Under review: SIAM Journal of Scientific Computing

A Fast Iterative Method for Eikonal Equations

Won-Ki Jeong and Ross Whitaker

March 23, 2007

Abstract

In this paper we propose a novel computational technique to solve the Eikonal equation. The proposed method manages the list of active nodes and iteratively updates the solutions on those nodes until they converge. Nodes are added to or removed from the list based on a convergence measure, but the management of this list does not entail the the extra burden of expensive ordered data structures or special updating sequences. The proposed method has suboptimal worst-case performance, but in practice, on real and synthetic datasets, performs fewer computations per node than guaranteed-optimal alternatives. Furthermore, the proposed method uses only local, synchronous updates and therefore has better cache coherency, is simple to implement, and scales efficiently on parallel architectures, such as cluster systems or graphics processing units. This paper describes the method, proves its consistency, and gives a performance analysis that compares the proposed method against the state-of-the-art Eikonal solvers.

1 Introduction

The applications of solutions to the Eikonal equation are numerous. The equation arises in the fields of computer vision, image processing, geoscience, and medical imaging and analysis. For example in computer vision, the shape-from-shading problem, which infers 3D surface shape from the intensity values in 2D image, can be modeled and solved with an Eikonal equation [1, 9]. Extracting the medial axis or skeleton of the shape can be done by analyzing solutions of the Eikonal equation with the boundaries specified at the shape contour [14]. Solutions to the Eikonal equation have been proposed for noise removal, feature detection and segmentation [6, 12]. In physics, the Eikonal equation arises in models of wavefront propagation. For instance, the calculation of the travel times of the optimal trajectories of seismic waves is a critical process for seismic tomography [8, 13].

The Eikonal equation, a special case of nonlinear Hamilton-Jacobi partial differential equations (PDEs), is given by

$$H(\mathbf{x}, \nabla \mathbf{x}) = |\phi(\mathbf{x})|^2 - \frac{1}{f^2(\mathbf{x})} = 0, \ \ \forall \mathbf{x} \in \Omega \tag{1}$$

where Ω is a domain in R^n, $\phi(\mathbf{x})$ is the travel time or distance from the source, and $f(\mathbf{x})$ is a positive speed function defined on Ω. For discretization, the Godunov upwind difference scheme is widely used [9, 11, 19]. For example, the first order Godunov upwind discretization of Eq 1 on a 2D uniform grid is given by

$$g(U, \mathbf{x} = (i,j)) = [(U_{i,j} - U_{i,j}^{x\min})^+]^2 + [(U_{i,j} - U_{i,j}^{y\min})^+]^2 = \frac{1}{f_{i,j}^2} \tag{2}$$

where $U_{i,j}$ is the discrete approximation to ϕ at $\mathbf{x} = (i,j)$, and $U_{i,j}^{x\min} = \min(U_{i-1,j}, U_{i+1,j})$, $U_{i,j}^{y\min} = \min(U_{i,j-1}, U_{i,j+1})$, and $(x)^+ = \max(x, 0)$. A 3D discretization also can be defined in a similar fashion. Most solvers for the Eikonal equation rely on the following observation: *once the* upwind *neighborhood values of a grid point are properly determined, Eq 2 can be easily solved using standard solutions to quadratic equations.* The *upwind* neighbors of a grid point are those neighbors whose solutions have values less than or equal to the grid point in question.

A number of different numerical strategies have been proposed to efficiently solve the Eikonal equation, e.g. iterative schemes [9], expanding box schemes [2, 3, 5, 15, 18], expanding wavefront schemes [4, 7, 10, 11, 17], and the sweeping schemes [16, 19]. The iterative approaches rely on fixed-point methods that solve a quadratic equation at each grid point with a Jacobi update and repeat this process until it converges. A drawback of this strategy is that it may require many iterations to converge and the complexity (worst case) behaves as $O(N^2)$ where N is the number of points on the grid. The expanding box schemes start from a source point and explore the neighborhood of points contained in an expanding, square-shape wavefront. A single pass is efficient $O(N)$, but it cannot solve instances with large speed contrasts with only a single update, and therefore additional correction steps (iterations) may be required.

The *expanding wavefront schemes* do not restrict the advance of the wavefront to any special shape. Instead, the wavefronts expand *in the order of the causality* given by the equation and the speed function. An early work of Qin et. al. [7] introduces this approach but they use a brute-force type sorting method that impairs the performance. The *fast marching method*

(FMM) [17, 10, 11], one of the most popular expanding wavefront methods, uses a heap data structure to keep the updating order, and the complexity is $O(N \log N)$, which is worst-case optimal (as in all Dijkstra-type problems). Zhao [19] recently proposed the *fast sweeping method* (FSM) which employs a Gauss-Seidel type update with a Godunov upwind discretization, and has computational complexity $O(kN)$, where k depends on the complexity of the speed function.

The most efficient Eikonal equation solvers are based on adaptive updating schemes and data structures. For instance, FMM uses a heap data structure, which includes nodes from the entire wavefront. During each iteration, the heap determines the as yet unsolved grid value that is guaranteed to depend only on neighbors whose values are solved. The heap must be updated with each updated grid value ($O(\log N)$) and each grid point is solved only once. For this algorithm, the heap becomes a bottleneck that does not allow for massively parallel solutions, such as those available with *single instruction multiple datastream* (SIMD) architectures. Furthermore, grid points must be updated one at a time, in a way that does not guarantee locality, limiting opportunities for coherency in cache or local memory.

FSM uses a Gauss-Seidel update that requires reading from and writing to the single memory location that stores the grid value. This requirement is inefficient or prohibited on some of the most efficient parallel architectures (e.g., SIMD). Furthermore, as we shall see in the results of this paper, the data-dependent performance parameter k in FSM can be quite large for the complex input data. Thus, there remains a need for fast Eikonal solvers that can run efficiently on *both* single processors and SIMD architectures for complex data sets.

The proposed method relies on a modification of an iterative solution of the upwind discretization of the equation. Unlike the traditional iterative schemes, the proposed method selectively updates the points chosen by a convergence measure and avoids using expensive ordered data structures or special updating orders. The method is simple to implement and faster than the existing methods on a wide range of speed functions in two and three dimensions. In addition, the proposed algorithm can be easily ported to parallel architectures, including SIMD processors, cluster systems, or graphics processing units.

In this paper we focus on the development of the algorithm and the evaluation on CPUs in order to make direct comparisons against other methods. This paper presents a series of systematic comparisons of existing Eikonal solvers in the literature. These empirical comparisons shed light on the theoretical worst-case claims of the earlier work. While the worse-case performance of the proposed algorithm is not optimal, it performs much better than worst case on a variety of complex data sets. While several papers argue [4, 19] that the complexity of proposed algorithms *is as low as* $O(N)$ (i.e. best case), there have not been systematic studies to understand how often such methods achieve these best cases or what the constants are that affect the complexity. The actual performance of these algorithms is affected by many different factors, including the size of the datasets or the geometric configurations of the speed functions. One of the contributions of this paper is a careful empirical analysis of the performance of the existing Eikonal solvers, and comparisons against the proposed method in order to better understand the benefits and limitations of each method.

The remainder of this paper proceeds as follows. In the next section we introduce several most popular existing Eikonal equation solvers and discuss benefits and limitations of those methods. In Section 0.3 we introduce the proposed *fast iterative method* (FIM) in some detail. In Section 0.4 we show numerical results on a number of different examples and compare with the existing methods. In section 0.5 we summarize the paper and discuss the future research directions related to this work.

2 Previous Work

In this section we discuss three recent proposed solvers for the Eikonal equations in some detail. These solvers represent several different strategies which are indicative of state-of-the-art solvers. This discussion will give the relevant background for the proposed method and provide a basis for understanding the empirical comparisons among these algorithms.

2.1 Fast Marching Method

Tsitsiklis [17], and later Sethian [10, 11] proposed a worst-case optimal method for solving the Eikonal equation. The so-called *fast marching method* uses a heap to sort points on the moving wavefront. This is based on the property of the solution that guarantees the characteristics are a steepest descent on ϕ. The solution ϕ of the Eiknoal equation therefore can be constructed by expanding a wavefront from a source to a downwind direction using an upwind discretization. The solution at each point depends only on points with smaller values, and updating the minimum value on the wavefront using the heap maintains this condition.

The algorithm uses three different regions, *Alive*, *Close*, and *Far* [11]. *Alive* is a set of points whose values are already found and fixed, and the boundary of the set is called *wavefront*. *Close* is the set of points that are adjacent to *Alive*. *Far* consists of those points that are not yet explored, i.e., they are note *Alive* or *Close*. The algorithm starts by putting the boundary conditions into the set *Alive*, and computing upwind solutions for all the adjacent points and adding them to *Close*. The tentative upwind solutions for the set *Close* are used in a min-heap datastructure on that set. At each iteration, the top of the heap p is removed and the value of that node is set to the tentative solution (which must be correct by the causality condition). The point p is moved from *Close* to *Alive*, and the values of its 1-neighbors (on the grid) are updated and added to *Close*. This find-and-update procedure is repeated until *Close* is empty, which indicates that all points have been explored and are *Alive*.

The performance of the algorithm depends on the size of the input data and the data structure. Inserting/deleting an element in a heap requires $O(\log h)$, where h is the size of the set *Close* (size of the wavefront). This is done for each point on the grid, which gives $O(N \log h)$, and because $h < N$ we have $O(N \log N)$ as the asymptotic worst case, which is optimal. As we might expect, the algorithm performs consistently, and variations in performance depend only on the complexity of the wavefront, which changes the average values of h. In practice, however, the $\log h$ cost of maintaining the heap can be quite significant, especially for large grids or when solving for wavefronts on 3D or higher dimensions.

2.2 Group Marching Method

FMM updates only a single point at a time because whenever a point is updated the heap also must be updated to maintain the correct order. Consider the worst case, in which the value of every point on the current wavefront (*Alive*) depends on the update of a single point. In this case, the update of the point forms a computational bottleneck, and FMM, which finds the point and maintains the heap in $O(\log N)$ time, is optimal. Kim [4] observes that this worst-case scenario is rare and in practice one can safely update multiple points on a given wavefront. Thus, he proposes the *group marching method* (GMM) based on the following theorem:

Theorem 2.1. *Let L be the close set, and define the set G as a subset of L*

$$G = \{\mathbf{x} \in L : \phi(\mathbf{x}) \leq \phi_{L,min} + \frac{1}{\sqrt{2} f_{L,min}}\} \tag{3}$$

where $\phi_{L,min} = \min\{\phi(\mathbf{x}) : \mathbf{x} \in L\}$ and $f_{L,min} = \min\{f(\mathbf{x}) : \mathbf{x} \in L\}$. Then the solutions for the set G can be found simultaneously at a time.

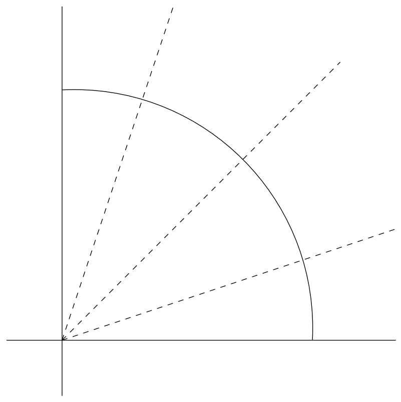
GMM avoids using a heap but maintains a list of points using a simple data structure, e.g. a linked list, and updates a group of points in the list that satisfies the condition of theorem 0.2.1 at the same time. The author argues that the computational complexity of the proposed method is, in principle, $O(N)$.

However, the condition in Eq 3 requires a knowledge of the minimum ϕ and f in the close set L to determine the group G in every update step. In general, it may require $O(N)$ to find this value. To resolve this, [4] suggests using a global bound for G which increases incrementally as the wavefront propagates. Thus, for each update step, G is a set of points $\mathbf{x}$ in L that satisfies $\phi(\mathbf{x}) \leq TM$, where the global bound TM is initially set as the minimum ϕ in the initial narrow band and is increased by $\frac{1}{f_{\Omega,max}}$ where $f_{\Omega,max} = \max\{f(\mathbf{x}) : \mathbf{x} \in \Omega\}$, which is computed in a preprocessing step. If the speed function has large contrasts then only a few points (or even no points) can be updated at each iteration and the algorithm may require a large number of iterations, which impairs performance.

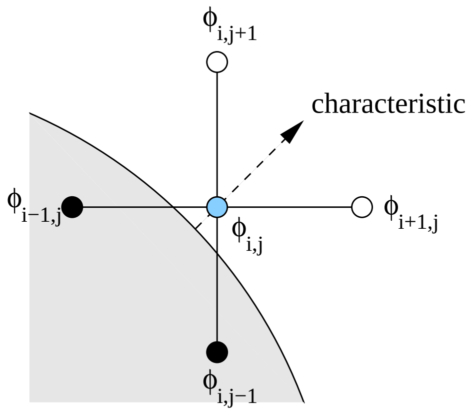
2.3 Fast Sweeping Method

In some cases, the characteristic path, which is the optimal trajectory from one point to another, does not change its direction, as is the case in computing the shortest path on an homogeneous media. If such cases, we can solve the Eikonal equation by updating solutions along a specific direction without explicit checks for causality. Based on this observation, Zhao [19] proposed the *fast sweeping method*, in which the Eikonal equation is solved on an n-dimensional grid using at least 2^n directional sweeps, one per each quadrant, within a Gauss-Seidel update scheme.

Consider a simple 2D example where the input data is homogeneous media (f is constant) and the source is the origin. For the continuous case, the t-level set of the Eikonal equation is a circle with a radius t centered at the origin, and on the discrete grid we have some approximation to this given by the upwind scheme. The characteristic paths for this example are the straight lines passing through the origin (Fig 1a). Hence, every point in the first quadrant depends only on its left and/or down neighborhood points to solve the Eikonal equation (Fig 1b). Thus, sweeping from the origin to the right and up direction can produce exact solutions for the points in the first quadrant. There are two possible sweeping directions for each axis, so n-dimensional data needs at least 2^n sweepings. This method works very well on speed functions for which the characteristic paths do not undergo many changes in direction, however, some applications have inhomogeneous



(a) Characteristic paths (dotted lines)



(b) Neighborhood for $\phi_{i,j}$

Figure 1: 2D example of characteristic paths and neighbors for solving the Eikonal equation in the first quadrant

speeds with complex characteristic paths. In such cases the sweeping method must iterate until it converges, and the performance is significantly reduced.

3 Fast Iterative Method (FIM)

In this section we propose an Eikonal solver based on a selective iterative method, which we call the *Fast Iterative Method* (FIM). The main design goals in order to produce good overall performance, cache coherence, and scalability across multiple processors are:

1. the algorithm should not impose a particular update order
2. the algorithm should not use a separate, heterogeneous data structure for sorting, and
3. the algorithm should be able to simultaneously update multiple points

Criterion 1 is required for cache coherency and streaming architectures. For example, FMM updates solutions based on causality and requires a random access on the memory which impairs cache coherency. Streaming architectures, such as GPUs, are optimized for certain memory access patterns and the special update orders given by the program could force one to violate those patterns. Control of the update order allows an algorithm to update whatever data is in cache (e.g., local on the grid). Criterion 2 is required for SIMD and streaming architectures. For example, FMM manages a heap, which is relatively smaller than the main solution grid, to keep track of causality in the active list. Operation on a heap is multiple instructions on the small datasets, and this cannot be efficiently implemented on SIMD or streaming architecture that is more favorable to process large datasets with a common operation. Criterion 3 allows an algorithm to fully utilize the parallel architecture of cluster systems or SIMD/multi-core processors to increase throughput.

3.1 Algorithm description

FIM is developed based on observations from the two well-known numerical Eikonal solvers. One approach is the iterative method given by Rouy et al. [9], which updates the solution for every point iteratively until it converges. This method does not rely on the causality principle and is simple to implement. However, the method is inefficient because every grid point must be visited (and a quadratic evaluated) until the solutions on the entire grid have converged. The other related approach is FMM [10, 11], which uses the idea of a narrow band of points on the wavefront, and thereby updates points selectively (one at a time) by managing a heap.

The main idea of FIM is to solve the Eikonal equation selectively on the grid points without maintaining expensive data structures. FIM maintains a narrow band, called the *active list*, for storing the grid points that are being updated. Instead of using a special data structure to keep track of exact causal relationships, we maintain a looser relationship and update all points in the active list simultaneously. During each iteration, we expand the list of active points, and the band thickens or expands to include all points that could be influenced by the current updates. A point can be removed from the active list when the solution is converged.

To update the solutions of the points in the active list, we use the Godunov upwind discretization of the Eikonal equation (Eq 2) and solve the equation using a quadratic solver. The pseudo code for computing the solution of 3D Eikonal equation, which takes as input values a, b, c of upwind neighbors in the cardinal directions of the grid and f as a speed, is as follows:

Algorithm 3.1: SolveQuadratic(a, b, c, f)

```
comment: Returns the value $u = U_{\mathbf{x}}$ that solves $g(U, \mathbf{x}) = 0$, where $a \leq b \leq c$
$u \leftarrow c + 1/f$
if $u \leq b$ return $(u)$
$u \leftarrow (b + c + \text{sqrt}(-b^2 - c^2 + 2bc + 2/f^2))/2$
if $u \leq a$ return $(u)$
$u \leftarrow (2(a + b + c) + \text{sqrt}(4(a + b + c)^2 - 12(a^2 + b^2 + c^2 - 1/f^2)))/6$
return $(u)$
```

The main algorithm consists of two parts, the initialization and the updating. In the initialization step, we set the boundary conditions on the grid, and set the values of the rest of the grid points to infinity (or some very large value). Next, the neighbors of the source points in the cardinal directions (4 in 2D and 6 in 3D) are added to the active list L. In the updating step, for every point $\mathbf{x} \in L$ we compute the new $U_{\mathbf{x}}$ by solving Eq 2 and check if the point is converged by comparing the previous and new $U_{\mathbf{x}}$ values. If so, we remove the point from L, and add any 1-neighbor points to L that are not in L and have not converged yet. Note that newly added points must be updated in the next updating iteration, so these points should be added before the deleted point in L. In our implementation, we use a doubly linked list for L, and new points are added right before the deleted point. The updating step is repeated until L is empty. The following pseudo code explains the proposed algorithm.

Algorithm 3.2: UPDATE(**X**)

```
comment: 1. Initialization ($\mathbf{X}$ : set of grid points, $L$ : active list)
for each $\mathbf{x} \in \mathbf{X}$
  do if $\mathbf{x}$ is source
       then $U_{\mathbf{x}} \leftarrow 0$
       else $U_{\mathbf{x}} \leftarrow \infty$
for each $\mathbf{x} \in \mathbf{X}$
  do if any neighbor of $\mathbf{x}$ is source
       then add $\mathbf{x}$ to $L$

comment: 2. Update points in $L$
while $L$ is not empty
  do for each $\mathbf{x} \in L$
       do $p \leftarrow U_{\mathbf{x}}$
          $q \leftarrow g(U_{\mathbf{x}})$
          $U_{\mathbf{x}} \leftarrow q$
          if $|p - q| < \epsilon$
            then for each 1-neighbor $\mathbf{x}_{nb}$ of $\mathbf{x}$
                   do if $\mathbf{x}_{nb}$ is not in $L$
                        then $p \leftarrow U_{\mathbf{x}_{nb}}$
                             $q \leftarrow g(U_{\mathbf{x}_{nb}})$
                             if $p > q$
                               then $U_{\mathbf{x}_{nb}} \leftarrow q$
                                    add $\mathbf{x}_{nb}$ to $L$
                 remove $\mathbf{x}$ from $L$
```

3.2 Properties of the algorithm

In this section we describe how the algorithm works in detail. Figure 2 shows the schematic 2D example of FIM frontwave expanding in the first quadrant. The lower-left corner point is the source point, the black points are fixed points, the diagonal rectangle containing blue points is the active list, and the black arrow represents the narrow band's advancing direction. Figure 2 (a) is the initial stage, (b) is after the first update step, and (c) is after the second update step. Because blue points depend only on the neighboring black points, all of the blue points in the active list can be updated at the same time. If the characteristic path does not change its direction to the other quadrant, then all the updated blue points will be fixed (become black points) and their 1-neighbor white points will form a new narrow band.

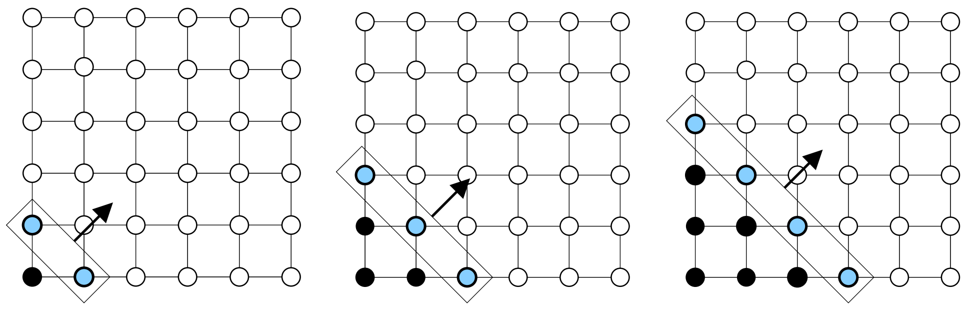

(a) Initial stage (b) After first update (c) After second update

Figure 2: Schematic 2D example of FIM frontwave propagation.

FIM is an iterative method, meaning that a point is updated until its solution converges. However, for many data sets most points require only a single update to converge. This can be interpreted as follows. If the angle between the direction of the characteristic path and the the narrow band's advancing direction is smaller than 45 degree, then the exact solution at the point can be found only in a single update, as in the fast sweeping method. If the angle is larger than 45 degrees, the point at the location where the characteristic path changes the direction will have an initial value that is computed using the wrong up-wind neighborhood, and it will be revised in successive iterations as neighbors refine their values. Thus, that point will not be removed from the active list and will be updated until the correct value is computed. Figure 3 shows this situation. Unlike FMM, where the wavefront propagates with closed, 1-point-thick curves, the FIM can result in thicker bands that split in places where the characteristic path changes the direction (Fig 3 (a) red point). Also, the wavefront can move over solutions that have already converged, and reactivate them to correct values as new information is propagated across the image. Thus, the worst-case performance of FIM is suboptimal. The following section gives the results of empirical studies, including situations where this worst-case behavior undermines computational efficiency of FIM and compares the results with those of the other state-of-the-art solvers.

To prove correctness of the algorithm, we follow reasoning similar to that described in [9].

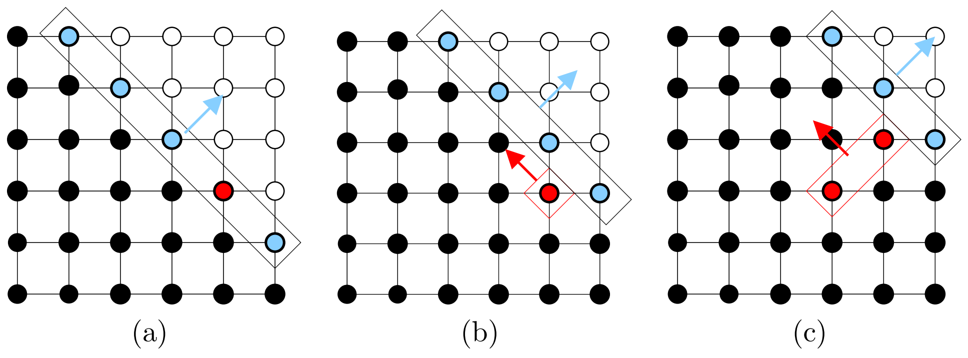


Figure 3: Schematic 2D example of the change of the characteristic direction.

Lemma 3.1. *For strictly positive speed functions, the FIM algorithm appends every grid point to the active list at least once.*

Proof. For every non-source point, any path in the domain from that point to the boundary conditions has cost $< \infty$. As shown in the initialization step in pseudo code 0.3.2, all non-source points are initialized as ∞. Hence, the active list grows outward from the boundary condition in one-connected rings until it passes over the entire domain. $\square$

Lemma 3.2. *FIM algorithm converges.*

Proof. For this we rely on monotonicity (decreasing) of the solution and boundedness (positive). From the pseudo code 0.3.2 we see that a point is added to the active list and its tentative solution is updated only when the new solution is smaller than the previous one. All updates are positive by construction. $\square$

Lemma 3.3. *The solution U at the completion of FIM algorithm with $\epsilon = 0$ (error threshold) is consistent with the corresponding Hamiltonian given in Equation 1.*

Proof. Each point in the domain is appended to the active list at least once. Each point $\mathbf{x}$ is finally removed from $\mathcal{L}$ only when $g(U, \mathbf{x}) = 0$ and the upwind neighbors (which impact this calculation) are also inactive. Any change in those neighbors causes $\mathbf{x}$ to be re-appended to the active list. Thus, when the active list is empty (the condition for completion), $g(U, \mathbf{x}) = 0$ for the entire domain. $\square$

Theorem 3.4. *FIM algorithm, for $\epsilon = 0$ gives an approximate solution to Equation 1 on the discrete grid.*

Proof. The proof of the theorem is given by the convergence and consistency of the solution, as given lemmas above. $\square$

4 Results and Discussion

Several algorithms from the literature report worst or best case performances, but to understand the actual performance of algorithms in realistic settings we have conducted a systematic empirical comparison. We have implemented and tested the various algorithms on an Windows PC, equipped with a Pentium 3.6 gigahertz CPU and 2 gigabytes of main memory. For this paper, we focus only on single processor performance, with the understanding that this performance and intrinsic scalability will decide the parallel performance of the algorithms.

We have constructed five different synthetic speed examples for input to the Eikonal solvers. These speed functions capture various situations that are important to the relative performance of the different solvers including constant speeds, large speed contrasts, and large changes in characteristic directions. The boundary conditions consist of a single point with a distance of zero, unless stated otherwise, and the domain is normalized to the size $\Omega = [0,1]^3$. The tables given below measure the CPU times in seconds. The relative difference in solutions that constiutes convergence for both FSM and FIM is 10^{-6}. The speed examples we use for the experiment are as follows:

Example 1 $f = 1$, constant speed with a single source

Example 2 $f = \begin{cases} 3 & \text{for } \mathbf{x} \in [\frac{1}{3}, \frac{1}{3}] \times [\frac{1}{3}, \frac{1}{3}] \times [\frac{1}{3}, \frac{1}{3}] \\ 0.001 & \text{otherwise} \end{cases}$

Example 3 Correlated random speed between 50 to 250

Example 4 $f = 1$, constant speed with maze-like barriers

Example 5 $f = \begin{cases} 1 & \text{if } x \in [0, \frac{1}{5}] \\ 10 & \text{if } x \in [\frac{1}{5}, \frac{2}{5}] \\ 100 & \text{if } x \in [\frac{2}{5}, \frac{3}{5}] \\ 1000 & \text{if } x \in [\frac{3}{5}, \frac{4}{5}] \\ 10000 & \text{if } x \in [\frac{4}{5}, \frac{5}{5}] \end{cases}$

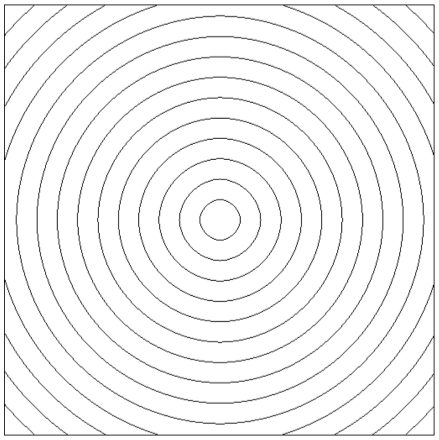
In the following section, we compare the running time of Eikonal solvers on the different speed examples. In section 0.4.2, we give a detailed performance analysis using average number of computations per point, especially focusing on FMM and FIM.

4.1 Running time comparison

Example 1 is the simplest case, computing a distance field over a homogeneous media. Fig 4 (a) shows the level sets of the solution. The speed function is constant, so the t-level set of ϕ is the circle/sphere with the radius t from the source. Table 1 compares the CPU time of 2D and 3D Eikonal solvers for speed example 1. FSM converged only after five sweeps in 2D and nine sweeps in 3D due to the simplicity of the example. GMM, the FSM, and FIM run about two to four times faster than FMM on this example.

	2D				3D			
	256^2	512^2	1024^2	2048^2	32^3	64^3	128^3	256^3
FMM	0.141	0.563	2.516	11.547	0.094	0.922	10.812	129
GMM	0.062	0.312	1.328	6.079	0.062	0.469	4.469	39
FSM	0.063	0.266	1.484	5.968	0.062	0.5	5.532	44
FIM	0.078	0.313	1.282	5.516	0.047	0.406	3.578	30

Table 1: Running time on speed example 1


(a) Example 1

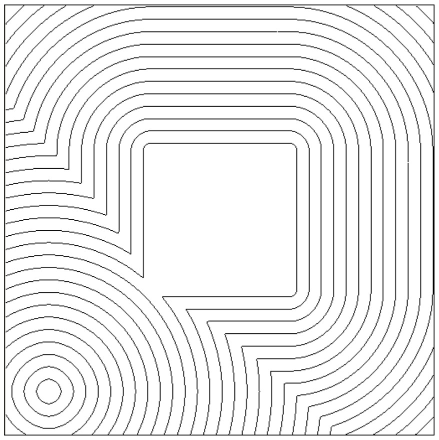

(b) Example 2

Figure 4: Level set of the solution of speed function examples 1 and 2

Example 2 is a binary speed image. The speed of the inner square is 3 and the outer region is near zero. This example models a domain that consists of two different materials. This could represent, for instance, water and rock, which are often found in simulations of seismic datasets. Figure 4

(b) shows the level sets of the solution, and Table 2 shows the running time of the Eikonal solvers on this example. In 2D, FSM converges after five sweeps, the fastest of those we compared, while FIM and FMM both require a similar (slower) time. However, in 3D, FIM is the fastest. The difference between 2D and 3D results can be explained by the increased area of the wavefront, which penalizes FMM, and the increased number of sweep directions, which drives the compute time of FSM (20 sweeps were required for this example). GMM is the slowest both in 2D and 3D for this speed function because the performance of the method highly depends on the input speed contrasts. GMM updates a group of points selected by a gradually increasing global bound, and the amount of increment is decided by the reciprocal of the largest speed value. Hence, if the speed contrast is large and the current wavefront is located on the slow region, then the method takes many iterations to move the current wavefront on that region.

	2D				3D			
	256^2	512^2	1024^2	2048^2	32^3	64^3	128^3	256^3
FMM	0.141	0.641	3.813	31.82	0.093	0.937	11.20	165
GMM	14.79	67.67	304	1336	5.64	56.04	954	12916
FSM	0.063	0.266	1.484	5.937	0.093	0.922	11.57	109
FIM	0.141	0.672	4.032	31.61	0.062	0.531	6.609	50

Table 2: Running time on speed example 2

Example 3 is generated by blurring and rescaling the random noise image to create correlated random speed values. This example resembles the kind of speed functions that are observed in real seismic data. Fig 5 (a) shows the speed example (black : 50, white : 250), and (b) is the level set of the solution on that example. Table 3 compares the running time of the solvers on 2D (one slice) and 3D (entire volume) examples. FIM and GMM perform far better than FMM and FSM on this example. Especially, GMM runs about three times faster than FMM because the example does not have large speed contrasts. For 3D, FSM requires 35 sweeps to converge on this example.

Example 4 is designed to simulate a dataset with varying characteristic directions. Several barriers with open holes at the ends are placed to force the characteristic paths to turn several times. The solutions on the barriers are not computed, making no characteristic path can pass through the barriers. Figure 6(a) shows the barriers and the level sets of the solution. Example 4 contains numerous changes in the directions of the characteris-

	2D (256^2)	3D (256^3)
FMM	0.141	174
GMM	0.078	54
FSM	0.172	188
FIM	0.141	108

Table 3: Running time on speed example 3

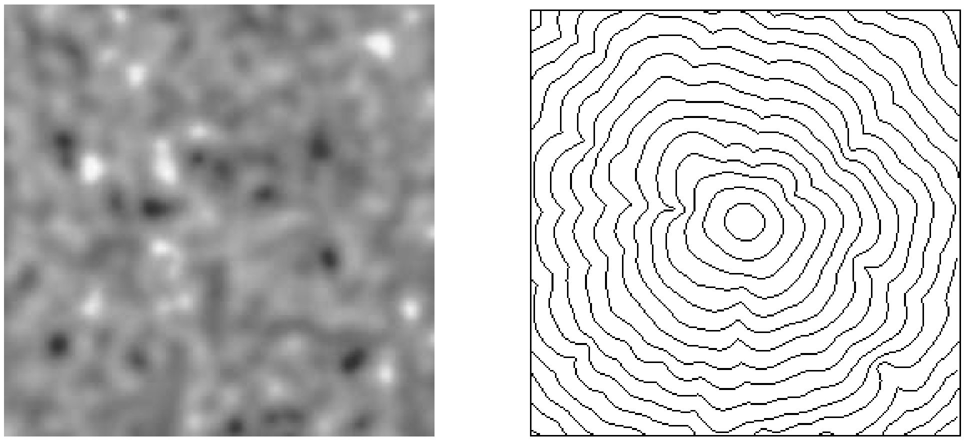

(a) Example 3 speed map (b) Example 3 solution

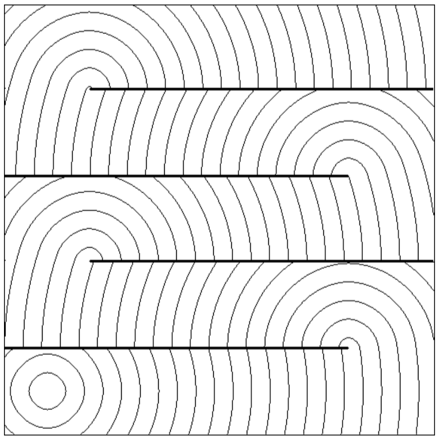
Figure 5: Example 3 speed map and the level sets of the solution

tics, and thus FSM takes up to 10 sweeps in 2D and 22 sweeps in 3D to converge. Meanwhile, FIM and GMM work nicely on this example because the length of the narrow band is short, allowing the updating of the list to take lesser time than the previous examples. Table 4 shows that FIM operates the fastest, about four times faster than FMM or FSM in 3D. GMM is slightly slower than FIM but also runs very efficiently on this example.

Example 5 is designed so that the Euclidean distance for the points in the upper-left corner region is longer than the actual distance based on the speed function for those. Such a configuration will impair the performance of FIM because multiple updates are required for each point. Figure 6 (b) shows the color map of the solution on speed example 5 (red:max dist, blue:min dist). As expected, FIM did not work well on this example, but interestingly GMM also performs poorly. This is because the example has very large speed contrasts (minimum 1 to maximum 10000). FSM is the fastest on this example because of the relatively simply geometries.

	2D				3D			
	256^2	512^2	1024^2	2048^2	32^3	64^3	128^3	256^3
FMM	0.109	0.469	2.078	9.141	0.078	0.812	9.656	128
GMM	0.062	0.281	1.203	4.985	0.046	0.422	4.015	39
FSM	0.125	0.516	2.937	11.79	0.14	1.188	13.57	111
FIM	0.078	0.297	1.172	4.703	0.046	0.359	3.156	29

Table 4: Running time on speed example 4


(a) Example 4

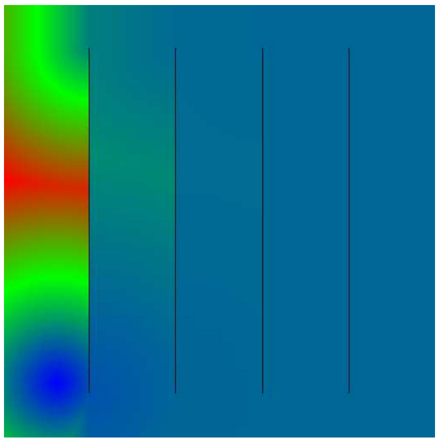

(b) Example 5

Figure 6: Level sets of the solution of speed example 4 and 5

Fig 7 compares the increasing rate of the running time on different data sizes (in 3D) on speed example 1. The graphs represent the scaled CPU time to show how much the running time increases on larger datasets compared to that of the smallest data size. FIM and the show the slowest growing CPU times, about half of that of FMM. The CPU time for FSM grows slightly faster than FIM or GMM but much slower than FMM.

4.2 Performance analysis

In the previous section, performance of the Eikonal solvers are compared based on the actual running time of the code. Because running time can be affected by many factors, e.g. implementation or cache performance, we measure the number of computations for a more careful performance analysis. The most time consuming operation in the Eikonal solver is solving

	2D				3D			
	256^2	512^2	1024^2	2048^2	32^3	64^3	128^3	256^3
FMM	0.125	0.532	2.328	10.7	0.078	0.922	13.87	290
GMM	9.937	51.18	265.25	1217	3.687	35.79	376.68	6493
FSM	0.11	0.453	2.656	9.5	0.109	1.203	16.28	154
FIM	0.125	0.516	2.235	9.5	0.125	1.328	16.61	233

Table 5: Running time on speed example 5

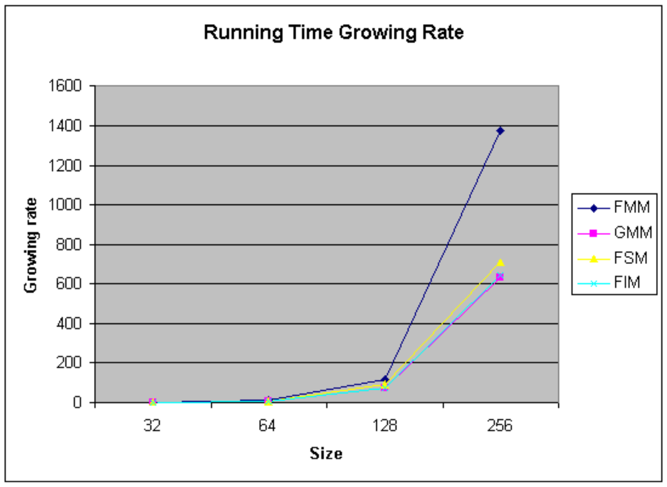


Figure 7: Comparison of CPU time increasing rate

the quadratic equations. Table 6 compares the average number of the solved quadratic equations per point on different input 3D speed examples.

FMM is a one-pass algorithm, meaning that a point is fixed with only a single update, and thus the average number of computations is same for all the speed examples. However, the tentative solutions on the wavefront must be computed multiple times. Each point has six neighbors in a 3D grid and we can assume that about half of neighbors are fixed when a point is updated. Thus, each point is updated roughly three times on average. GMM is also a one-pass algorithm, but each point is updated at most twice to ensure stability, so the average number of computations is bound by about twice that of FMM, which is six. The average number of computations is same as the number of sweeps for FSM, and it highly depends on speed examples. FIM also depends on the speed examples, but due to its local operations, the average number of computations is not as big as FSM. For

	Example 1	Example 2	example 3	Example 4	Example 5
FMM	2.98	2.98	2.98	2.97	2.97
GMM	5.83	2.14	3.75	5.364	2.75
FSM	9	21	35	22	30
FIM	4.98	6.9	9.97	4.97	23.05

Table 6: Average number of solving quadratic equation per point on 256^3

most speed examples, the FIM needs on average less than ten operations per point. On a special configuration like example 5, FIM requires a large number of operations.

Even though FMM's average number of operations is much smaller than that of FIM, the actual running time is much longer than FIM in all 3D speed examples shown above. The reason is that managing a heap in FMM is very expensive, especially for 3D. Let k_1 and k_2 as the cost for a quadratic solver computation and a heap updating operation respectively, and P_{FMM} and P_{FIM} are the average number of operations per point in FMM and FIM respectively (as in table 6). Let h be the average heap size. The total cost for FMM and FIM on the grid size N can be defined asymptotically as follows.

$$\begin{aligned} C_{\text{FMM}} &= N(k_1 P_{\text{FMM}} + k_2 P_{\text{FMM}} log_2(h)) \\ &= N k_1 P_{\text{FMM}}(1 + \frac{k_2}{k_1} log_2(h)) \\ C_{\text{FIM}} &= N k_1 P_{\text{FIM}} \end{aligned}$$

Hence, for FIM to outperform FMM, the following condition must hold.

$$N k_1 P_{\text{FMM}}(1 + \frac{k_2}{k_1} log_2(h)) > N k_1 P_{\text{FIM}}$$

$$P_{\text{FIM}} < P_{\text{FMM}}(1 + \frac{k_2}{k_1} log_2(h))$$

For example, empirically measured $\frac{k_2}{k_1} log_2(h)$ for speed example 1 on 256^3 grid is around 2.5, and P_{FMM} is around 3 (Table 6). Hence, if P_{FIM} is less than 10 then FIM would outperform FMM. The ratio $\frac{k_2}{k_1} log_2(h)$ depends on several factors, i.e. size of the heap or value of elements to be updated. For example 5, $\frac{k_2}{k_1} log_2(h)$ goes up to around 8, so FIM will run faster than FMM if P_{FIM} is smaller than 27.

Table 7 summarizes the performance of the discussed Eikonal solvers over several data categories. '+' implies that the method works well on the category, and '$-$' vice versa. The proposed method, FIM, runs faster than the other methods on most cases, and GMM also works well on most cases except the data with extremely large speed contrasts. FSM performs well only on the dataset with a few changes of characteristic direction, making FSM less suitable for solving general Eikonal equations. The performance of FMM is not significantly affected by factors other than the size of the dataset, but the method is already very expensive for 3D datasets compared to GMM or FIM.

	FMM	**GMM**	**FSM**	**FIM**
Large Speed Contrasts	$+$	$-$	$+$	$+$
Large Data Size	$-$	$+$	$+$	$+$
Varying Speed Values	$-$	$+$	$-$	$-$
Char. Dir. Changes	$+$	$+$	$-$	$+$
Parallelization	$-$	$+$	$-$	$+$

Table 7: Overall performance comparison on data categories

5 Conclusion and Future Work

In this paper we propose a novel Eikonal solver based on the selective iterative method. The method employs the narrow band approach to keep track of the points to be updated, and iteratively updates the solutions until they converge. Instead of using an expensive sorting data structure to keep the causality, the proposed method uses a simple list to store active points and updates all of them at the same time until they converge. The points in the list can be removed from or added to the list based on the convergence measure. The method is simple to implement and runs faster than the existing solvers on general Eikonal equations. The method is also easily portable to parallel architectures, e.g. graphics processors or cluster systems, which is difficult or not available on the existing methods.

The development of the algorithm is initially motivated by the effort of implementing an efficient Eikonal solver on the graphics hardware, so the GPU implementation of the proposed algorithm is a topic for an upcoming paper by the authors. Developing a fast iterative method for anisotropic Hamilton-Jacobi PDEs would also be another useful extension of this work.

References

[1] A. Bruss. The eikonal equation: some results applicable to computer vision. *Journal of Math. Phy.*, 23(5):890–896, 1982.

[2] J. Dellinger. Anisotropic finite-difference traveltimes. In *SEG Technical Program Expanded Abstracts*, pages 1530–1533, 1991.

[3] W. Schneider Jr., K. Ranzinger, A. Balch, and C. Kruse. A dynamic programming approach to first arrival traveltime computation in media with arbitrarily distributed velocities. *Geophysics*, 57(1):39–50, 1992.

[4] S. Kim. An O(N) level set method for eikonal equation. *SIAM Journal on Scientific Computing*, 22(6):2178–2193, 2001.

[5] S. Kim and R. Cook. 3-D traveltime computation using second-order ENO scheme. *Geophysics*, 64(6):1867–1876, 1999.

[6] R. Malladi and J. Sethian. A unified approach to noise removal, image enhancement, and shape recovery. *IEEE Trans. on Image Processing*, 5(11):1554–1568, 1996.

[7] F. Qin, Y. Luo, K. Olsen, W. Cai, and G. Schuster. Finite-difference solution of the eikonal equation along expanding wavefronts. *Geophysics*, 57(3):478–487, 1992.

[8] N. Rawlinson and M. Sambridge. The fast marching method: an effective tool for tomographics imaging and tracking multiple phases in complex layered media. *Exploration Geophysics*, 36:341–350, 2005.

[9] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. *SIAM Journal of Numerical Analysis*, 29:867–884, 1992.

[10] J. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Natl. Acad. Sci.*, volume 93, pages 1591–1595, February 1996.

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

[12] J. Sethian. *Level set methods and fast marching methods.* Cambridge University Press, 2002.

[13] R. Sheriff and L. Geldart. *Exploration Seismology.* Cambridge University Press, 1995.

[14] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. Zucker. The hamilton-jacobi skeleton. In *Proc. International Conference on Computer Vision*, pages 828–834, September 1999.

[15] J. Trier and W. Symes. Upwind finite-difference calculation of traveltimes. *Geophysics*, 56(6):812–821, June 1991.

[16] Y. Tsai. Rapid and accurate computation of the distance function using grids. *J. Comput. Phys.*, 178(1):175–195, 2002.

[17] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. on Automatic Control*, 40(9):1528–1538, September 1995.

[18] J. Vidale. Finite-difference calculation of traveltimes in three dimensions. *Geophysics*, 55(5):521–526, 1990.

[19] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation*, 74:603–627, 2004.