toy problem is as follows

$$\begin{aligned} x_1 &= \sin(\pi x_2) \\ \dot{x}_2 &= 1.5 + x_1 \end{aligned} \tag{3.4}$$

where the first equation is the constraint. In order to derive the velocity field the constraint is differentiated. The velocity field is as follows

$$\begin{aligned} \dot{x}_1 = v_1 &= \pi v_2 \cos(\pi x_2) \\ \dot{x}_2 = v_2 &= 1.5 + x_1 \end{aligned}$$

The constraint surface is a sine curve in the vertical direction, while the dynamics move along the sine curve faster on the right side of the curve than on the left.

The implicit surface representing the initial reach set is a hyperplane with a normal of $[0, 1]$ which passes through the origin. The problem was defined on a domain of $[-1.5, +1.5] \times [-0.2, +3.7]$, while the distance between grid nodes is defined as 0.02. The PDE is executed from 0 to 3 time units and reachable sets are calculated every 0.25 time units. The toy example is calculated using the closest point method and without using the method in order to demonstrate the purpose of the method. It takes 620 seconds to solve the problem using the closest point method and 130 seconds to solve it without using the

closest point method. The closest point method takes additional time for calculation because it has to interpolate the closest point values for all nodes at each timestep. Figure 3.1 demonstrates that the implicit surface which uses the closest point method is perpendicular to the constraint surface, while the implicit surface which does not use the method is not perpendicular to the constraint. The solution using the closest point method is calculating the reachable set on the manifold, while the solution without the closest point method is calculating the reachable set for the whole domain. The calculation solely on the manifold makes it much clearer to view where the reachable set intersects with the manifold and it makes it more numerically stable as well.

The closest point method also improves the quantitative solution as time progresses in the calculation. This improvement is shown by using Matlab's DAE solver on (3.4) to very accurately estimate the position of the trajectory at each timestep, then the implicit surface value is interpolated at each position of the estimated trajectory. The values at each timestep can be seen in Table 3.4, and since the reach set interface is defined as the zero level set all values should be zero. In the first couple of timesteps the method which does not use the closest point method has a slight advantage, but midway through the calculation that method loses its advantage and becomes far more inaccurate. The closest point method allows the level set calculation to maintain better accuracy throughout the calculation.

A convergence plot for $t = 3.0$ time units, in Figure 3.2, shows superlinear convergence for the two dimensional toy example. The numerical values for the rates of convergence are calculated in Table 3.5. We are not sure why the convergence rate becomes negative when 600 grid nodes are used. These values average out to a convergence rate of 1.3910, which is not quite the quadratic rate expected from the second order accurate scheme used in this research. This may be due to a slight mis-approximation in the closest point calculation where some closest points are slightly off the constraint surface.

3.3.2 Three-dimensional DAE Toy Problem with Codimension 1

The DAE for the three-dimensional toy problem is as follows

$$\begin{aligned} x_1 &= \sin(\pi x_2) \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= -x_2 \end{aligned}$$

where the first equation is the constraint and the last two equations are the differential equations. In order to derive the velocity field the constraint is differentiated. The velocity field is as follows

Timestep (time units)	No Closest Point	Closest Point	No CP - CP
0.00	0.0000	0.0000	0.0000
0.25	0.0000	0.0009	-0.0009
0.50	0.0001	0.0008	-0.0008
0.75	0.0005	0.0010	-0.0005
1.00	0.0010	0.0017	-0.0007
1.25	0.0069	0.0129	-0.0060
1.50	0.0509	0.0109	0.0400
1.75	0.1723	0.0157	0.1566
2.00	0.2439	0.0076	0.2363
2.25	0.2338	0.0098	0.2240
2.50	0.2565	0.0116	0.2450
2.75	0.2571	0.0125	0.2446
3.00	0.2634	0.0217	0.2418

Table 3.4: Interpolated implicit surface value for each timestep of the two-dimensional toy DAE problem. Values using the closest point method and values without using the method are compared.

Number Grid Nodes	Error	Rate
75	0.04923	-
150	0.0078	2.6580
300	0.002721	1.5193
600	0.002729	-0.0042

Table 3.5: Convergence of error for the last timestep, $t = 3.0$ time units, for the two dimensional toy example.

$$\dot{x}_1 = v_1 = \pi v_2 \cos(\pi x_2)$$
$$\dot{x}_2 = v_2 = x_3$$
$$\dot{x}_3 = v_3 = -x_2$$

The constraint surface is a sine curve in the x_2 direction, while the dynamics move along the sine curve in a clockwise circle in the x_2 versus x_3 plane.

The initial reachable set is a cuboid (a rectangular box) which has an infinite length in the x_1 dimension. The upper corner of the rectangle is at $(x_1, 0.25, 0.75)$ and the lower corner of the rectangle is at $(x_1, -0.25, 0.25)$. The problem was defined on a domain of $[-1.0, +1.0]^3$, while the distance between grid nodes is defined as 0.05. The PDE is executed from 0 to 2π time units and reachable sets are calculated every $\pi/4$ time units. The toy example is calculated using the closest point method and without using the method in order to demonstrate the purpose of the method. It takes 848 seconds to solve the problem using the closest point method and 392 seconds to solve it without using the closest point method.

The closest point method also slightly improves the quantitative solution for the three-dimensional example. This improvement is shown by using Matlab's DAE solver to calculate the position of many sample trajectory at each timestep. These trajectories are initiated from points along the target set, these points have a spacing of 0.02. For each trajectory, the implicit surface value at each position is interpolated. The average values at each timestep can be seen in Table 3.6 and the maximum value at each timestep can be seen in Table 3.7, since the constraint surface is defined as the zero level set all values should be zero. On average at each timestep the solution which uses the closest point method is better than the solution which does not use the method. On the other hand, the maximum value is higher at each timestep when using the closest point method.

Timestep (time units)	No Closest Point	Closest Point	No CP - CP
0.00	0.0003	0.0004	0.0000
0.7854	0.0103	0.0083	0.0020
1.5708	0.0163	0.0151	0.0012
2.3562	0.0213	0.0191	0.0023
3.1416	0.0256	0.0205	0.0052
3.9270	0.0300	0.0252	0.0048
4.7124	0.0334	0.0316	0.0018
5.4978	0.0371	0.0343	0.0028
6.2832	0.0404	0.0380	0.0024

Table 3.6: The average interpolated implicit surface value for multiple ODE trajectories at each timestep of the three-dimensional toy DAE problem. Values using the closest point method and values without using the method are compared.

Timestep (time units)	No Closest Point	Closest Point	No CP - CP
0.00	0.0036	0.0036	0.0000
0.7854	0.0479	0.0483	-0.0003
1.5708	0.0595	0.0645	-0.0050
2.3562	0.0695	0.0757	-0.0062
3.1416	0.0759	0.0799	-0.0039
3.9270	0.0827	0.0963	-0.0136
4.7124	0.0877	0.1014	-0.0137
5.4978	0.0933	0.1158	-0.0225
6.2832	0.0979	0.1273	-0.0294

Table 3.7: The maximum interpolated implicit surface value for multiple ODE trajectories at each timestep of the three-dimensional toy DAE problem. Values using the closest point method and values without using the method are compared.

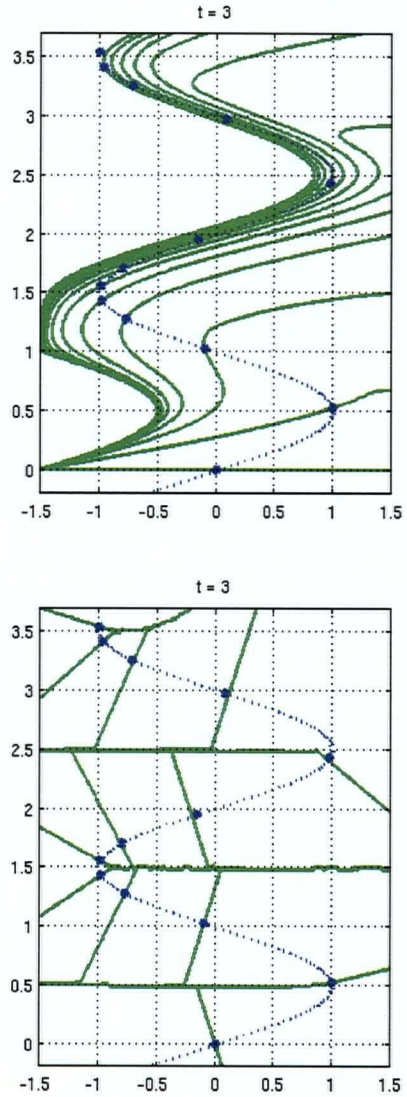

Figure 3.1: Reachable set plots for the two dimensional toy example. The top image doesn't use the closest point method, while the bottom image does use the method. In both images, the dotted curve represents the manifold, the stars represent the solution calculated using Matlab's DAE solver at intervals of 0.25 time units and the solid curves represent the reachable set solution at intervals of 0.25 time units. Notice how the reachable set is perpendicular to the manifold near the manifold when using the closest point method.

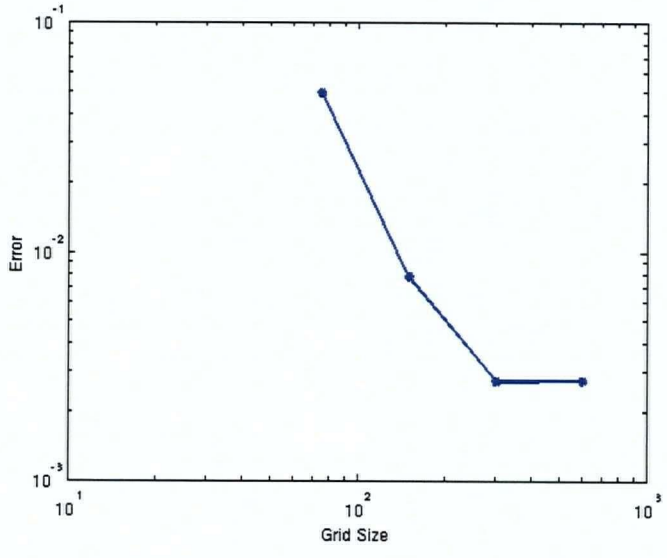

Figure 3.2: Convergence rates for the last timestep, $t = 3.0$ time units, for the two dimensional toy example. The x-axis defines the number of grid nodes in the x_1 dimension of the grid, they are as follows 75, 150, 300 and 600.

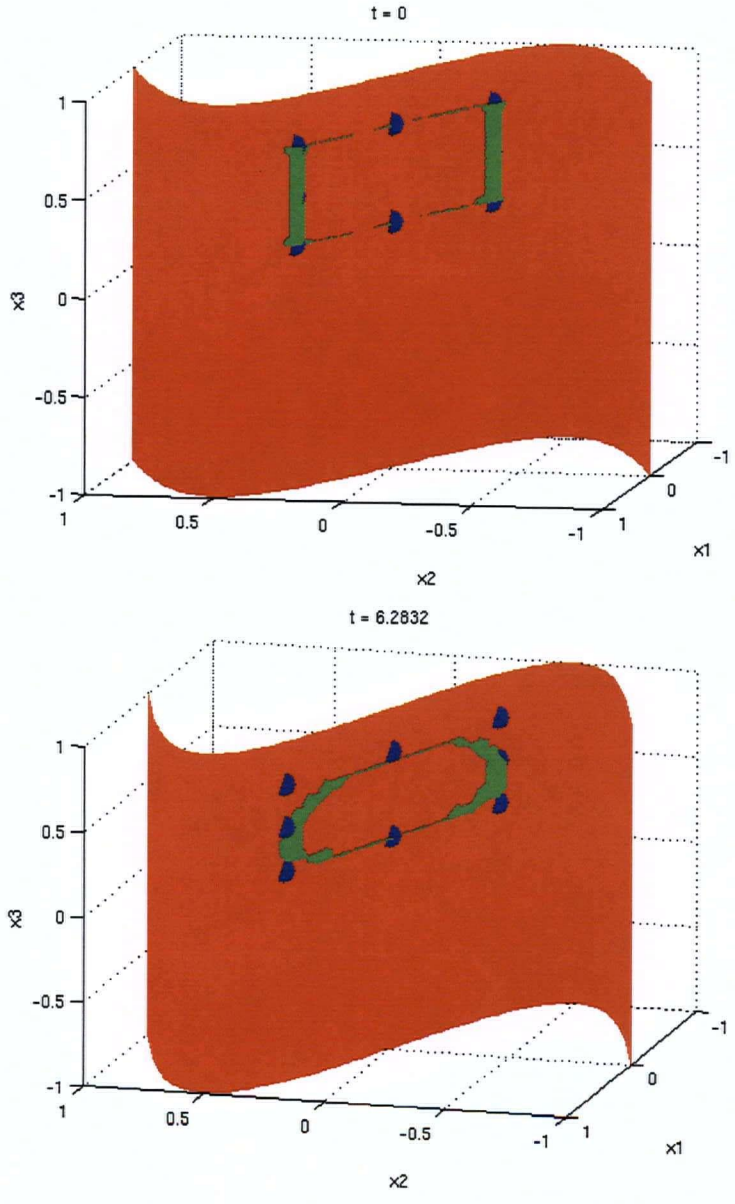

Figure 3.3: Reachable set plots for the three dimensional toy example in 3D. In both images, the red surface (darker surface) is the manifold, the green curve (lighter curve) is the reachable set on the manifold and the blue dots represent the solution calculated using Matlab's DAE solver. The top image represents the system at the starting time, $t = 0$ time units, and the bottom image represents the system at the end time, $t = 2\pi$ time units.

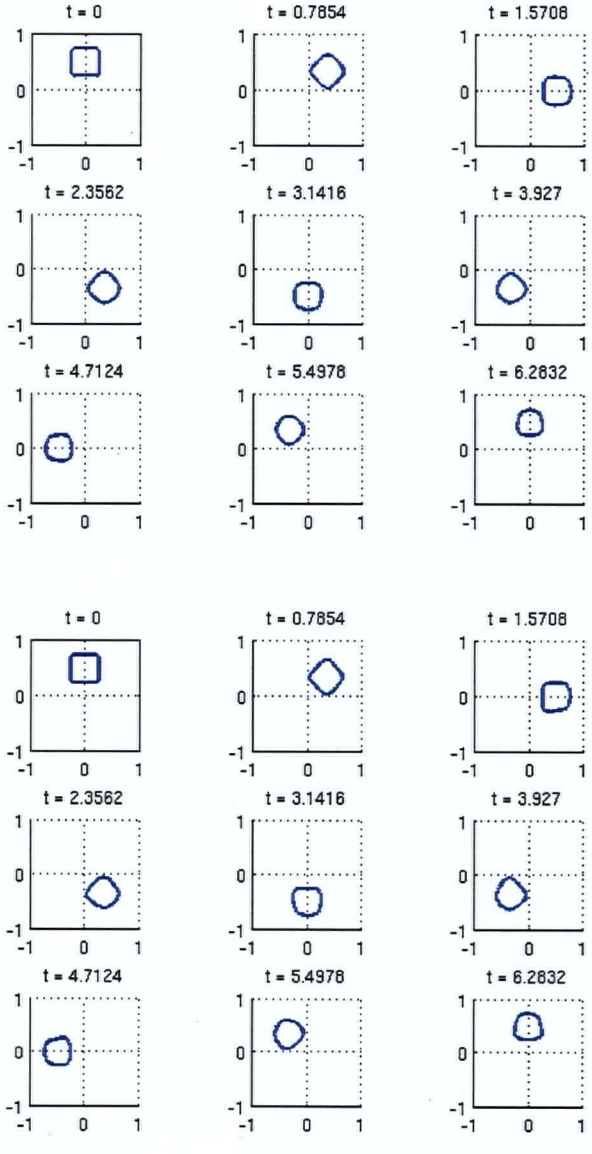

Figure 3.4: Plot of the reachable set on the manifold in x_2 versus x_3 dimensions for the three dimensional toy example. The top image doesn't use the closest point method, while the bottom image does use the method.

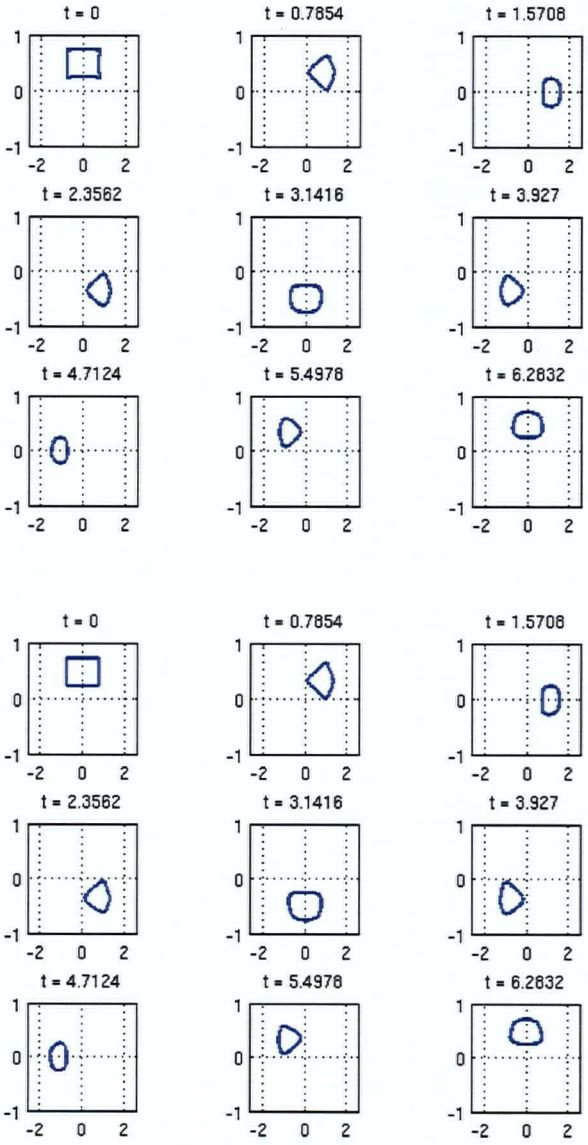

Figure 3.5: Plot of the reachable set on the manifold in s, the horizontal distance along the manifold from the origin in the x_2 dimension, versus x_3 dimensions for three dimensional toy example. The top image doesn't use the closest point method, while the bottom image does use the method.

Chapter 4

Predicting Voltage Instability of a Power System

4.1 Background

Voltage instability is an important factor to consider when constructing power systems. In the case of a load variation or an event disturbance the system should be able to maintain an acceptable level of voltage. Failure to maintain this level can result in a brown out, which can damage electrical equipment. A numerical method for predicting voltage instability was introduced in [13], and this method models the voltage dynamics by a nonlinear hybrid automata. The automaton combines continuous voltage dynamics with discrete operations of the power system. The research in this thesis focuses on numerical methods for solving the continuous component of the voltage dynamics, which can be modeled with a DAE.

The following DAE which models these continuous voltage dynamics was introduced in [17]

$$\begin{aligned} T'_{d0}\dot{E}' &= -\frac{x_1 + x'_d}{x'}E' + \frac{x_d - x'_d}{x'}\frac{E^2 + x'Q(E)}{E'} + E_{fd}, \\ T\dot{E}_{fd} &= -(E_{fd} - E^0_{fd}) - K\left\{E_G(E) - E_r\right\}, \\ 0 &= E'^2E^2 - (x'P)^2 - \left\{x'Q(E) + E^2\right\}^2 \\ &= g(E', E; x_1), \end{aligned} \tag{4.1}$$

where

$$\begin{aligned} E_G(E) &= \frac{1}{E}\sqrt{(x_1P)^2 + \{x_1Q(E) + E^2\}^2}, \\ x' &= x_1 + x'_d, \\ P &= P_m, \\ Q(E) &= Q_0 + HE + BE^2. \end{aligned} \tag{4.2}$$

The physical meaning and values of the variables and parameters are described in Table 4.1.

Several surfaces are defined for the DAE of (4.1). The first is the constraint surface

$$L = \left\{ (E', E_{fd}, E) \in \Re^3 \,|g(E', E) = 0 \right\},$$

which consists of a two-dimensional manifold in a three-dimensional space. The second surface

$$S = \left\{ (E', E_{fd}, E) \in \Re^3 \,\middle|\, \frac{\partial g}{\partial E}(E', E) = 0 \right\}$$

is a singular surface in which the model breaks down because the solution does not hold uniqueness properties. The constraint from (4.2) is differentiated in order to derive the following velocity field

$$\begin{aligned} T'_{d0}\dot{E}' &= -\frac{x_1 + x'_d}{x'}E' + \frac{x_d - x'_d}{x'}\frac{E^2 + x'Q(E)}{E'} + E_{fd}, \\ T\dot{E}_{fd} &= -(E_{fd} - E^0_{fd}) - K\left\{E_G(E) - E_r\right\}, \\ \dot{E} &= \frac{E'\dot{E}'E}{2(x'Q(E) + E^2) - E'^2} \end{aligned} \tag{4.3}$$

The denominator for $\dot{E}$ provides the equation for $\frac{\partial g}{\partial E} = 2(x'Q(E) + E^2) - E'^2$, and the singular surface occurs where $2(x'Q(E) + E^2) - E'^2 = 0$. In mathematical terms this part of the surface needs to be modeled with a DAE with a index higher than one.

4.2 Closest Point Method used as an Extension Velocity

The singular surface S is a challenge: not only is the model inaccurate at S, but as we approach S the ODE (4.3) equivalent to the DAE (4.1) becomes ill conditioned (the term for $\dot{E}$ blows up) so something must be done to the ODE in order to make it behave in an acceptable manner. Some of the undesirable states of the power system are defined by the unsafe set

$$G = \{(E', E_{fd}, E) | E \leq E_c\}$$

The states in G are physically unacceptable for operations because the load bus voltages in those states are too low. Thus, these unsafe states can be used to define the target set, $\mathcal{T}_0 = G$, which is shown in Figure 4.1. The dynamics in the target set can be damped out because they are irrelevant to computing the reach set outside the target set. The following equations are used to damp out the dynamics

Description	Name	Value
generator voltage behind transient reactance	E'	
field excitation	E_{fd}	
load bus voltage	E	
generator bus voltage	E_G	
open-circuit transient time constant	T'_{d0}	$5s$
transmission reactance (two routes)	x_1	0.1
d-axis synchronous reactance	x_d	1.2
d-axis transient reactance	x'_d	0.2
time constant of first-order model of AVR	T	$1.5s$
nominal field excitation	E^0_{fd}	1.6
gain constant of first-order model of AVR	K	7
set-point value of generator bus voltage	E_r	1
mechanical input power to generator	P_m	1.0
constant reactive power of load	Q_0	$0.5P_m$
current source of load	H	0
impedance load	B	0
critical value of load bus voltage	E_c	0.7

Table 4.1: Physical Meaning of Variables and Parameters in Nonlinear Hybrid Automaton [13]

$$H(\phi_0) = \begin{cases} 0 & \phi_0 < -2\epsilon \\ \frac{1}{2} + \frac{(\phi_0+\epsilon)}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi(\phi_0+\epsilon)}{\epsilon}\right) & -2\epsilon \le \phi_0 \le 0 \\ 1 & \epsilon < 0 \end{cases}$$

$$F(x) = H(\phi_0)F(x)$$

where $\epsilon = 2h$ and h represents the grid spacing [7]. This damping function is equal to 1 everywhere outside the target set, so outside the target set the function being damped stays the same. Inside the target set the damping function smoothly and quickly damps out $F(x)$ to essentially zero. The function goes to zero within four grid cells. The implementation in this research uses $\phi_0(x)$ not $CP(\phi_0(x))$ as input to the damping function. The initial $\phi_0(x)$ function is smoother, thus after damping this function it produces better results in the reach tube calculations.

Unfortunately, not all of the singular surface is inside the target set, so another technique needs to be used to make the dynamics outside the target set behave in an acceptable manner. The extension velocity method, described in [1] and in Section 2.1.3, extends a speed function, which moves the interface in its normal direction, off an interface. This method maintains the signed distance property for the level sets representing the interface. Several attempts were made to adapt this well known extension velocity method to the case

of extending velocities off manifolds. This proved difficult and inappropriate for several reasons. In order to calculate the extension velocity the velocity function had to be converted to a speed function in the normal direction of the implicit surface representing the reach tube. This conversion had to be done at each timestep because the reach tube is evolving at each timestep. After the conversion into a speed function, the extension velocity method was used to extend the speed function two times. It was firstly used to extend the speed function in the normal direction of the constraint manifold and then the resulting speed function was extended in the direction of the reach set implicit surface. This conversion seemed inappropriate for the case of extending off the constraint manifold since the extension velocity was being converted to a speed function in the normal direction of the reach set and not in the normal direction of the constraint manifold. Another reason the conversion was inappropriate is that even though the velocity function and the manifold do not change over time, a calculation was being done at every timestep to try and extend the velocity function off of the manifold. This calculation at every timestep was an unnecessary amount of overhead.

A second attempt was made at extending the velocity field off of the constraint manifold. This attempt involved using the previously described extension velocity method to extend each component of the initial velocity field separately off of the constraint manifold. This extension only had to be done once before the level set calculation began because the constraint manifold and the initial velocity do not change throughout the level set calculation, and this extension could be done separately on each component of the velocity field because the components are independent of each other. This extension got rid of the singularity problems in the dynamics and maintained the correct dynamics on the manifold. Unfortunately, running these extension velocity functions took a lot of extra time and this extra time seemed wasteful when a closest point function was already being computed to reinitialize the implicit surface function representing the reach tube. Thus a new extension velocity method is proposed in this research which makes use of the closest point function.

This new method for extending velocities off of the constraint manifold assumes that the dynamics are well defined on the manifold but may have problems somewhere else in the domain. The simple solution proposed in this research is to use the closest point operator, described in Section 3.1.2, to extend each component of the velocity function off of the manifold. For example, if the dynamics $F(x)$ are unsuitable for the HJE's (3.2) or (3.3) we replace it by

$$F_{ext}(x) = F(CP(x)) \tag{4.4}$$

The initial velocity function is parallel to the manifold on the manifold, so off of the manifold the extension velocity function will be parallel to the closest part of the manifold. The dynamics are well behaved on the entire manifold because the singular surface, S, only intersects with the manifold inside the target set and the dynamics inside the target set have already been damped out. Thus, extending these well behaved dynamics off of the manifold will

create well behaved dynamics throughout the entire domain.

Two types of figures are generated to show the validity of the method. Figure 4.2 compares trajectory simulations generated in three different ways by Matlab's DAE/ODE solver. The first method for computing the trajectories uses Matlab's DAE solver with the dynamics described in (4.1). The second method for computing the trajectories uses Matlab's ODE solver with the dynamics described in (4.3). The final method also uses Matlab's ODE solver but it uses the extension velocity dynamics described in (4.4). These extension velocity dynamics are only defined on a discrete grid, so values off the grid must be interpolated by `interpn`. In the figure, all three trajectories are practically indistinguishable, and this similarity shows that the extension velocity method does not perturb the dynamics significantly.

Figure 4.3 calculates some sample trajectories using the interpolated extension velocity, and the starting points of these trajectories are shifted off of the manifold. These shifted trajectories remain parallel to the solutions on the manifold and this shows that the extension velocity contains dynamics which are constant in the normal direction of the manifold. This consistency in the normal direction ensures that extension velocity is a natural extension of the initial surface velocity. Thus, the extension velocity technique used in this research removes the singularities in the domain, and creates well behaved dynamics throughout the entire domain.

4.3 Implementation

The implementation of the power generator problem is similar to the implementation of the two toy examples in Chapter 3. The first difference is that the closest point operator is used to extend the velocities off the manifold as described in the previous section. This extension is only done once during initialization. The second difference is that all derivative approximations are third order accurate.

4.4 Examples

The DAE for the Power Generator was described in (4.1) and the ODE which represents the velocity field was described in (4.3).

The initial reachable tube is a hyperplane with a normal of $(1, 0, 0)$ passing through the point $(E_c, 0, 0)$. The zero level set of the initial reach tube is a plane spanning the E' and E_{fd} dimensions, and this plane divides the E dimension at E_c. This zero level set can be seen in Figures 4.1 and 4.4. The problem was computed on a domain of $[+0.5, +1.7] \times [+0.8, +1.8] \times [-0.1, +0.7]$, while the distance between grid nodes is defined as 0.02. The domain values in the E_{fd} dimension were scaled down by a factor of 10 compared to the original problem specifications in [13] and in [17]. The PDE is executed from 0 to 5 time units and reachable tubes are calculated every 0.2 time units. It takes 2806 seconds to

solve the problem using the closest point method when the closest point function has already been precomputed. The problem cannot be calculated without the closest point method.

Figures 4.4, 4.5 and 4.6 demonstrate the reachable tube at every integer time unit of the calculation. These figures show that the closest point reinitialization for the implicit surface function forces the reach tube to remain perpendicular to the manifold throughout the calculation. Figure 4.7 shows several views of the reach tube at the final timestep, $t = 5$. Figure 4.8 compares the interface for the reachable tube at the final timestep with simulations calculated by Matlab's DAE solver using the dynamics described in (4.1). This figure shows that the level set calculation is doing a poor job calculating the reachable tube. The calculation is overestimating the size of the reachable set, so a lot of safe trajectories are being lost. Several ways to improve the quality of the results could be to increase the grid resolution or to increase the accuracy of the calculation. The closest point function could also be slightly inaccurate, so improvements to the closest point method in order to calculate the closest points more accurately could improve the results.

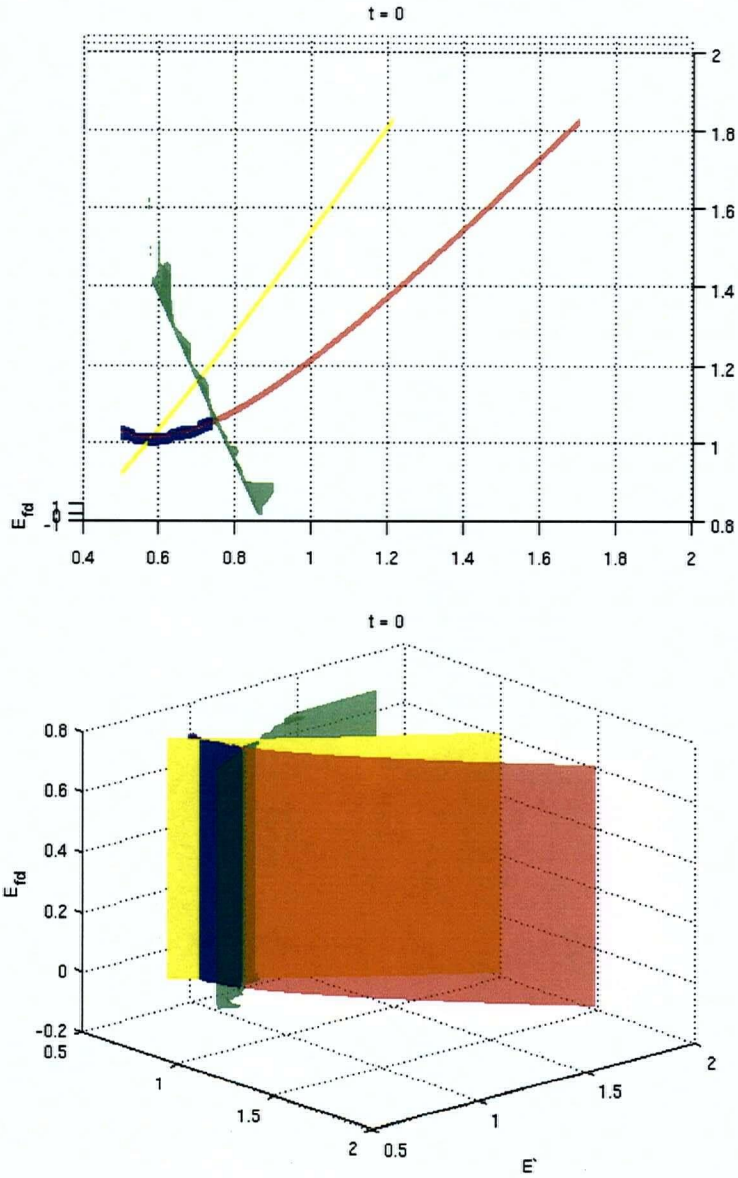

Figure 4.1: Target set and singular set plots for the power generator problem. The top image is a top view of the intersecting surfaces and the bottom image is a side view of the intersecting surfaces. In these images, the red surface is the constraint manifold, the green surface is the interface for the target set, the blue surface represents the target set and the yellow surface represents the singular set.

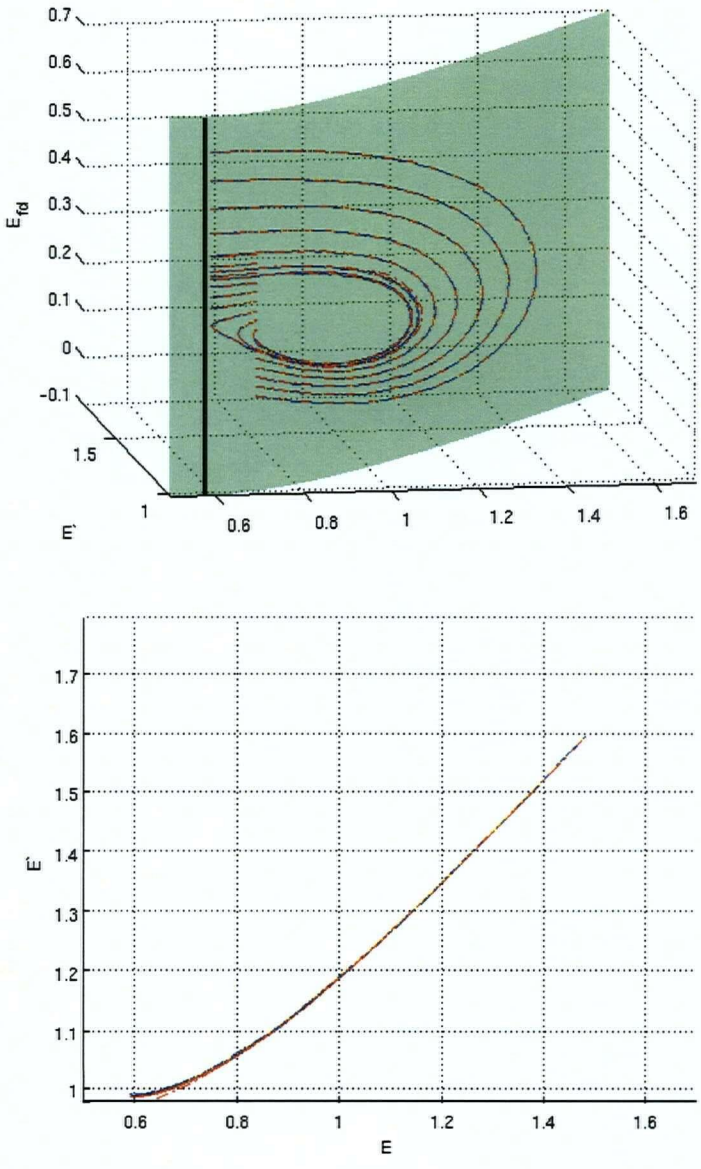

Figure 4.2: Trajectory simulations generated in three different ways by Matlab's DAE/ODE solver. The blue trajectories are calculated by Matlab's ODE solver using the dynamics described in (4.3), the red trajectories are calculated by Matlab's DAE solver using the dynamics described in (4.1) and the yellow trajectories were calculated by Matlab's ODE solver using the extension velocity dynamics. The side view (top) demonstrates that the three trajectories are very similar when looking at them head on, but the top view (bottom) demonstrates that the dynamics force the trajectories slightly off of the constraint manifold in the bottom left hand corner of the plot.

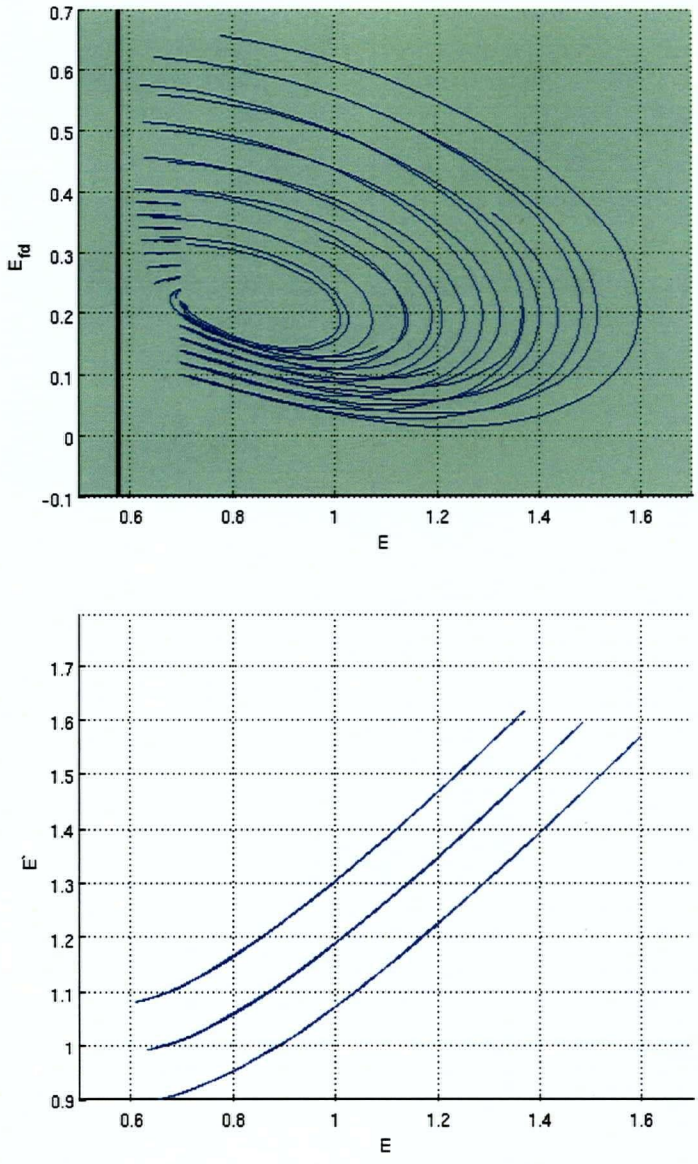

Figure 4.3: Sample trajectories using the interpolated extension velocity. The top image is a side view of the trajectories on the manifold and the bottom image is a top view of the trajectories. The middle set of trajectories is on the manifold and the outer two sets of trajectories are shifted off of the manifold by 0.1 units in the E' dimension. The trajectories remain parallel to the manifold after the shift off of the manifold.

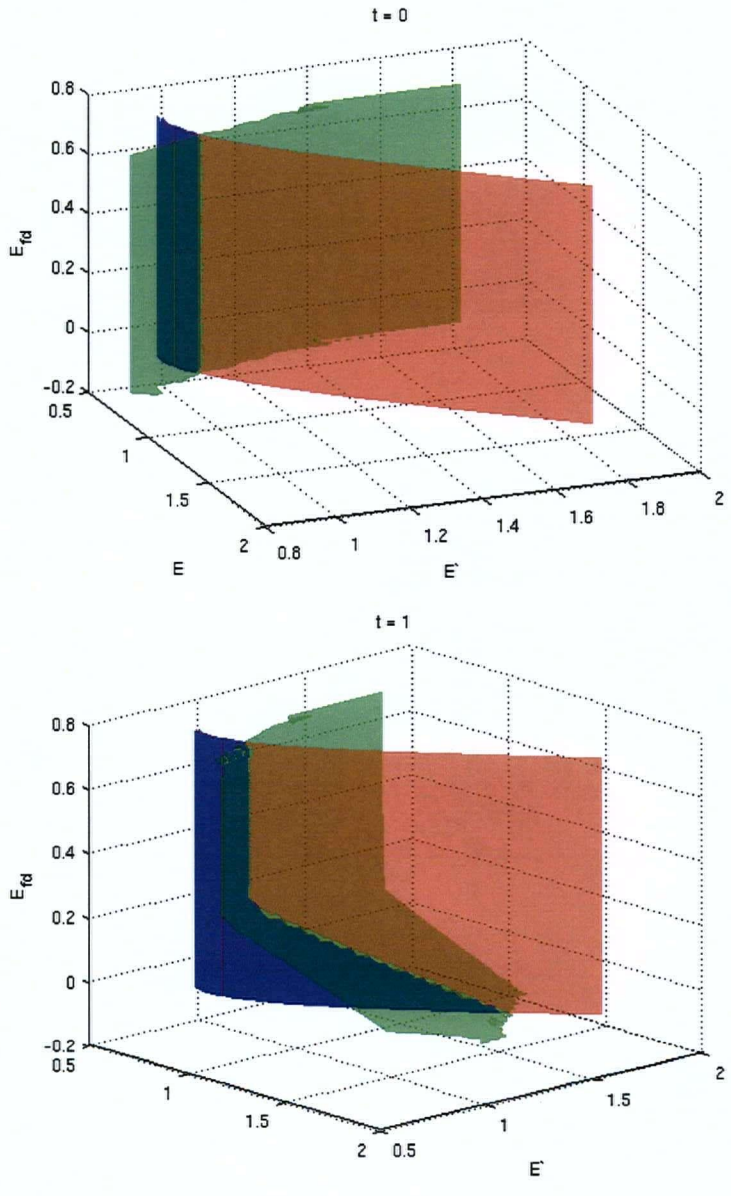

Figure 4.4: View of reach tube at $t = 0$ (top) and $t = 1$ (bottom). In this image, the red surface (darker surface) is the manifold, the green surface (lighter curve) is the interface for the reachable tube and the blue surface represents the reachable tube.

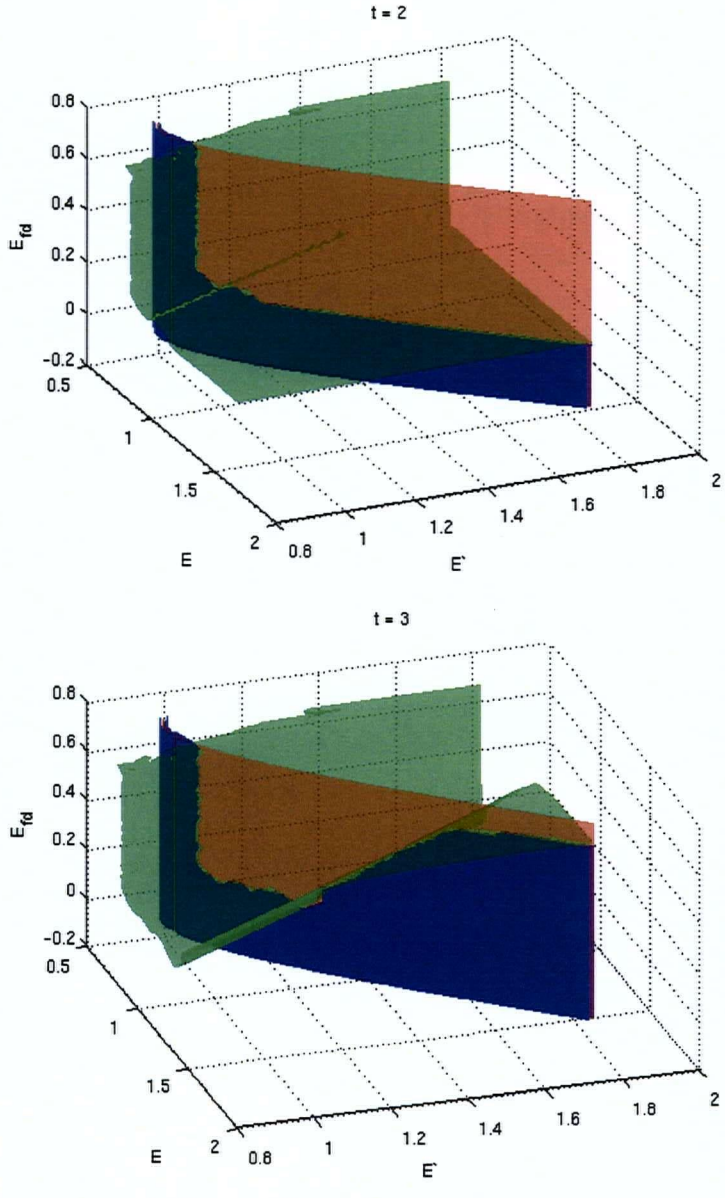

Figure 4.5: View of reach tube at $t = 2$ (top) and $t = 3$ (bottom). In this image, the red surface (darker surface) is the manifold, the green surface (lighter curve) is the interface for the reachable tube and the blue surface represents the reachable tube.

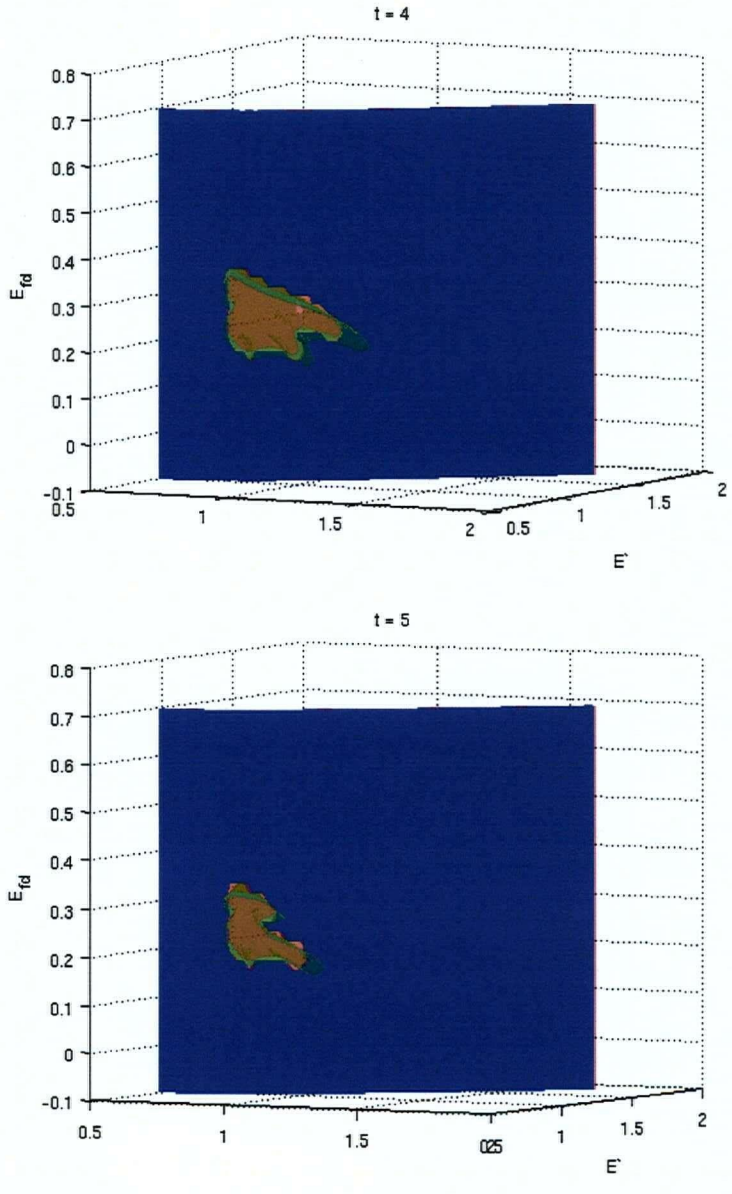

Figure 4.6: View of reach tube at $t = 4$ (top) and $t = 5$ (bottom). In this image, the red surface (darker surface) is the manifold, the green surface (lighter curve) is the interface for the reachable tube and the blue surface represents the reachable tube.

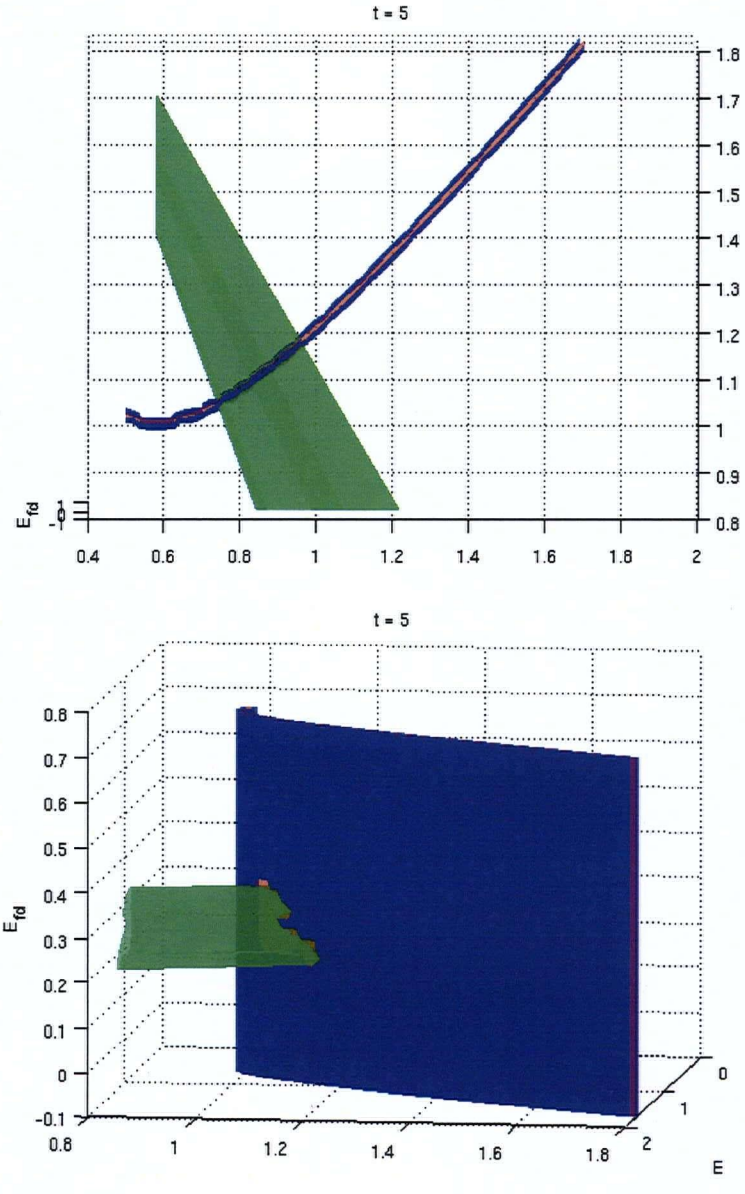

Figure 4.7: Several views of the reach tube at the final timestep. In this image, the red surface (darker surface) is the manifold, the green surface (lighter curve) is the interface for the reachable tube and the blue surface represents the reachable tube.

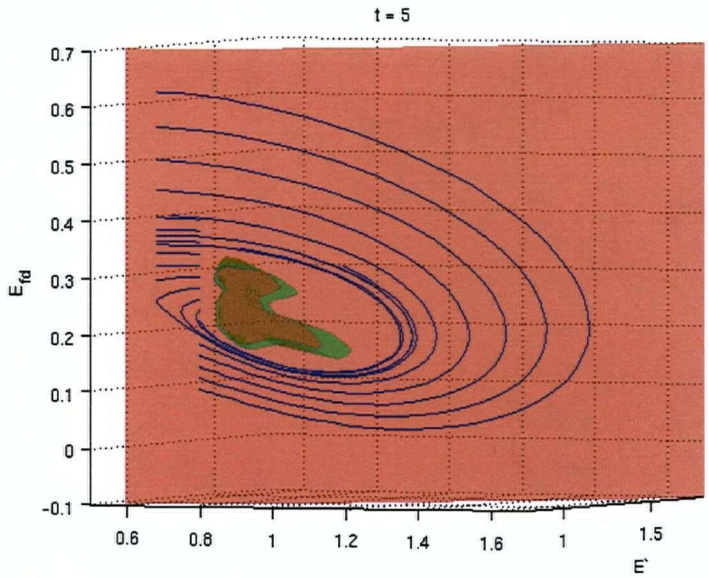

Figure 4.8: The plot compares the interface for the reachable tube at the final timestep with simulations calculated by Matlab's DAE solver using the dynamics described in (4.1). The red surface represents the constraint manifold, the green surface represents the interface for the reachable tube and the blue lines represent the trajectories calculated by the DAE solver.

Chapter 5

Conclusion

This thesis has made two contributions to the intellectual community. Firstly, it has added the fast marching method, signed distance and extension velocity functionality to Ian Mitchell's Level Set Toolbox. These functions use Dijkstra's algorithm to solve the non-linear Eikonal equation. This algorithm traverses each node in the grid once and it relies on a min-heap data structure to traverse these nodes in an optimal $O(N \log N)$ time. The methods were implemented in Matlab in order to work with the Level Set Toolbox, but the min-heap structure's operation time was very slow in this language so the functions had to be reimplemented in C and connected to Matlab via a MEX interface. This greatly improved the running time for all three methods. The implementation of the methods was proved accurate by running some well known example problems.

Secondly, a method was developed, proven and tested for calculating reachable sets on manifolds or equivalently DAEs. The algebraic constraint of the DAE represents the manifold and the differential constraint represents the dynamics of the reachable set. A common practice for solving reachable sets in a full dimensional space is to represent the reachable set surface as an implicit surface function and to use level set methods to evolve the surface. The research in this thesis extends the level set computational technique by reinitializing the implicit surface function at each timestep with the closest point method. This method naturally extends surface gradients off of the constraint manifold onto the entire computational domain, and it does so by assigning each point in the implicit surface function the same value as the value of the closest point on the constraint manifold. This procedure ensures that the gradients of the implicit surface function are constant in the normal direction of the constraint manifold. The closest point method was proven effective by solving two toy DAE problems and comparing the results to those calculated when the closest point method was not used.

This method for solving reach sets on manifolds is used to solve a real life power generator example where the voltage dynamics are modeled with a DAE. This DAE has an additional problem that the model in inaccurate for a subset of the domain. In this subset the actual dynamics are index-2 or higher so the index-1 DAE which is being used to model the dynamics becomes singular at these points. Two fixes are done to the model to deal with the singular surface. The first fix damps out the dynamics inside the target set, since this set of points have already been marked as unsafe. The second fix uses the closest point method to extend the well modeled dynamics on the manifold throughout the entire domain; thus suggesting that the closest point method can be used

to extend both surface functions and velocity functions off of a manifold onto the entire domain in a manner which yields motion on the manifold consistent with the original dynamics.

Bibliography

[1] D. Adalsteinsson and J.A. Sethian. The fast construction of extension velocities in level set methods. *Journal of Computational Physics*, 148:2–22, 1999.

[2] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* Society for Industrial and Applied Mathematics, 1998.

[3] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[4] Ron Kimmel and James A. Sethian. Optimal algorithm for shape from shading and path planning. *Journal of Mathematical Imaging and Vision*, 14(3):224–241, 1992.

[5] Ian M. Mitchell. Comparing forward and backward reachability as tools for safety analysis. In Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo, editors, *Hybrid Systems: Computation and Control*, number 4416, pages 428–443. Springer Verlag, 2007.

[6] Ian M. Mitchell. A toolbox of level set methods version 1.1. 2007.

[7] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces.* Springer-Verlag, 2003.

[8] J. Qian, Y.-T. Zhang, and H.-K. Zhao. Fast sweeping methods for eikonal equations on triangular meshes. *SIAM Journal on Numerical Analysis*, 45:83–107, 2007.

[9] Elisabeth Rouy and Agnes Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal on Numerical Analysis*, 29(3):867–884, June 1992.

[10] Steve J. Ruuth and Barry Merriman. A simple embedding method for solving partial differential equations on surfaces. 2006. `http://www.math.sfu.ca/~sruuth/`.

[11] J.A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences, USA*, 93(4):1591–1595, 1996.

[12] J.A. Sethian. Fast marching methods. *SIAM Review*, 41(2):199–235, July 1999.

[13] Yoshihiko Susuki and Takashi Hikihara. Predicting voltage instability of power system via hybrid system reachability analysis. In *Proceedings of the 2007 American Control Conference*, pages 4166–4171, New York City, USA, July 2007.

[14] Claire J. Tomlin, Ian Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91:986–1001, July 2003.

[15] Yen Hsi Richard Tsai. Rapid and accurate computation of the distance function using grids. *Journal of Computational Physics*, 178:175–195, 2002.

[16] John N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, September 1995.

[17] V. Venkatasubramanian, H. Schättler, and J. Zaborszky. Voltage dynamics: Study of a generator with voltage control transmission, and matched mw load. *IEEE Transactions on Automatic Control*, 37(11):1717–1733, November 1992.