SIAM J. SCI. COMPUT.
Vol. 32, No. 5, pp. 2853–2874

SOME IMPROVEMENTS FOR THE FAST SWEEPING METHOD*

STANLEY BAK†, JOYCE MCLAUGHLIN‡, AND DANIEL RENZI§

Abstract. In this paper, we outline two improvements to the fast sweeping method to improve the speed of the method in general and more specifically in cases where the speed is changing rapidly. The conventional wisdom is that fast sweeping works best when the speed changes slowly, and fast marching is the algorithm of choice when the speed changes rapidly. The goal here is to achieve run times for the fast sweeping method that are at least as fast, or faster, than competitive methods, e.g. fast marching, in the case where the speed is changing rapidly. The first improvement, which we call the locking method, dynamically keeps track of grid points that have either already had the solution successfully calculated at that grid point or for which the solution cannot be successfully calculated during the current iteration. These locked points can quickly be skipped over during the fast sweeping iterations, avoiding many time-consuming calculations. The second improvement, which we call the two queue method, keeps all of the unlocked points in a data structure so that the locked points no longer need to be visited at all. Unfortunately, it is not possible to insert new points into the data structure while maintaining the fast sweeping ordering without at least occasionally sorting. Instead, we segregate the grid points into those with small predicted solutions and those with large predicted solutions using two queues. We give two ways of performing this segregation. This method is a label correcting (iterative) method like the fast sweeping method, but it tends to operate near the front like the fast marching method. It is reminiscent of the threshold method for finding the shortest path on a network, [F. Glover, D. Klingman, and N. Phillips, *Oper. Res.*, 33 (1985), pp. 65–73]. We demonstrate the numerical efficiency of the improved methods on a number of examples.

Key words. eikonal equation, fast marching, fast sweeping, static Hamilton–Jacobi equation

AMS subject classifications. 65N06, 65N22

DOI. 10.1137/090749645

1. Introduction.

In this paper we consider the isotropic Eikonal equation,

$$|\nabla T| = f(x), \tag{1.1}$$

where f is a positive function. This equation is most often used to model the first arrival times of a wave or moving interface. As a first order partial differential equation, the solution can be obtained by the method of characteristics. However, even if the boundary conditions and the speed function, f, are smooth, the characteristics can cross and the solution, T, becomes multivalued. At the same time, there is a strong connection between static Hamilton–Jacobi equations like the Eikonal equation and hyperbolic conservations laws, especially in one dimension, where the derivative of a solution to the Hamilton–Jacobi equation exactly satisfies a hyperbolic conservation law. Much like the entropy conditions for hyperbolic conservations laws, a unique physically relevant solution, the so-called viscosity solution (see [2]) can be obtained by defining T to be the infinum of all the multivalued solutions. Again, paralleling

*Received by the editors February 12, 2009; accepted for publication (in revised form) May 3, 2010; published electronically September 23, 2010.
http://www.siam.org/journals/sisc/32-5/74964.html

†University of Illinois, Department of Computer Science, Chicago, IL (sbak2@illinois.edu). The first author was partially supported by ONR grants N00014-05-1-0600 and N00014-08-1-0432.

‡Mathematical Sciences Department, Rensselaer Polytechnic Institute, Troy, NY 12180 (mclauj@rpi.edu). The second author was partially supported by ONR grants N00014-05-1-0600 and N00014-08-1-0432.

§5341 Regal Road, Sun Prairie, WI 53590 (danielprenzi@yahoo.com). The third author was partially supported by NSF VIGRE grant DMS 9983646.

hyperbolic conservation laws and monotone numerical schemes have been devised for the numerical computation of (1.1). Level set methods are also described in [15].

There are two general methods for computing the solution to (1.1). The first is to introduce a time-dependent term and iterate until convergence; see [25]. The advantage of this idea is that high-order methods are well developed for time-dependent Hamilton–Jacobi equations; for a review see [21]. However, there is a CFL condition that restricts the time step size, and the iterations are on the order of one of the spatial dimensions.

The second general method is to discretize the static equation (1.1), which results in a nonlinear system. There are two different algorithms to solve this nonlinear system efficiently. The first is the fast marching method [5], [16], [17], [18], [23]. This method takes advantage of causality. For the Eikonal equation, the characteristics are in the direction, ∇T, so information flows from small times to big times. Because of this causality, if all the points, x, in the domain, D, are put in ascending order of arrival times, the solution can be obtained by calculating each point once. Of course ordering the points in this way requires knowledge of the unknown arrival times T. The fast marching method determines the proper order during the calculation using a heap and uses an upwind numerical scheme that preserves the causality of the arrival times T. Because this method requires some sorting, the running time of the method is $O(n \times \log(n))$, where n is the number of points in the domain. More recently two $O(n)$ fast marching methods have been developed [10], [24]. The speedup in the first method is gained by placing the arrival times into bins, each of which contain all the points which have arrival times in a small range. The bins are ordered, but the points in each bin are not, so no sorting is required. This introduces some additional error that is bounded by the bin size. The second method accepts groups of points at a time. The size of the group is chosen so that each point can be calculated in at most two iterations. Fast marching methods have also been extended to triangular domains, [11], [19], and convex anisotropy [19], [20].

The second algorithm, the fast sweeping method, uses Gauss–Seidal iterations with alternating directions to solve the nonlinear system [1], [8], [4], [9], [12], [13], [14], [22], [27]. On rectangular grids, in one dimension the iterations alternate from left to right and in two dimensions the iterations alternate between up-left, up-right, down-left, and down-right. In general there are 2^N directions, where N is the number of dimensions in the problem. During each iteration, all of the characteristics that lie in the direction of the sweep are followed as long as the characteristic direction continues to lie in the direction of the sweep. In this way groups of characteristics are covered simultaneously. While in general the convergence is governed by a contraction property and is quite slow if the discretization scheme preserves the causality the number of iterations required essentially depends on how many times the characteristics change direction; see [27]. This depends strongly on the speed function, f, but for fixed f, is constant; hence the running time of the method is $O(n)$ with a constant that depends on the speed function. Fast sweeping methods have also been extended to triangular domains, [14], and convex anisotropy [9], [13]. One important property of the fast sweeping method is that it is an iterative method, and the value at each grid point does not need to be correct after the first visit. This is advantageous for high-order methods, [25], [26], and more general problems [8].

In this paper we describe some improvements to the fast sweeping method that increase the speed of the algorithm with more significant increases in the cases where the speed is changing rapidly. While the fast sweeping method is already very fast, whenever the sweep direction does not match the characteristic direction or the arrival

time has already been successfully computed in a previous iteration, the calculations made are unnecessary and take up a large percentage of the total running time, especially when the number of iterations is large. To remove most of these unnecessary calculations, we introduce a locking procedure that locks out points whose arrival times are already calculated or cannot be successfully calculated in the current iteration. While the logic to control the locking procedure takes some additional time to compute, we emphasize that it

- is based only on causality and requires no sorting,
- is small compared to the computation time,
- significantly cuts down on the number of calculations,
- converges to the same answer as the original fast sweeping method.

While this locking technique significantly improves the speed of the fast sweeping method, it still visits the exact same number of points as the original method. The improved speed is gained by doing a quick check to determine if an update is needed at each grid point. This means that there is some mild dependence on the speed function. As the number of times the characteristic direction changes is increased, the running time increases linearly just like the original fast sweeping method. However, the constant is based on the time to do a quick check (no calculations) as opposed to the time to do the calculations at one grid point, a significant improvement. This locking method performs particularly well when the number of sweeps is small (less than 10).

To further remove the dependence of the running time on the speed function, we develop a second method that no longer visits every point during every sweep. Instead, all of the points that are not locked out are placed in a queue. Unfortunately, it is not possible to keep the fast sweeping ordering without at least occasionally sorting. We believe it is possible to keep the ordering with only O(1) sorts and will pursue this possibility in a future work. Instead, points are taken out of the queue in an almost increasing order, much like the $O(n)$ fast marching method that uses bins, but the locking procedure described above detects points where the causality of the discretization scheme has been violated and recalculates these points.

The rest of this paper is composed as follows. The discretization scheme is described in section 2. A review of the fast sweeping method is given in section 3. In section 4 we describe the locking and queue procedures. Discussion of when each method is appropriate is given in section 5. Some examples are given in sections 6 and 7 to demonstrate accuracy and efficiency. Finally some concluding remarks are given in section 8.

2. Discretization. Solving for the arrival times relies on using the isotropic Eikonal equation

$$|\nabla T| = f(x).$$

We use the Godonov Hamiltonian to discretize (1.1) because it is an upwind scheme and converges very quickly. At interior grid points in a two-dimensional rectangular mesh, we are solving the equation

$$[(T(x,y) - x_{neb})^+]^2 + [(T(x,y) - y_{neb})^+]^2 = f^2 h^2,$$

where x_{neb} is the smaller arrival time of the two neighbors in the x direction ($x_{neb} = \min(T(x-1,y), T(x+1,y))$), y_{neb} is the smaller arrival time of the two neighbors in the y direction ($y_{neb} = \min(T(x,y-1), T(x,y+1))$), and $x^+ = \max(x, 0)$. Solving

for $T(x,y)$, the time at the current point yields

$$\mathrm{T}(x,y) = \begin{cases} \min(x_{neb}, y_{neb}) + f(x,y)h & |x_{neb} - y_{neb}| \geq f(x,y)h, \\ \frac{x_{neb}+y_{neb}+\sqrt{2f(x,y)^2h^2-(x_{neb}-y_{neb})^2}}{2} & |x_{neb} - y_{neb}| < f(x,y)h. \end{cases} \tag{2.1}$$

Using this equation at every grid point results in a nonlinear system. In the next section we describe how to use the fast sweeping method to solve this nonlinear system.

3. The fast sweeping method. The fast sweeping method, briefly described earlier, is one way of solving for arrival times. Rather than ordering the points in strictly increasing order of arrival times while calculating, as in the fast marching method, the algorithm sweeps across the entire grid using alternating Gauss–Seidel iterations, performing calculations to determine $T(x,y)$ at each point. However, the calculations are only correct at points where the characteristic travels in the sweeping direction and the neighboring points with smaller arrival times have already been successfully calculated. This means that you have to perform sweeps in each direction, and if the direction of the wave changes often, more sweeps are required. On a rectangular grid, in one dimension, there are two sweeping directions (left and right), and in two dimensions, there are four sweeping directions (up-left, up-right, down-left, down-right). In general there are 2^N sweeping directions, where N is the number of dimensions in the problem. This method has a running time of $O(n)$ because the number of times the characteristic changes direction gives an upper bound on the number of iterations required; see [27]. The algorithm is as follows.

Initialization.

1. Set initial arrival time at all points to infinity.
2. Set proper arrival times for any fixed values (sources).

While sweeping.

1. Compute the arrival time at each point using (2.1).
   - If the computed time is greater than or equal to the previous time at the point, do nothing.
   - If the computed time is less than the previous time at the point, update the time at the point.

Stopping condition.

- During a single sweep, no points are updated.

Note that the "While sweeping" loop is actually 2^N loops in N dimensions. On a 2D rectangular domain the four loops would be

1. for i=0:I, for j=0:J,
2. for i=I:0, for j=0:J,
3. for i=I:0, for j=J:0,
4. for i=0:I, for j=J:0.

Figure 1 shows a one-dimensional illustration of the method with $f = 1$ and the initial condition $T(x) = 0$, where x is the center of the computation line. We show the initial state of the T in Figure 1(A). The arrival times are shown again after sweeping from left to right in Figure 1(B). Notice that everywhere to the right of the point source, where the characteristic direction is to the right, the solution is now correct. However, the points more than one grid point to the left of the point source are incorrect, and their calculation was unnecessary. After an additional sweep from the right to the left, the correct answer is obtained (Figure 1(C)). Note that the points to the right of the point source have been recalculated even though, at those points, the solution was already correct.

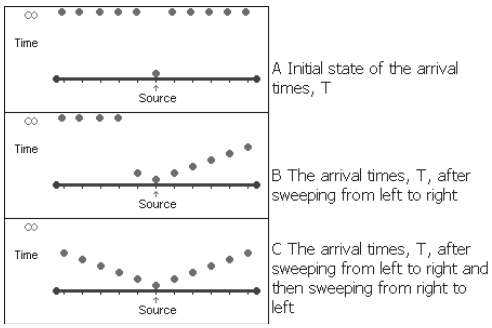


FIG. 1. *The fast sweeping method in one dimension.*

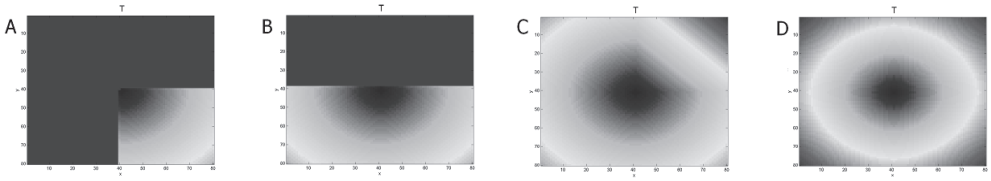


FIG. 2. *The fast sweeping method in two dimensions.* (A) *shows the arrival times after sweeping down-right.* (B) *is an image of the arrival times after a further sweep down-left.* (C) *displays the arrival times, T, after a further sweep up left.* (D) *shows the final state of the arrival times, which is obtained after a further sweep up-right. A fifth sweep is required which does not update any values for the method to reach its termination condition.*

Figure 2 is an illustration of the method in two dimensions with $f = 1$ and the initial condition $T(x) = 0$, where x is the center of the computation area. The first sweep (Figure 2(A)) is down-right, the next one (Figure 2(B)) is down-left, then up-left (Figure 2(C)), and finally up-right (Figure 2(D)). In Figure 2(C), although the entire top half was updated, the arrival times are only accurate in the upper-left quadrant, and points in the upper-right quadrant will get updated to the correct values in the next sweep. Another sweep is required (not shown) that does not change the arrival times but rather determines that the method has finished converging. This means that by the time the method is completed, every point has been calculated five times and only one of these calculations actually updated the point to a correct value. Ideally the method would realize when an update would yield meaningful information and only then perform a time-consuming calculation.

Another drawback of the regular fast sweeping method is that the number of iterations (sets of sweeps in each direction) depends on how many turns the gradient takes. An example to illustrate this would be a speed profile where there is an impassable spiral with the source in the center so that the arrival times propagate outward through the spiral. Because we need to travel through the spiral to get the proper arrival time on the outside, the number of sweeps needed (and therefore the calculation time) depends on the number of times the spiral wraps around. This dependence of the calculation time on the speed profile would ideally be minimized or eliminated. We illustrate this in section 7.4.

4. Fast sweeping method speedups. In this section we introduce several improvements to the fast sweeping method. First, in section 4.1 we introduce a locking procedure that locks grid points that have either already been correctly calculated or cannot be successfully calculated at the current iteration. This avoids time-consuming

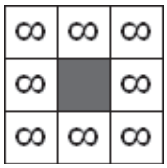


FIG. 3. *Since the surrounding points currently have T values of infinity, an update at the center grid point will not yield useful information.*

calculations. Next, in section 4.2 we store the unlocked grid points in a data structure so that locked points no longer have to be visited and checked to see if useful calculations can be made. Depending on the speed profile, however, this can actually slow down the speed of the computation. We solve this in section 4.3 by using a slightly different data structure. We emphasize that the solution obtained with both methods is exactly the same as that obtained from the original fast sweeping method.

4.1. Locks. As mentioned earlier, one of the drawbacks of the fast sweeping method is that time-consuming calculations are performed at points where an update will not yield any meaningful information (Figure 3). This is especially true during the first few sweeps, where the arrival time at most points will be its initial value of infinity. Also, there is no need to calculate the time for a point whose neighbors have not changed since the last time we calculated it (since we will just recompute the same value). In addition, from the discretization scheme (2.1), it is obvious that when the arrival time at a point is updated, the only points that can be affected by this new arrival time are neighboring points with larger arrival time. In our locking scheme, we will maintain a locking structure at each point; points we should calculate will be unlocked, and points we should skip over without calculating (points where calculations would yield no useful information) will be locked. The method, on a 2D rectangular grid, is as follows.

Initialization.

1. Set initial arrival time at all points to infinity.
2. Set proper arrival times for any fixed values (sources).
3. Lock every point.
4. Unlock the points directly north, south, east, and west of any fixed values (sources).

While sweeping.

1. If a point is locked, proceed to the next one without computing anything.
2. If a point is unlocked, compute the arrival time at it using the Godunov numerical flux.
   - If the computed time is greater than or equal to the previous time at the point, do nothing.
   - If the computed time is less than the previous time at the point, update the time at the point. For each locked point north, south, east, and west of the point, if the time at the neighbor is more than the time at the current point, unlock the neighbor.
3. Lock the point.

Stopping condition.

- During a single sweep, no points are updated (every point is locked).

This method has the advantage of quickly skipping over points that would otherwise be unnecessarily calculated. We measured the improvement on a 640×640

example with a constant f and a single point source at the center of the computation area. The time values obtained largely depend on what computer used to measure them as well as the state of the computer (processes running, CPU load) at the time of the measurement. We used a computer with an Intel Pentium M 1600 MHz processor with 512MB RAM running Windows XP SP2. Measuring the total time it took to do the calculation 10 times obtained the following results:

- Regular sweeping = 2.7 seconds;
- Locking sweeping = 1.8 seconds;
- Only initialization = 0.5 seconds.

The percent of time the locking method takes compared to the regular fast sweeping method is

$$100 \times (1.8 - 0.5)/(2.7 - 0.5) = 59\%.$$

This is close to the best-case example for the fast sweeping method, where only 1.25 sweeps are required. In examples where the fast sweeping method requires more sweeps (see the example in section 7.4), the fast sweeping method performs much worse than the locking sweeping method. Using this method, the number of unnecessary calculations is reduced and thus addressed the first issue mentioned at the end of section 3. However, the second issue is not addressed: this method is still takes the same number of iterations as the traditional fast sweeping method on a given speed profile. Each iteration is faster, but the dependence of the number of sweeps on the speed profile still exists. If many sweeps are required, a lot of time is wasted during the Gauss–Seidal iterations performing the check required to skip locked nodes (in the locking algorithm this is the first step performed while sweeping). While the locked nodes are "quickly" skipped over, for some speed profiles a locked node may be skipped over dozens or, in extreme cases, hundreds of times. In addition, in extreme cases where the speed profile changes very rapidly (see the example in section 7.3), the fast sweeping algorithm performs extremely poorly. While the locking scheme performs better than the fast sweeping method on these examples, it is still not competitive with fast marching methods. We address these issues in the next two sections.

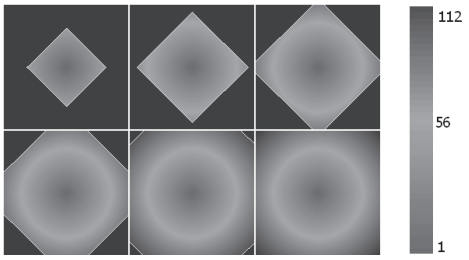
4.2. A queue. In this section we remove the need to visit and quickly skip over locked nodes by storing the unlocked nodes in a data structure. Unfortunately it is not possible to maintain the fast sweeping ordering without at least occasionally sorting, so we drop the Gauss–Seidal sweeps in favor of another iteration method. The simplest way to accomplish this is to use a queue to store the unlocked points over which we iterate. Points to calculate are taken from the front of the queue, while points that are unlocked are put in the back of the queue. On a 2D rectangular grid the single queue algorithm is as follows.

Initialization.

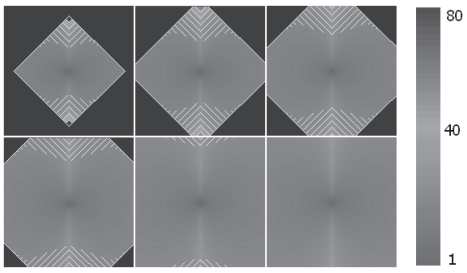
1. Set initial arrival time at all points to infinity.
2. Set proper arrival times for any fixed values (sources).
3. Lock every point.
4. Unlock the points directly north, south, east, and west of any fixed values (sources) and place them in a queue.

While queue is not empty,

1. Take a point from the front of the queue. Compute the arrival time at this point using (2.1).
   - If the computed time is greater than or equal to the previous time at the point, do nothing.



(a) The arrival times for a single point source constant speed example using the single queue method. The bright line near the edges represents the points currently in the queue. The method finishes optimally, calculating each point exactly once.



(b) The arrival times for a poor performing example for the single queue method. The bright lines near the edges represent the current points in the queue. The stripes of points within the queue continuously find faster arrival times for points directly above and below the source (center). These additional recalculations slow down the method significantly. Some of the points in the queue are within the already-computed area, which indicates that times are being recalculated.

FIG. 4. *Two examples using the single queue method.*

- If the computed time is less than the previous time at the point, update the time at the point. For each unlocked point north, south, east, and west of the point, if the time at the neighbor is more than the time at the current point, unlock the neighbor and place it at the back of the queue.

2. Lock the point.

Figure 4(a) shows this method on an example with constant speed and a single source point in the center of a 2D computation area. The bright lines near the edges represent the points currently in the queue at various points in the computation. Since the points in the queue are all on the edge of the computation area, this example shows good performance. Every point is calculated once, which is optimal.

This single queue method performs well when the speed is uniform throughout the region. However, when the speed varies throughout the computation area, the performance declines dramatically. This is due to the necessary recomputation of points whose fastest arrival time occurs by traveling though fast regions, rather than directly from the source point. This is illustrated in Figure 4(b). In this example, there is a single source point at the center of the computation area. There speeds are arranged in vertical stripes such that the further you go from the source in the x direction, the faster you move (Figure 5). The result is that many points are recomputed as better arrival times are found. This is an illustration of poor performance because many points are inserted into the queue over and over (the bright stripes indicate the points in the queue).

4.3. Two queues. In this section we improve upon the one queue method by calculating the arrival times in an "almost increasing" order. This is accomplished by segregating all of the unlocked points into two groups. The first group of unlocked points have a small predicted arrival time and should be calculated soon. The second group of unlocked points have a large predicted arrival time and should not be calculated soon. This is accomplished using two queues and a queue cutoff. Points to be calculated are removed from the first queue, and its locked neighbors with larger arrival times are

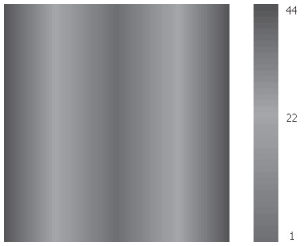


FIG. 5. *The value of f for an example where the single queue method performs poorly. The resulting arrival times are shown in Figure* 4(b). *The speeds are small near the center of the image and remain the same along the y axis. Going left or right from the center, however, results in an increase in speed.*

added to either the first or the second queue, depending on their predicted arrival time (large or small). The simplest way to predict the arrival time of a recently unlocked point is to use the arrival time of the point that just was updated. If this arrival time is smaller (greater) than the queue cutoff, the neighboring point is inserted at the back of the first (second) queue. When the first queue becomes empty, the cutoff is increased and the grid points in the second queue are moved to the first queue. In this way, points with predicted smaller arrival times (points whose neighbors had smaller arrival times) are given priority when calculating over points with predicted larger arrival times. Since we are still only inserting into one of two queues, the insertion operation is still $O(1)$. The two queue algorithm on a 2D rectangular grid is as follows.

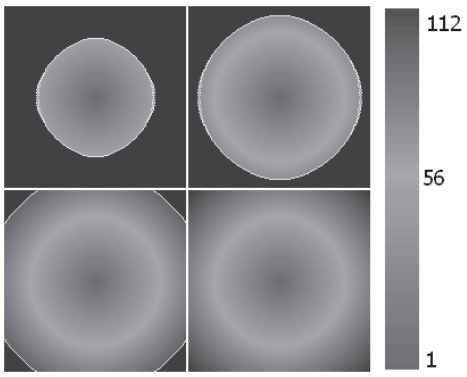
Initialization.

1. Set initial arrival time at all points to infinity.
2. Set proper arrival times for any fixed values (sources).
3. Lock every point.
4. Unlock the points directly north, south, east, and west of any fixed values (sources) and place them in the first queue.

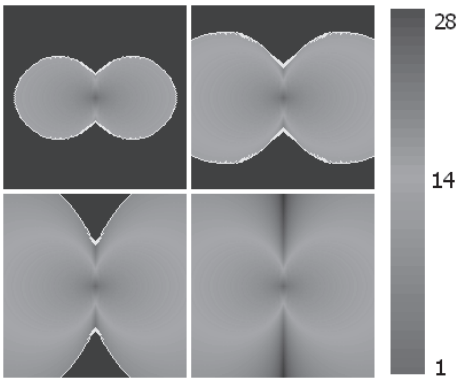
While at least one of the two queues is not empty,

1. While the first queue is not empty,
   - Take a point from the front of the first queue. Compute the arrival time at this point using (2.1).
     - If the computed time is greater than or equal to the previous time at the point, do nothing.
     - If the computed time is less than the previous time at the point, update the time at the point. For each unlocked point north, south, east, and west of the point, if the time at the neighbor is more than the time at the current point, unlock the neighboring point. Place the newly unlocked neighboring points at the back of the first (second) queue if the arrival time at the current point is less (greater) than the queue cutoff.
   - Lock the point.
2. Remove all the points in the second queue and place them into the first queue.
3. Increase the queue threshold (see sections 4.3.1 and 4.3.2 below).

Figure 6(a) is an illustration of the method on a single point source constant speed example. The bright lines near the edges indicate points that are in either of the queues. The solution progresses in a circular shape rather than a diamond shape as in the one queue method (Figure 4(a)). This is similar to how the fast marching method would perform the calculation. Since this method does not compute points



(a) The two queue method calculating a single point source constant speed example. Bright points near the edges are the points inside either of the queues. The two queue segregation method used here is the static cutoff scheme discussed in section 4.3.1.



(b) The two queue method on the increasing speed example of Figure 4(b). Bright points near the edges are the points in either queue. The two queue segregation method used here is the dynamic cutoff scheme discussed in section 4.3.2.

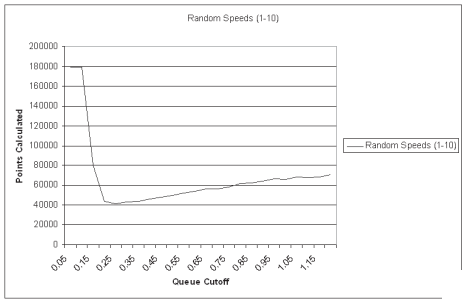
FIG. 6. *The two queue method computing two examples.*

with much larger arrival times, the method performs well on the increasing speed example in Figure 4(b). Notice that in Figure 6(b) there are less points in the queues (bright lines) than in the same example using one queue method. This indicates that fewer points are being recalculated unnecessarily. Also, since there are not points in the queues in the middle of the computation area, computations are not being redone, as was the case in Figure 6(b).

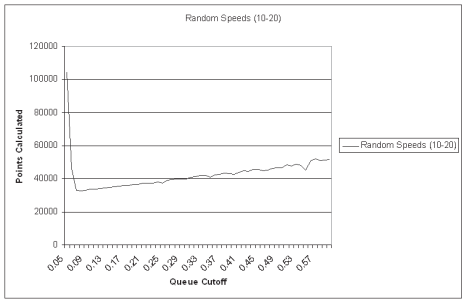
4.3.1. Static queue cutoff. When using the two queue method, you must have a cutoff which is used to distinguish which points should be inserted into the first queue versus which points should be inserted into the second queue. Care must be taken when selecting this cutoff as it will affect the speed of the method, although, as mentioned earlier, the final calculated arrival times will always be the same. A cutoff that is too large will result in all of the points being inserted into the first queue, and the method will perform identically to that of the single queue method. A cutoff that is too small will result in most of the points being inserted into the second queue, and the method will also perform similar to the single queue method. To take advantage of any speedups offered by the two queue method, the cutoff must be set somewhere between these two extremes.

A strategy is also needed when increasing the queue cutoff after the first queue becomes empty. The simplest approach, which we use in the following examples, is to always increase the current cutoff by a constant amount. For example if the cutoff was initially 0.1, points unlocked from any point with an arrival time less (more) than 0.1 would be put into the first (second) queue. After the first queue becomes empty, all of the grid points with an arrival time less than 0.1 are computed correctly. The points in the second queue are moved to the first queue, and the queue cutoff is increased to 0.2. Now points unlocked from any point with an arrival time less (more) than 0.2 are put into the first (second) queue. The next time the first queue becomes empty the queue cutoff would be increased to 0.3, and so on until no more points remain in either queue indicating the calculation is complete.

Since the single queue method performs optimally when the speed is constant, we use a different example to test the effect of changing the queue size. The example we



(a) The effect of the queue cutoff on the number of points calculated in a 160×160 random speed example. Here speeds are random at all points between 1 and 10, inclusive. The regular fast sweeping nethod takes calculates 1,049,600 points on this example (about 25 times as many as the static queue method does with an ideal queue cutoff).



(b) The effect of the queue cutoff on the number of points calculated in a 160×160 random speed example. Here speeds are random at all points between 10 and 20, inclusive. The regular fast sweeping method takes calculates 1,049,600 points on this example.

FIG. 7. *The effect of various static queue cutoffs on examples with random speeds.*

use in Figure 7(a) has a single source point in the center and speeds randomly selected between 1 and 10 at all points. When the queue cutoff is set too low, the method performs identically to the single queue method. With a well-chosen queue cutoff, however, we can obtain the same answer with only about 25% of the calculations. The ideal queue cutoff, for this example, appears to be between 0.2 and 0.3, with a larger cutoff being preferred over one that is too small. It would appear a good choice for a cutoff would be one that is slightly more than $1/AvgSpeed$, where $AvgSpeed = f_{avg} \times h$, where f_{avg} is the average of f over the region of calculation. In this case, $AvgSpeed = 5.5$ and $h = 1$, and $1/AvgSpeed = 0.1818$. A safer cutoff would use the minimum speed rather than the average speed to make sure the cutoff is not too small (which leads to particularly poor performance). A safe cutoff for this example would be $1/MinSpeed = 1/1 = 1.0$. Using the safe cutoff, however, leads to slightly worse performance.

This heuristic is also correct when we change the speeds to be randomly chosen between 10 and 20. In this case, $AvgSpeed = 15$ and $1/AvgSpeed = 0.0667$. Figure 7(b) shows the effect of different queue cutoffs on the number of calculations in this example. The ideal queue cutoff lies between 0.070 and 0.120.

In our experience, a formula for determining a good queue cutoff when the speeds do not vary by more than an order of magnitude throughout the computation area is

$$QueueCutoff = 1.5/AvgSpeed, \tag{4.1}$$

$$AvgSpeed = f_{avg} \times h. \tag{4.2}$$

4.3.2. Dynamic queue cutoff. In examples with speeds that vary significantly, using $1.5/AvgSpeed$ may lead to poor performance. For example, consider a computation area where the speeds on the left half are random between 1 and 10, as in Figure 7(a), and the speeds on the right half are random between 101 and 110, as in Figure 7(b). The calculated ideal queue cutoff is around $1.5/55.5 = 0.027$, while the ideal queue cutoff using this scheme is actually closer to 0.2. Finding the ideal cutoff requires solving the entire problem many times so is therefore not always feasible. One possible strategy requiring further research would be to downsample the prob-

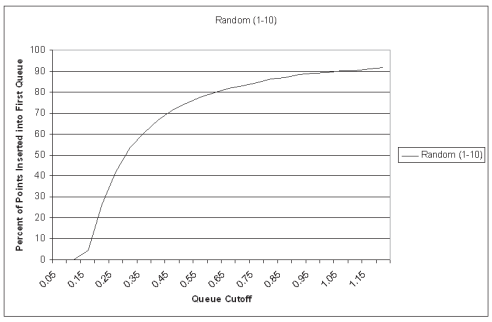


FIG. 8. *The effect of the queue cutoff on the percent of points inserted into the first queue. When the queue cutoff is close to the ideal one for the example (about* 0.3*; see Figure* 7(a)*), the percent of points being inserted into the first queue is slightly under* 50%.

lem, find the ideal cutoff, then solve the larger problem. However, large changes in speeds in different areas may create a situation where there is no single queue cutoff that is best (one may be good while we are calculating in a slower region, another may be better for a faster region, while one in between may perform poorly in both areas).

For these reasons, we introduce a method of dynamically changing the queue cutoff so that performance stays high and we do not need to solve the problem to determine if a specific cutoff is adequate. To determine whether the cutoff needs to be increased or decreased, we track the number of points we are inserting into each queue. For the examples in Figures 7(a) and 7(b), the ideal cutoff occurs when we insert about 50% of the points into the first queue. Figure 8 shows the percent of points inserted into the first queue for different queue cutoff sizes in the random (1–10) example of Figure 7(a). Therefore, if we track this percent while calculating and notice that we are inserting less than this percent into the first queue, we should increase the queue cutoff. If the percent that is going into the first queue is too large, we should decrease the queue cutoff.

One observation is that a cutoff that is too large performs only slightly more calculations than the ideal cutoff, but one that is too small does significantly more calculations. For this reason, the proposed method is slightly cautious in that it tries to maintain the number of points being inserted into the first queue between 65% and 75%, instead of at exactly 50%.

To track the percent, every inserted point increases the count of the number of points we have inserted. If the point happens to go into the first queue, the count of the number of points that we have inserted into the first queue increases. The queue cutoff is adjusted when the first queue becomes empty using the follow approach:

$minPercent := 65$,
$maxPercent := 75$,
$percent := 100 \times \frac{insertedIntoFirstQueue}{insertedTotal}$,
if $(percent < minPercent)$, $cutoffStep := 1.5 \times cutoffStep$;
if $(percent > maxPercent)$, $cutoffStep := \frac{cutoffStep}{2}$;
$queueCutoff := queueCutoff + cutoffStep$,
$insertedIntoFirstQueue := 0$,
$insertedTotal := 0$.

In this way, the percent of points we insert into the first queue tries to move between 65 and 75, and performance is close to ideal. The initial value of *cutoffStep*

is set using the heuristic at the end of section 4.3.1, as it is probably close to a cutoff which will give good performance. Even a poor initial cutoff will eventually be fixed, but the method may take some time before it converges on a good value.

We multiply by 1.5 and divide by 2 so that there is a convergence to a value (if we used the same number we may end up constantly overshooting and undershooting a value). We chose to divide by the higher number because too low a percent is fixed more quickly than too high of a percent (since the first queue will become empty quicker and that is when *cutoffStep* is adjusted). An area for future research would be to find another way to adjust the queue cutoff so that it converges on an ideal percent more quickly (perhaps by not waiting until the first queue becomes empty).

5. Method uses. We have outlined two basic methods for solving the isotropic Eikonal equation with a few different variations of each of them. Here we detail the strengths and weaknesses of each variation.

The locking sweeping method is the first attempt to reduce the number of calculations performed by the fast sweeping method. This outperforms the fast sweeping method on all except the most trivial examples. Its computational complexity is $O(n)$ but with a smaller constant than the fast sweeping method. When the speed function has large variations, the locking sweeping method is much faster than the fast sweeping method, but on small to medium size problems such as the ones we tested, this method still takes longer than the fast marching method. Like the fast sweeping method, this method is also very simple and extremely easy to code.

The single queue method is an attempted improvement on the locking sweeping method which eliminates the need to sweep over points where calculations are unnecessary. This method performs optimally when the speed is constant throughout the entire computation area. It also performs well on piecewise constant speed profiles, where it is generally the fastest method of those described in this paper. Slight changes within the computation area, even as small as 1% of the speed, are enough to slow this method down and make one of the two queue methods preferable. We note that this algorithm is essentially the D'Esopo–Pape algorithm for finding the shortest path on a network adapted to our problem. We also note that there are two well-known procedures to improve the D'Esopo–Pape algorithm, small label first (SLF), and large label last (LLL). These methods try to keep small arrival times at the front of the queue and large labels at the end of the queue. We refer the interested reader to the second chapter of [3].

The static two queue method uses a heuristic based on the speed profile to provide a means to segregate the points with predicted low arrival times from the points with predicted high arrival times. This heuristic is reasonably accurate when the speed is uniform or uniformly distributed throughout the computation area. So the method will perform well when the speed is uniformly distributed throughout the region, such as in section 7.2, but not so well when the speeds change greatly over the computation area (section 7.6). In this example, using an ideal queue cutoff, rather than the heuristic, still performs more calculations than the dynamic two queue method.

The dynamic two queue method is designed to address this concern of the static two queue method. The queue cutoff is allowed to change as the computation progresses. In our trials this method has always performed well, always outperforming both the fast sweeping method and the fast marching method. It does take some time to compute the new cutoff, so on problems where the static two queue method works well, this method is slightly slower, though still very good. However, there are

TABLE 1

The L_1 and L_∞ error using the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods on the point source problem. Each method calculates the exact same arrival times, and the errors clearly demonstrate first order convergence.

Size	FS	FS	LS	LS	SQ	SQ	TQS	TQS	TQD	TQD	FM	FM
	L_1	L_∞	L_1	L_∞	L_1	L_∞	L_1	L_∞	L_1	L_∞	L_1	L_∞
80	0.0309	0.0174	0.0309	0.0174	0.0309	0.0174	0.0309	0.0174	0.0309	0.0174	0.0309	0.0174
160	0.0183	0.0105	0.0183	0.0105	0.0183	0.0105	0.0183	0.0105	0.0183	0.0105	0.0183	0.0105
320	0.0106	0.0061	0.0106	0.0061	0.0106	0.0061	0.0106	0.0061	0.0106	0.0061	0.0106	0.0061
640	0.0061	0.0035	0.0061	0.0035	0.0061	0.0035	0.0061	0.0035	0.0061	0.0035	0.0061	0.0035
1280	0.0034	0.0020	0.0034	0.0020	0.0034	0.0020	0.0034	0.0020	0.0034	0.0020	0.0034	0.0020

many problems where there is no good static cutoff and a dynamic cutoff is necessary for good performance. We again note that the two queue methods are essentially the threshold method, [6], for finding the shortest path on a network adapted to our problem. The SLF and LLL algorithms mentioned above can also be applied to the two queue methods to improve performance.

6. Accuracy. All of the methods described in this paper solve the nonlinear system of equations obtained using the first order discretization scheme (2.1) applied at each grid point. This paper is primarily concerned with the efficiency each method solves this system of equations, but we first give a brief demonstration that these methods are first order. The efficiency of these methods are explored in the next section.

To demonstrate the first order accuracy we solve two classic problems. The first is a single point source with constant speed:

$$|\nabla T(\vec{x})| = 1, \vec{x} \in [-2,2] \times [-2,2], \tag{6.1}$$

$$T(0,0) = 0. \tag{6.2}$$

The exact solution for this problem is $T(x,y) = \sqrt{x^2+y^2}$. We solve this problem for decreasing step sizes and tabulate the L_1 and L_∞ errors for each method in Table 1. These errors clearly show first order convergence for all methods. Note that the arrival times found by each method are identical.

The second problem is

$$|\nabla T(x)| = 1, x \in [-2,2] \times [-2,2], \tag{6.3}$$

$$T(-1,-1) = 0, T(1,1) = 0. \tag{6.4}$$

The exact solution for this problem is $T(x,y) = \min(\sqrt{(x-1)^2+(y-1)^2}, \sqrt{(x+1)^2+(y+1)^2})$. The solution for this example is not smooth along the line $y = -x$. We solve this problem for decreasing step sizes and tabulate the L_1 and L_∞ errors for each method in Table 2. Once again the arrival times found by each method are identical, and the errors clearly show first order convergence.

7. Examples. In this section we test the efficiency of our new algorithms on a number of examples. While the purpose of this paper is to improve the fast sweeping method, we also give results for the fast marching method for added perspective. It is already well known, [7], that the fast sweeping method is generally faster when the speed function is roughly constant, and the fast marching method is generally faster when the speed function has large changes. The purpose of our method is to improve the performance of the fast sweeping method when the speed function has large gradient.

TABLE 2

The L_1 and L_∞ error using the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods on a constant speed problem with two point sources. The arrival times for this problem are not smooth at points equidistant to the two point sources. Each method calculates the exact same arrival times, and the errors clearly demonstrate first order convergence.

Size	FS L_1	FS L_∞	LS L_1	LS L_∞	SQ L_1	SQ L_∞	TQS L_1	TQS L_∞	TQD L_1	TQD L_∞	FM L_1	FM L_∞
80	0.0256	0.0133	0.0256	0.0133	0.0256	0.0133	0.0256	0.0133	0.0256	0.0133	0.0256	0.0133
160	0.0156	0.0082	0.0156	0.0082	0.0156	0.0082	0.0156	0.0082	0.0156	0.0082	0.0156	0.0082
320	0.0092	0.0049	0.0092	0.0049	0.0092	0.0049	0.0092	0.0049	0.0092	0.0049	0.0092	0.0049
640	0.0053	0.0028	0.0053	0.0028	0.0053	0.0028	0.0053	0.0028	0.0053	0.0028	0.0053	0.0028
1280	0.0030	0.0016	0.0030	0.0016	0.0030	0.0016	0.0030	0.0016	0.0030	0.0016	0.0030	0.0016

TABLE 3

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods on the point source problem. The regular fast sweeping method does the most calculations while the single queue method performs optimally. The fast marching method is the slowest, followed by the regular fast sweeping method. The locking sweeping method and the single queue method finish calculating in the quickest amount of time.

Size	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
80	5	0	1.280	0	1	0.01	1.381	0	1.427	0	1.975	0
160	5	0.02	1.265	0.01	1	0.01	1.483	0.02	1.591	0.011	1.988	0.01
240	5	0.06	1.260	0.02	1	0.02	1.538	0.03	1.641	0.04	1.992	0.05
320	5	0.1	1.258	0.04	1	0.03	1.571	0.05	1.657	0.05	1.994	0.08
400	5	0.17	1.256	0.071	1	0.05	1.593	0.08	1.669	0.08	1.995	0.13
480	5	0.18	1.255	0.1	1	0.07	1.613	0.12	1.680	0.13	1.996	0.18
560	5	0.26	1.254	0.13	1	0.09	1.626	0.15	1.697	0.161	1.996	0.261
640	5	0.341	1.254	0.17	1	0.13	1.634	0.221	1.702	0.23	1.997	0.33
720	5	0.42	1.253	0.21	1	0.151	1.642	0.26	1.707	0.291	1.997	0.431
800	5	0.52	1.253	0.261	1	0.2	1.650	0.351	1.710	0.35	1.998	0.541
880	5	0.63	1.253	0.31	1	0.241	1.656	0.411	1.717	0.43	1.998	0.661
960	5	0.751	1.253	0.37	1	0.291	1.660	0.501	1.722	0.53	1.998	0.801
1040	5	0.882	1.252	0.43	1	0.35	1.663	0.611	1.723	0.611	1.998	0.942
1120	5	1.021	1.252	0.51	1	0.43	1.666	0.711	1.722	0.721	1.998	1.112
1200	5	1.172	1.252	0.591	1	0.491	1.667	0.821	1.727	0.851	1.998	1.311
1280	5	1.342	1.252	0.661	1	0.541	1.669	0.951	1.730	0.971	1.998	1.512

7.1. Point source, constant speed. In this example, the computation area has a constant speed of one everywhere and the source is at the center. The fast marching method finishes calculating last, followed by the regular fast sweeping method, even though the fast sweeping method finishes after only five passes through the computation area. The single queue method performs optimally by computing each point once. The two queue methods perform slightly worse on this example, but are still better than the regular fast sweeping method and the fast marching method. The relative calculation times and the calculations per grid point are given in Table 3. Calculations were performed starting with a computation area with $80 \times 80 = 6400$ points and increasing the width and height in increments of 80, up to $1280 \times 1280 = 1,638,400$ points. We are interested in improving the performance of the fast sweeping method, especially when there are sharp changes in speed, and will now show a number of examples that are generally problematic for the fast sweeping method. This will be the only example we show where the regular fast sweeping method outperforms the fast marching method.

TABLE 4

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods with random speeds between 1 *and* 10. *The regular fast sweeping method does the most calculations while the two queue methods and the fast marching method have the least. The fast sweeping, locking sweeping, and single queue methods perform terribly on this problem both in terms of calculation time and CPG. The two queue methods perform the calculations the quickest. The fast marching method takes about twice as long on the larger examples.*

Size	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
80	29	0.03	5.19	0.01	4.02	0.01	1.63	0	1.888	0.01	1.974	0.01
160	41	0.16	9.448	0.05	7.003	0.05	1.634	0.02	1.842	0.02	1.987	0.02
240	57	0.38	13.049	0.231	9.992	0.161	1.660	0.03	1.869	0.04	1.991	0.05
320	61	0.711	15.787	0.5	11.729	0.36	1.671	0.06	1.865	0.06	1.993	0.09
400	77	1.392	17.977	0.911	13.342	0.671	1.667	0.09	1.901	0.1	1.994	0.13
480	89	2.333	21.522	1.542	15.662	1.232	1.669	0.121	1.9066	0.141	1.995	0.2
560	101	3.535	23.0472	2.314	17.070	1.903	1.663	0.17	1.9021	0.19	1.996	0.28
640	109	4.937	24.165	3.154	17.939	2.664	1.659	0.211	1.900	0.251	1.996	0.38
720	121	6.929	25.953	4.377	19.359	3.705	1.658	0.28	1.900	0.32	1.996	0.481
800	125	8.802	27.598	5.858	21.031	5.037	1.662	0.351	1.899	0.4	1.997	0.591
880	133	11.696	29.090	7.31	21.889	6.389	1.662	0.451	1.859	0.481	1.997	0.741
960	133	13.69	30.301	9.233	22.364	7.791	1.659	0.531	1.827	0.581	1.997	0.891
1040	153	18.106	31.073	11.657	23.351	9.604	1.660	0.631	1.898	0.751	1.997	1.081
1120	149	20.48	33.268	13.77	25.027	12.237	1.663	0.741	1.832	0.832	1.997	1.272
1200	165	26.238	35.109	17.265	26.429	14.871	1.660	0.882	1.906	1.002	1.998	1.493
1280	185	33.007	36.878	20.92	27.772	17.605	1.660	1.042	1.864	1.152	1.998	1.743

7.2. Random speeds 1–10. This example considers a computation area free of obstacles and with an integer speed at each point randomly chosen between 1 and 10, inclusive. The source is in the center of the computation area. The random speeds produce many changes in the characteristic direction, depending on the size of the computation area. For this reason, methods that depend on the changes in the characteristic, particularly the regular fast sweeping method and the locking method perform progressively worse on progressively larger examples. The nonuniform speed also makes the single queue method perform poorly. The fastest method is the two queue static cutoff method, closely followed by the two queue dynamic cutoff method and then the fast marching method. Calculations were performed starting with a computation area with $80 \times 80 = 6400$ points and increasing the width and height in increments of 80, up to $1280 \times 1280 = 1,638,400$ points. The calculations per grid point and the calculation time are displayed in Table 4. This is a particularly difficult problem for the fast sweeping method as well over 50 sweeps are required on the largest examples. The method does not appear linear because more randomness is introduced as the size of the problem gets bigger (and thus the number of sweeps required also increases).

7.3. Random speeds 1–10,000. In this example we consider a computation area that contains random integer speeds between 1 and 10,000, inclusive, at every point with the source point at the center. This makes the characteristic change many times and therefore requires many sweeps by the fast sweeping method. The results are similar to that of the random 1–10 example in section 7.2, except that methods with dependencies on the speed profile (regular fast sweeping, locking), perform even worse. The nonuniform nature of the speed profile also makes the single queue method perform poorly. The number of calculations per grid point and the calculation time is shown in Table 5.

TABLE 5

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods with random speeds between 1 *and* 10,000. *The regular fast sweeping method does the most calculations while the two queue methods and the fast marching method have the least. The fast sweeping, locking sweeping, and single queue methods perform terribly on this problem both in terms of calculation time and CPG. The two queue methods perform the calculations the quickest. The fast marching method takes about twice as long on the larger examples.*

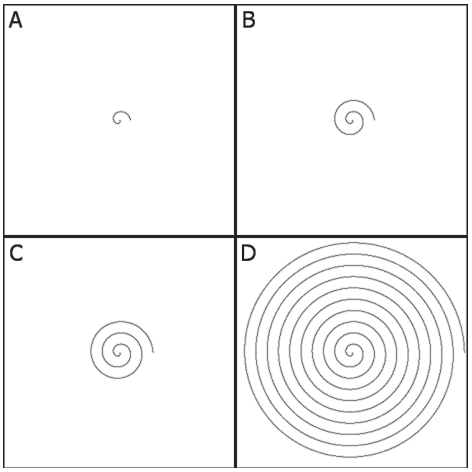
Size	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
80	25	0.08	4.586	0.01	4.447	0.01	1.806	0	2.083	0.01	1.975	0
160	45	0.3	10.681	0.07	8.619	0.07	1.778	0.01	2.089	0.02	1.988	0.02
240	65	0.461	13.034	0.251	11.190	0.18	1.809	0.04	2.068	0.04	1.992	0.05
320	73	0.881	17.995	0.581	14.006	0.441	1.811	0.05	2.067	0.06	1.994	0.091
400	77	1.452	19.469	1.001	16.002	0.851	1.804	0.091	2.105	0.1	1.995	0.14
480	89	2.404	22.820	1.673	17.887	1.442	1.810	0.13	2.099	0.141	1.996	0.21
560	101	3.685	25.368	2.523	19.395	2.243	1.802	0.18	2.067	0.19	1.996	0.291
640	105	4.987	26.660	3.465	20.954	3.225	1.812	0.241	2.103	0.251	1.997	0.391
720	117	7	29.138	4.837	23.543	4.646	1.813	0.31	2.100	0.32	1.997	0.511
800	129	9.514	32.177	6.499	25.379	6.329	1.812	0.38	2.061	0.4	1.998	0.651
880	141	12.588	35.898	8.663	28.349	8.933	1.802	0.48	2.054	0.481	1.998	0.801
960	149	15.933	37.535	10.926	30.222	10.856	1.812	0.611	2.107	0.581	1.998	0.981
1040	169	20.87	38.626	13.92	30.606	12.889	1.809	0.711	2.037	0.751	1.998	1.181
1120	169	24.215	39.388	16.423	32.366	15.903	1.810	0.852	2.104	0.832	1.998	1.412
1200	189	31.555	42.223	20.65	34.162	19.498	1.806	0.982	2.059	1.002	1.998	1.663
1280	193	35.982	42.701	23.354	35.510	22.833	1.809	1.162	2.103	1.152	1.998	1.952

7.4. Constant speed spirals. In this example we consider a computation area of size 640×640 that has constant speed everywhere with a point source at the middle. There is an impassable spiral wall that emits outward from the source point which requires that the characteristic change direction often. We consider many similar speed profiles where the spiral turns an increasing number of times. The number of spins in the spiral determines the number of sweeps necessary. Figure 9 shows the speed profiles. Table 6 shows the calculations per grid point and calculation time of the methods. The fast sweeping method has a large dependence on the number of turns, while the locking sweeping scheme has a much smaller one. The other methods do not depend on the number of turns in the spiral. The fastest method on this example is the single queue method, as the speed is constant at every passable point. The spirals are generated using the following C code:

$a = 5;$
$for(t = 0.001; t <= numTurns * 2 * pi; t+ = 0.001)$
$speed[(int)(a * t * cos(t))][(int)(a * t * sin(t))] = 0;$

where a is the number of grid points between spiral walls, $numTurns$ is the number of turns in the spiral, *speed* is a 2D array of the speeds at every point, and (int) is an explicit type cast.

7.5. Random speed spirals. This example is similar to the constant spiral speed example in section 7.4 except that instead of a constant speed everywhere, the speed is randomly selected to be either 100 or 101 at every point. The computation area is 640×640 and the source is at the center. This slight change in speed is enough to change the characteristic to require the regular fast sweeping method to do six sweeps on a spiral with one turn instead of two sweeps (which would happen if the speed was constant). Table 7 displays the number of calculations per grid



(a) Images of the speed profiles used. A shows a spiral example with one turn, B has two turns, C has three turns, D has ten turns. The dark region indicates speed zero (the wave can not pass through these points). The source is located at the center of the spiral.

FIG. 9. *The speed profile for the constant speed spiral speed profile presented in section* 7.4.

TABLE 6

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods with increasing numbers of spirals on a 640 *by* 640 *grid. The performance of the fast sweeping method depends heavily on the number of spirals. The performance of the locking sweeping method also depends on the number of spirals, but this dependence is greatly reduced. The rest of the methods display very little dependence on the number of spirals. The single queue methods perform the calculations the quickest because the speed is constant.*

Turns	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
1	9	0.501	1.265	0.21	1.020	0.131	1.577	0.2	1.638	0.21	1.997	0.34
2	13	0.671	1.241	0.24	1.136	0.14	1.546	0.2	1.635	0.2	1.996	0.33
3	17	0.862	1.215	0.27	1.033	0.13	1.677	0.2	1.750	0.22	1.994	0.34
4	21	1.022	1.219	0.31	1.011	0.12	1.748	0.22	1.774	0.22	1.992	0.341
5	25	1.201	1.219	0.35	1.004	0.13	1.773	0.22	1.806	0.22	1.990	0.331
6	29	1.342	1.210	0.39	0.996	0.13	1.785	0.23	1.766	0.22	1.987	0.331
7	33	1.502	1.195	0.421	0.990	0.131	1.773	0.22	1.762	0.22	1.983	0.32
8	37	1.672	1.169	0.461	0.985	0.12	1.786	0.22	1.745	0.21	1.979	0.33
9	41	1.822	1.136	0.49	0.978	0.13	1.768	0.22	1.726	0.22	1.974	0.33
10	45	1.983	1.096	0.531	0.972	0.13	1.673	0.2	1.646	0.2	1.969	0.321

point and calculation time for each of the methods versus the number of turns in the spiral. Again, the regular fast sweeping method and the locking method display a linear dependence on the number of turns in the spiral. The small change in speed (varying by 1/100th) was enough to slow down the single queue method and make it take 400% of the calculation time of the same method on the constant speed spiral example (0.13 seconds on constant speed and 0.521 seconds on the slightly random speed). The two queue methods' calculation time and the fast marching method's calculation time appear unaffected by the slight changes in speed.

TABLE 7

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods with increasing numbers of spirals and a small variations in speed on a 640 *by* 640 *grid. The performance of the fast sweeping method depends heavily on the number of spirals. The performance of the locking sweeping method also depends on the number of spirals, but this dependence is greatly reduced. The two queue and the fast marching methods display very little dependence on the number of spirals. The two queue methods are the fastest. The slight change in speed significantly slows down the single queue method, making it run in* 400% *of the time that it took when the speeds were constant (*0.13 *seconds on constant speed versus* 0.521 *seconds on the slightly random speed).*

Turns	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
1	25	1.302	1.730	0.41	1.481	0.19	1.556	0.21	1.643	0.22	1.996	0.351
2	29	1.462	1.810	0.441	1.612	0.2	1.537	0.2	1.613	0.23	1.995	0.35
3	37	1.832	1.571	0.51	2.001	0.23	1.662	0.22	1.765	0.25	1.994	0.34
4	37	1.823	1.501	0.491	2.430	0.27	1.728	0.22	1.771	0.23	1.992	0.341
5	41	2.013	1.455	0.491	2.289	0.26	1.755	0.24	1.806	0.23	1.989	0.331
6	49	2.363	2.563	0.741	2.636	0.3	1.780	0.23	1.777	0.241	1.986	0.341
7	49	2.433	3.031	0.781	5.289	0.571	1.786	0.23	1.785	0.221	1.982	0.34
8	53	2.584	2.249	0.751	4.322	0.46	1.796	0.23	1.769	0.231	1.978	0.33
9	57	2.764	2.624	0.751	4.024	0.441	1.767	0.22	1.721	0.231	1.974	0.331
10	57	2.734	2.314	0.791	4.837	0.521	1.668	0.21	1.647	0.211	1.969	0.341

7.6. Horizontally increasing speeds. This example focuses on a speed profile which increases horizontally from the center. The source is located in the center of the computation area. Since the fastest way to get to a point under the source is to first go to a side, then go down, then come back, the single queue method continuously finds better paths to these points. This results in terrible performance for the single queue method, as times are recalculated many times before the correct one is reached. Examples such as this one are our motivation in section 4.3 to create a two queue method. Also, the heuristic used for the two queue static cutoff method is a poor one in this example, and the performance is near the single queue method's. The heuristic is bad because the speed varies by such a significant amount that the average speed is not representative of the entire area. A heuristic based on the median may perform better on this particular example, but further analysis has revealed that no single queue cutoff performs as well as the dynamic two queue method which adjusts the cutoff during computation. The speed profile is displayed in Figure 5. The number of calculations per grid point and calculation time for the different methods is shown in Table 8. Calculations were performed starting with a computation area with $80 \times 80 = 6400$ points and increasing the width and height increments of 80, up to $1280 \times 1280 = 1{,}638{,}400$ points. Comparisons of calculation times for different static queue cutoffs are provided in Figure 10.

8. Conclusions. In this paper we have outlined a number of improvements to the fast sweeping method on rectangular grids. These improvements are aimed at improving performance and do not effect the answer obtained. The first speedup was to skip calculations of points where a calculation would not yield useful information. This was accomplished by maintaining a lock structure at each point. The next change was to remove the Gauss–Seidal iterations in order to stop iterating over locked points. This was accomplished by using a queue, which has $O(1)$ insert and remove operations. This method is extremely fast for some problems, in particular problems with piecewise constant speed. We showed that using a single queue, however, can in some cases result in many unnecessary computations and may actually

TABLE 8

The number of calculations per grid point (CPG) and calculation time performed by the fast sweeping (FS), locking sweeping (LS), single queue (SQ), two queue static (TQS), two queue dynamic (TQD), and fast marching (FM) methods with horizontally increasing speeds. The single queue and two queue static method perform very poorly on this problem. The fast marching method, locking sweeping method, and two queue dynamic method all perform about the same number of calculations per grid point. The two queue dymanic and locking schemes are the fastest because no sorting is required to determine the ordering.

Size	FS CPG	FS time	LS CPG	LS time	SQ CPG	SQ time	TQS CPG	TQS time	TQD CPG	TQD time	FM CPG	FM time
80	9	0.02	1.924	0.01	1.837	0	1.639	0.01	1.347	0	1.975	0
160	9	0.11	1.968	0.01	2.986	0.03	2.590	0.02	1.635	0.02	1.9875	0.02
240	9	0.12	1.973	0.04	4.035	0.061	3.669	0.06	1.819	0.04	1.991	0.05
320	9	0.13	1.974	0.07	4.984	0.15	4.696	0.151	1.984	0.07	1.993	0.09
400	9	0.21	1.974	0.11	5.861	0.36	5.685	0.36	2.069	0.1	1.995	0.13
480	9	0.31	1.974	0.17	6.698	0.68	6.597	0.691	2.114	0.15	1.995	0.19
560	9	0.411	1.974	0.221	7.504	1.111	7.466	1.112	2.137	0.201	1.996	0.27
640	9	0.551	1.974	0.28	8.290	1.663	8.280	1.683	2.194	0.28	1.996	0.351
720	9	0.691	1.974	0.35	9.048	2.323	9.046	2.373	2.144	0.351	1.997	0.451
800	9	0.861	1.974	0.431	9.793	3.244	9.792	3.285	2.120	0.441	1.997	0.561
880	9	1.041	1.974	0.541	10.531	4.176	10.531	4.226	2.079	0.531	1.997	0.691
960	9	1.222	1.974	0.621	11.258	5.277	11.258	5.368	2.094	0.651	1.997	0.822
1040	9	1.463	1.974	0.721	11.977	6.599	11.977	6.67	1.998	0.721	1.998	0.971
1120	9	1.682	1.974	0.861	12.685	8.101	12.685	8.182	1.944	0.811	1.998	1.152
1200	9	1.953	1.974	0.961	13.389	10.044	13.389	10.155	1.958	0.931	1.998	1.322
1280	9	2.244	1.974	1.101	14.085	11.967	14.085	12.367	2.032	1.132	1.998	1.522

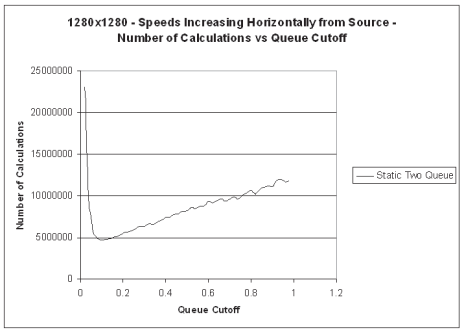


FIG. 10. *The number of calculations for different static queue cutoffs for the horizontally increasing speed example in section* 7.6 *on a* 1280×1280 *computation area. This method is capable of performing well when the queue cutoff is about* 0.1. *However, the heuristic computes* $1.5/AvgSpeed = 1.5/80.625 = 0.018605$, *which is a particularly bad queue cutoff. Using a safe cutoff of* $1/MinSpeed = 1/1 = 1$ *yields less calculations, but it still does* 150% *more calculations than it would with the the ideal cutoff. With an ideal cutoff this method takes about* 1.5 *seconds to finish computing; with the safe cutoff it takes about* 3.8 *seconds to finish. The dynamic queue cutoff scheme performs about* 3,500,000 *calculations on this example, less than any cutoff does with the static two queue method.*

hurt performance. This was remedied by using a second queue to separate points with predicted high arrival times from points with predicted lower arrival times. This is possible because the Godunov upwind difference scheme progresses the computation from smaller arrival times to larger arrival times. The segregation method was discussed, and a segregation heuristic based on the speed profile was given. Also, a method that dynamically changes the way to separate the predicted small arrival

times from the predicted large arrival times was given. This method works well on all tested speed profiles. Given a reasonable segregation method, the two queue method outperforms the fast sweeping method in all cases we have tested.

Finally, we give some comments on the extension of our methods to triangular domains and convex anisotrophic media. These methods are derived from shortest path algorithms which do not have an underlying grid structure, so we expect that they will carry over to triangulated domains without incident. This will be the subject of a future work. It would also be interesting to apply these methods to anisotropic media, but for anisotropic problems computing the arrival times in increasing order does not decouple the system of equations if the characteristic and the gradient are not in the same quadrant (see [20] and [13]). We expect that the locking scheme can still be applied as long as all of the neighbors are unlocked instead of just unlocking the neighbors with smaller arrival time. It is not clear how this would affect the two queue method because the method tries to calculate the arrival times in an "almost increasing" order.

Acknowledgments. We have benefited by discussions with Alexander Vladimirsky and Liz Rachele.

REFERENCES

[1] M. Boue and P. Dupuis, *Markov chain approximations for deterministic control problems with affine dynamics and quadratic costs in the control*, SIAM J. Numer. Anal., 36 (1999), pp. 667–695.
[2] M. G. Crandall and P. L. Lions, *Viscosity solutions of Hamilton-Jacobi equations*, Trans. Amer. Math. Soc., 277 (1983), pp. 1–42.
[3] D Bertsekas, *Network Optimization: Continuous and Discrete Models*, Athena Scientific, Boston, MA, 1998.
[4] Y. Cheng and C.-W. Shu, *A discontinuous Galerkin finite element method for directly solving the Hamilton-Jacobi equations*, J. Comput. Phys., 223 (2007), pp. 398–415.
[5] J. Helmsen, E. G. Puckett, P. Colella, and M. Dorr, *Two new methods for simulating photolithography development in 3D*, Proc. SPIE, 2736 (1996), pp. 253–261.
[6] F. Glover, D. Klingman, and N. Phillips, *New polynomial bounded shortest path algorithm*, Oper. Res., 33 (1985), pp. 65–73.
[7] P. A. Gremaud and C. M. Kuster, *Computational study of fast methods for the Eikonal equation*, SIAM J. Sci. Comput., 27 (2006), pp. 1803–1816.
[8] C. Y. Kao, S. J. Osher, and J. Qian, *Lax-Freidrichs sweeping schemes for static Hamilton-Jacobi equations*, J. Comput. Phys., 196 (2004), pp. 367–391.
[9] C. Y. Kao, S. J. Osher, and Y. H. Tsai, *Fast sweeping methods for static Hamilton–Jacobi equations*, SIAM J. Numer. Anal., 42 (2005), pp. 2612–2632.
[10] S. Kim, *An O(N) level set method for Eikonal equations*, SIAM J. Sci. Comput., 22 (2001), pp. 2178–2193.
[11] R. Kimmel and J. A. Sethian, *Fast marching methods on triangulated domains*, Proc. Natl. Acad. Sci. USA, 95 (1998), pp. 8341–8435.
[12] F. Li, C. W. Shu, Y. T. Zhang, and H. Zhao, *A second order discontinuous Galerkin fast sweeping method for Eikonal equations*, J. Comput. Phys., 227 (2008), pp. 8191–8208.
[13] J. Qian, Y. Zhang, and H. Zhao, *A fast sweeping method for static convex Hamilton–Jacobi equations*, J. Sci. Comput., 31 (2007), pp. 237–271.
[14] J. Qian, Y. Zhang, and H. Zhao, *Fast sweeping methods for Eikonal equations on triangular meshes*, SIAM J. Numer. Anal., 45 (2007), pp. 83–107.
[15] S. J. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Springer, New York, 2002.
[16] J. A. Sethian, *A fast marching level set method for monotonically advancing fronts*, Proc. Natl. Acad. Sci. USA, 93 (1996), pp. 1591–1595.
[17] J. A. Sethian, *Fast marching methods*, SIAM Rev., 41 (1999), pp. 199–235.
[18] J. A. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Sciences*, Cambridge University Press, Cambridge, UK, 1996.

[19] J. A. Sethian and A. Vladimirsky, *Fast methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes*, Proc. Natl. Acad. Sci. USA, 97 (2000), pp. 5699–5703.
[20] J. A. Sethian and A. Vladimirsky, *Ordered upwind methods for static Hamilton-Jacobi equations: Theory and algorithms*, SIAM J. Numer. Anal., 41 (2003), pp. 325–363.
[21] C. W. Shu, *High Order Numerical Methods for Time Dependent Hamilton–Jacobi Equations*, in Mathematics and Computation in Imaging Science and Information Processing, Lect. Notes Ser. Inst. Math. Sci. Natl. Univ. Singap. 11, S. S. Hoh, A. Ron, and Z. Shen, eds., World Scientific, Singapore, 2007, pp. 47–91.
[22] R. Tsai, L. T. Cheng, S. J. Osher, and H. K. Zhao, *Fast sweeping algorithms for a class of Hamilton–Jacobi equations*, SIAM J. Numer. Anal., 41 (2003), pp. 673–594.
[23] J. N. Tsitsiklis, *Efficient algorithms for globally optimal trajectories*, IEEE Trans. Automat. Control., 40 (1995), pp. 1528–1538.
[24] L. Yatziv, A. Bartesaghi, and G. Sapiro, *$O(N)$ implementation of the fast marching algorithm*, J. Comput. Phys., 212 (2005), pp. 393–399.
[25] Y. Zhang, H. Zhao, and S. Chen, *Fixed-point iterative sweeping methods for static Hamilton-Jacobi equations*, Methods Appl. Anal., 13 (2006), pp. 299–320.
[26] Y.-T. Zhang, H.-K. Zhao, and J. Qian, *High order fast sweeping methods for static Hamilton-Jacobi equations*, J. Sci. Comput., 29 (2006), pp. 25–56.
[27] H. Zhao, *A fast sweeping method for Eikonal equations*, Math. Comp., 74 (2004), pp. 603–627.