CONTINUOUS ENERGETICALLY OPTIMAL PATHS ACROSS LARGE DIGITAL ELEVATION DATA SETS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Jason Rickwald

July 2007

APPROVAL PAGE

TITLE: Continuous Energetically Optimal Paths Across Large Digital Elevation Data Sets

AUTHOR: Jason Rickwald

DATE SUBMITTED: July 2007

Dr. Zoë Wood	
Advisor or Committee Chair	Signature
Dr. Michael Haungs	
Committee Member	Signature
Dr. Franz Kurfess	
Committee Member	Signature

Abstract

Continuous Energetically Optimal Paths Across Large Digital Elevation Data Sets

by

Jason Rickwald

Until recently in human history, walking has been the primary mode of transportation. Because of this, the paths taken and the distances traveled by someone on foot are of significant interest to archaeologists, anthropologists, and historians. Previous work has focused on developing tools that use a human-centered metric, taking into account the geography between two points when considering possible paths. This thesis presents improvements for finding energetically optimal paths over large scale digital elevation data.

Specifically, we present tools which can support gigabytes of digital elevation data. Furthermore, Dijkstra's shortest path algorithm, which determines the cost to a data point using only graph edges, is traded for the fast marching algorithm, which accounts for continuous paths across the surface of the data. Other contributions of this work include an algorithm for fast marching over large data sets that is based on a domain decomposition method intended for parallelizing fast marching. Also, the tools are demonstrated on digital elevation data for a large portion of Oregon, and recommendations for future improvements are made based on the observed results.

Acknowledgements

Many thanks to Zoë Wood for her constant encouragement, insights, and aid through the development of this thesis. Also, thank you to Hughes Hoppe and Steven Gortler for their interest at the beginning of this work and their help and advice regarding discrete geodesic algorithms and the fast marching method.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For most of the time that man has roamed the Earth, this roaming has been done on foot. Therefore, tools for the analysis of terrain and modeling of pedestrian travel are of much interest to archaeologists, anthropologists, and historians. Recently, work has been done to create tools that use a human-centered metric to model paths and distance over geography. Specifically, the metric used for distance calculation and path creation in [1, 2] is the number of calories expended by a person to traverse terrain. Intuitively, this models a human's desire to minimize the energy used in travel. These tools can be applied to archaeological study to give a more interesting definition of geographic proximity. Traditionally, "as the crow flies" distance is used to determine, say, the proximity of an obsidian quarry to a village. However, this metric might take a traveler right through a mountain range. Energetic distance accounts for the difficulty of traveling around or through difficult areas like mountains.

The tool presented in [1, 2], called Energetic Analyst, was worthwhile for showing that such human-centered metrics could be useful to many fields of study. However, it was rather limited in its scope. This work aims to address some key limitations of Energetic Analyst. First, the tools in this work can support multiple gigabytes of digital elevation data. Secondly, this work uses the fast marching algorithm [3, 4, 5] to find lowest energetic paths. Though fast marching still only provides approximations, it provides more accurate results than Dijkstra's shortest path algorithm, which was used by Energetic Analyst. Finally, the performance impact of running the fast marching algorithm on multiple gigabytes of data drove us to develop a domain decomposition-based method for fast marching that shortens running time with some cost to accuracy.

The general motive behind this work, then, is for a researcher to be able to use these tools to accurately model pedestrian travel over large digital elevation data sets. To test the success of this work, we will analyze digital elevation data for a large portion of Oregon. Path start and end points are placed near to where the last leg of the Oregon Trail enters and terminates in Oregon. The Oregon Trail is a historical path taken by many Western pioneers in the 19th century. Initial results show a strong correlation between the calculated path and the historically documented route of travel, though the dissimilarities will lead us to make recommendations for future improvements that may provide better results.

1.1 Problem Definition

Before continuing, it would be worthwhile to give a more formal definition of the Computer Science problem that this work aims to solve. We want a tool that can **find the discrete energetic geodesic over multiple gigabytes of digital elevation data for a single source and multiple destinations**. We can define *discrete energetic geodesic* similarly to the way *discrete geodesic* is defined in [6].

> The *Discrete Energetic Geodesic Problem* is to find the shortest path in the energetic (caloric) metric from source point s to destination point t such that the shortest path is constrained to lie on the surface of the mesh P.

Given that we are working with digital elevation data, which comes as a two dimensional grid of height values, our P can be a triangle or quad mesh made from this underlying height field. This mesh is inherently regular and manifold, which simplifies the algorithms for finding geodesics. Finally, when we say that we want to solve this problem for a single source and multiple destinations, we mean that a single run of a geodesic-finding algorithm should calculate the energetic distance from a selected point to all other data points in our digital elevation data. This will be useful for visualizing the caloric cost to anywhere on our data.

1.2 Our Contributions

Our contributions to the work below are outlined here.

- We have developed a set of tools for performing energetic analysis of very large digital elevation data sets.
- We have adapted the fast marching algorithm to calculate energetic costs.
- We have developed a domain decomposition-based fast marching algorithm that can run in about a third of the time of standard fast marching when working with large data sets.
- We have compared a known historical trail to a calculated energetically optimal trail in order to make recommendations for future improvements to the analysis tools.

Chapter 2

Previous Work

This section will describe work relevant to our continuous energetic analysis tools. First we will describe the previous tool that this work builds on. Next, we will discuss a number of algorithms capable of computing discrete geodesics. Finally, we describe the domain decomposition fast marching method that we modify for speedier fast marching of large data sets.

2.1 Energetic Analyst

The work presented here is based primarily on the Energetic Analyst tool developed by Brian Wood [1, 2]. The purpose of this tool was to aid archaeologists by using a human-centered metric for geographic proximity. Wood created Energetic Analyst for applications such as determining what nearby natural resources, such as obsidian quarries, were used by tribes. It could also be used to find trails between known prehistoric villages. Wood does little to prove the superiority of an energetic metric over previously used metrics; though he does show that lowest energetic paths can save a significant amount of energy when compared to more direct paths. Intuitively, real paths taken by humans or the decision to set out to a natural resource should be affected by one's desire to conserve energy. For more information on other metrics used by archaeologists and anthropologists, we refer the reader to Wood's work [1].

The equations used in Energetic Analyst were based on existing research for modeling human metabolism and energy expenditure under a variety of conditions [7, 8, 9, 10, 11]. We use the same equations in our work. First, we calculate the metabolic rate, in watts, for traveling between two points. The equation for this rate is different depending on if one is traveling downhill, Equation 2.1, or uphill, Equation 2.2.

$$MR = M - C \tag{2.1}$$

$$MR = M \tag{2.2}$$

$$M = 1.5w + 2.0(w + l)(\frac{l}{w})^2 + n(w + l)(1.5v^2 + 0.35vg) \tag{2.3}$$

$$C = n(\frac{g(w + l)v}{3.5} - \frac{(w + l)(g + 6)^2}{w} + 25 - v^2) \tag{2.4}$$

MR is the metabolic rate in watts

w is the person's weight in kilograms

l is the load carried in kilograms

v is the walking speed in meters per second

g is the percent grade

n is the terrain factor (1 is the terrain factor for a treadmill)

Once the metabolic rate has been calculated, we use the time required to travel between points in order to convert the metabolic rate to energetic cost in kilocalories. Unfortunately, these equations can under-predict caloric cost when traveling slowly downhill. To correct for this, the calculated cost is compared to the cost calculated from the standing metabolic rate. If the standing metabolic rate is greater, we use that.

$$SMR = 1.2 \times BMR \tag{2.5}$$

$$BMR_{male} = 66 + (13.7 \times w) + (5 \times h) - (6.8 \times a) \tag{2.6}$$

$$BMR_{female} = 655 + (9.6 \times w) + (1.7 \times h) - (4.7 \times a) \tag{2.7}$$

SMR is the Standing Metabolic Rate in watts

BMR is the Basal Metabolic Rate in watts

w is the person's weight in kilograms

h is the person's height in centimeters

a is the person's age in years

The Energetic Analyst tool reads in GIS[1] digital elevation models. Digital elevation models, abbreviated as "DEMs," are available from a number of sources, including the United States Geological Survey [12] and the National Geophysical Data Center [13]. Energetic Analyst supports DEMs in the ArcInfo ASCII DEM format originally developed by ESRI [14]. This format is supported by a number

[1]GIS stands for Geographic Information System. It is a broad term for a number of software systems used by researchers, planners, and government to store and analyze a wide range of geographic data.

of GIS software packages, particularly those developed by ESRI. However, the majority of DEM data available is in the USGS SDTS format. This data must be converted to use. Due to the simplicity of the ArcInfo ASCII DEM format as compared to the USGS SDTS format, we also only support the ArcInfo ASCII DEM format.

Energetic Analyst loads DEM data in and constructs a graph representation of it. Vertices in the graph are the data points from the DEM, edges are created from each vertex to its eight neighbors, and edge costs are created using the caloric cost equations from above. This graph is then used to construct lowest calorie paths and caloric terrains by running Dijkstra's shortest path algorithm on it. A *caloric terrain* is what Wood calls his visual representation of caloric proximity from a point. Energetic Analyst allows the user to view DEM data, caloric paths, and caloric terrains in an OpenGL [15] viewer window. Unfortunately, there is very little interactivity built into this application. For example, the user must enter path start and end points as degrees latitude and longitude rather than simply selecting points on the displayed terrain. This is a problem that we address in our own viewer application. Furthermore, the use of Dijkstra's algorithm constrains paths to lie on graph edges, which also affects the correctness of the caloric terrains. We try to improve correctness through our use of the fast marching algorithm. Finally, the Energetic Analyst tool tries to keep all digital elevation data and data structures in memory, which severely limits the size of the terrain that the user can work with. Wood reports that using a machine with 512 megabytes of RAM he is able to support DEMs up to about 650 x 650 at 90 meter resolution. We address this issue by only keeping some data in memory and swapping the rest to disk. Theoretically, this approach would allow us to support as much DEM data as can fit on your storage system[2]. The Oregon data that we use, for example, is 23951 x 14037 at 30 meter resolution.

2.2 Discrete Geodesic Algorithms

In this section we discuss three different algorithms that could be used to find discrete energetic geodesics on digital elevation data. They are Dijkstra's shortest path algorithm, which was used in [1, 2], the MMP algorithm [16, 17], and the fast marching algorithm [3], which we choose to use in our work due to its balance of accuracy and speed.

2.2.1 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm is probably the easiest algorithm to understand and to implement for finding discrete geodesics. It is a relatively old algorithm developed by Edsger W. Dijkstra for finding the shortest path between two vertices on a weighted graph [18]. As was described in Section 2.1, this algorithm can be easily adapted for finding lowest calorie paths by converting the elevation data grid to be a directed weighted graph with edges between each data point and its eight neighbors. Two edges are made between a point and a neighbor, as can be seen in Figure 2.1 — one for each direction of travel. The cost associated with an edge is the caloric cost of traversing that edge in its given direction. The inputs to Dijkstra's algorithm are this graph and a starting point. Also, if you want the algorithm to terminate with a particular path, it may take the end point as an input. The algorithm is as follows.

[2]Though, this being research work, it is very likely that we have a number of size restrictions imposed by our data structures.

1. Label the start vertex with a cost of 0 and add it to the set I. Vertices in set I are considered final, and their cost labels will not change.

2. Label the neighbors of the start vertex with the costs of traversing their adjacent edges. Add these neighbors to set F. Set F makes up a fringe of trial vertices. These vertices' costs are still subject to change.

3. Add all other vertices to set O and label them with a cost of ∞. Vertices in O have not been seen yet.

4. Loop until all vertices are in set I.

   (a) Pick a vertex, v from set F with the smallest cost label. Move v from F to I.

   (b) For each neighbor of v in F, update its cost label with the cost of v plus the cost of traversing the adjacent edge, but only if this cost is less than its current cost label.

   (c) For each neighbor of v in O, set its cost label with the cost of v plus the cost of traversing the adjacent edge. Remove the neighbor vertex from O and put it in F.

Many varieties to Dijkstra's shortest path algorithm have been proposed over the years, but all algorithms are similar to the one above. Usually, a priority queue is used to perform step 4a, which gives the algorithm a running time complexity of $O(NlogN)$. This is because the loop runs over all N vertices and it takes $logN$ time to add a vertex to the priority queue.

Dijkstra's shortest path algorithm is good for finding discrete energetic geodesics for a number of reasons. First, its simplicity makes it easy to implement, and keeps its running times and space requirements fairly low. Also, the regularity

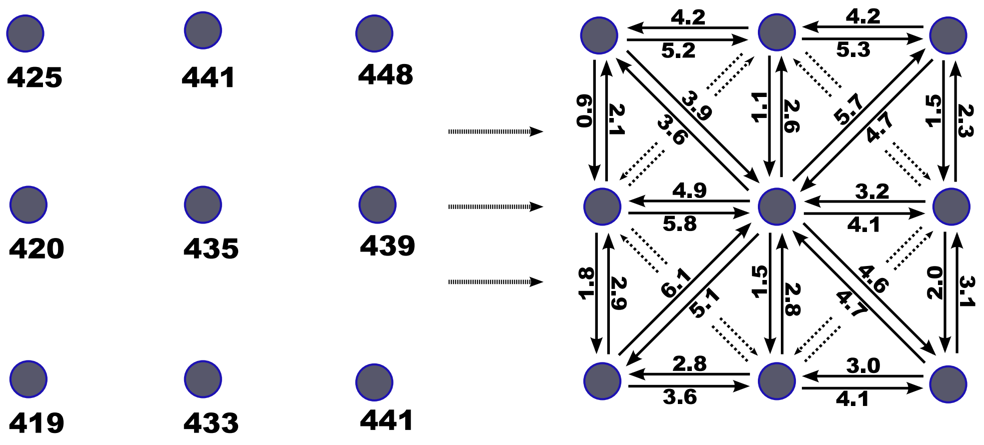


Figure 2.1: A demonstration of how a DEM grid, left, is converted to a weighted directed graph, right, that can be used as an input to Dijkstra's shortest path algorithm.

and density of our DEM data helps to reduce the amount of error caused by constraining our paths to lie on graph edges. For small DEMs, this error is quite tolerable. However, we decided to go with a more accurate method, as we wanted to be able to handle much larger DEMs without much error accumulating farther out from the start point. We did run a single test comparing the results of Dijkstra's shortest path algorithm to the results of the fast marching algorithm (described below) for a 11,976 x 6016 point DEM, and we found that the caloric costs calculated by Dijkstra's shortest path algorithm were 230 calories greater on average, with a maximum difference of 703 calories.

2.2.2 The MMP Algorithm

The MMP algorithm, sometimes referred to as "Continuous Dijkstra's," is an algorithm for finding exact discrete geodesics over triangle meshes. The algorithm and its data structures were originally defined by Mitchell, Mount, and Papadimitriou in 1987 [16]. However, it was not actually implemented and tested until 2005 when it was implemented in [17] by Surazhsky, Surazhsky, Kirsanov, Gortler, and Hoppe.

The algorithm works on a small number of basic principles which were proven in [16]. First, within a triangle the shortest path must be a straight line. Second, the shortest path that crosses an edge between two triangles must be a straight line in the planar unfolding of the two triangles. Finally, shortest paths may only pass through boundary vertices or vertices with total angle greater than or equal to 2π. The angle of a vertex is the sum of the angles formed by its adjacent edges. Vertices with total angle greater than 2π are called saddle vertices.

Using these principles, the algorithm proceeds by trying to track groups of shortest paths that can be parameterized together. Such a parameterization of a group of shortest paths is called a *window*. The algorithm stores one or more non-overlapping windows on each edge of the mesh. These windows are propagated out across the mesh in a Dijkstra-like sweep. An example window, w, is seen in Figure 2.2. A window, such as w, parameterizes a group of shortest paths as a 6-tuple $(b_0, b_1, d_0, d_1, \sigma, \tau)$. The endpoints of the window are b_0 and b_1, which are stored as small distance values along the edge. The values d_0 and d_1 are the distances to those endpoints, and the distance to any other point on the window can simply be found as the linear interpolation of those two values. Finally, σ is a binary value representing what side of the edge the source point is on. The parameter τ will be described momentarily.

Shortest paths that run through boundary or saddle vertices require some special treatment. There are conditions for which shortest paths will all follow the same strip of triangles back to such a vertex, s, and pass through it. Beyond vertex s, the shortest path back to the start point will be the simple geodesic leading from the start vertex to s. Therefore, when creating a window for a group

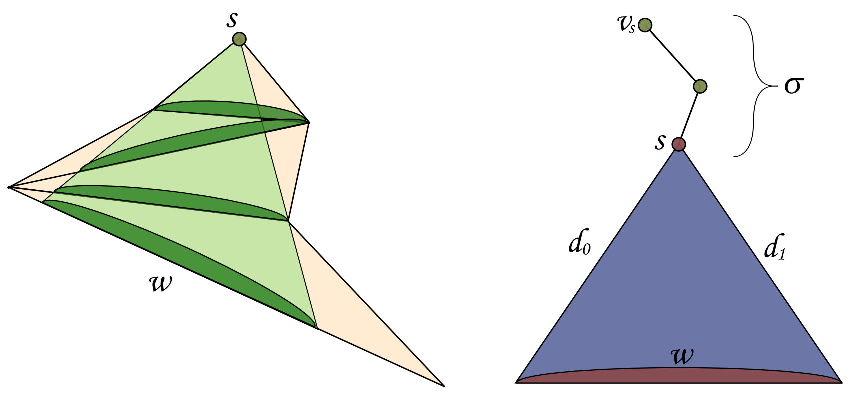


Figure 2.2: The left image shows windows that can be formed on the edges based on source vertex s. The shaded region represents the paths that form window w. The image on the right shows a window with a pseudo-source, and the simple path from the pseudo-source back to the source vertex.

of paths that lead back through s, we call s a pseudo-source. The distances d_0 and d_1 then represent the distance back to the pseudo-source, and τ is the distance from the pseudo-source back to the start point. This is also seen in Figure 2.2.

A few more things are necessary to do as we propagate windows out across the mesh. First, there cannot be overlapping windows on an edge, so the algorithm must choose a good intersection point for the two windows such that each window contains the shortest distances back to the source vertex. The authors of [17] elaborate on how this is done. Next, when a window w is adjacent to a boundary or saddle vertex, we may have to add extra windows for spots on edges that are not hit by propagating w. The boundary or saddle vertex becomes the pseudo-source for these windows. Finally, the authors of [17] note that the algorithm performs the best when windows are propagated out in a Dijkstra-like sweep. This means that we must keep a priority queue of windows with the lowest distance back to the source, and we pick the smallest each time we want to propagate a window.

The main benefit of the MMP algorithm is that it provides exact discrete geodesics over a triangle mesh. However, this comes at a cost. The running time and resource costs of the MMP algorithm are high — $O(N^2 logN)$ and $O(N^2)$ respectively. However, the authors of [17] note that these complexity results are rather pessimistic, and report better performance for typical meshes. They claim that this is because edges in a typical mesh will usually have a maximum of about $N^{\frac{1}{2}}$ windows, rather than the N windows that the authors of [16] use for their complexity analysis. Also, the authors of [17] develop a modification to the algorithm, called window merging, that improves space and time requirements while introducing some bounded error.

It seems possible that the MMP algorithm could be adapted to work with digital elevation data represented as a triangle mesh. However, we chose not to use the MMP algorithm for our energetic analysis tools for a number of reasons. First, it is a slightly more complex algorithm. Second, even with window merging, the space and time requirements make it a less appealing choice for working with multiple gigabytes of data. Third, we are working with a grid, not a triangulation. The grid can be triangulated, but is not inherently a triangle mesh. Finally, though it probably could be adapted to produce discrete energetic geodesics, this modification does not appear to be straightforward. For example, the second principle that the MMP algorithm operates on, that the shortest path crossing an edge is a straight line in the planar unfolding of the two triangles, is not true for shortest energetic paths.

2.2.3 The Fast Marching Algorithm

Another algorithm for finding discrete geodesics is the fast marching algorithm [3]. The fast marching algorithm is an algorithm that, like Dijkstra's shortest path algorithm, can only compute approximate discrete geodesics. However, unlike Dijkstra's algorithm, these approximate geodesics can cross faces in our mesh. Therefore, the results of the fast marching algorithm can be more accurate. Fast marching is actually an algorithm for tracking a wavefront over a surface, so it has a multitude of applications outside of finding geodesics. These include shape-from-shading, noise removal, and tracking interfaces in microchip fabrication [19, 5].

The fast marching algorithm and, more generally, level set methods, were developed to model and solve wavefront propagation problems. The goal is to solve the static Hamilton-Jacobi equation, also called the Eikonal equation, given below.

$$|\nabla T|F(p) = 1 \tag{2.8}$$

> Where $F(p)$ is the positive speed of the front at point p, and $T(p)$ is the arrival time of the front at point p. For finding the shortest distances, we say that $\frac{1}{F}$ is the travel cost and $T(p)$ is the distance traveled to p.

Efficient algorithms for solving Equation 2.8 were developed independently by Sethian [4, 5] and Tsitsiklis [20]. Their algorithms are rather similar, though each developed them using a different approach. Tsitsiklis saw it as a trajectory optimization problem, and describes an optimal control approach [20, 21]. Sethian developed a solution based on upwind numerical schemes [5, 21].

Fast marching works with the "boundary value formulation" of the problem. The algorithm tracks a wavefront by developing an arrival surface $T(p)$ that

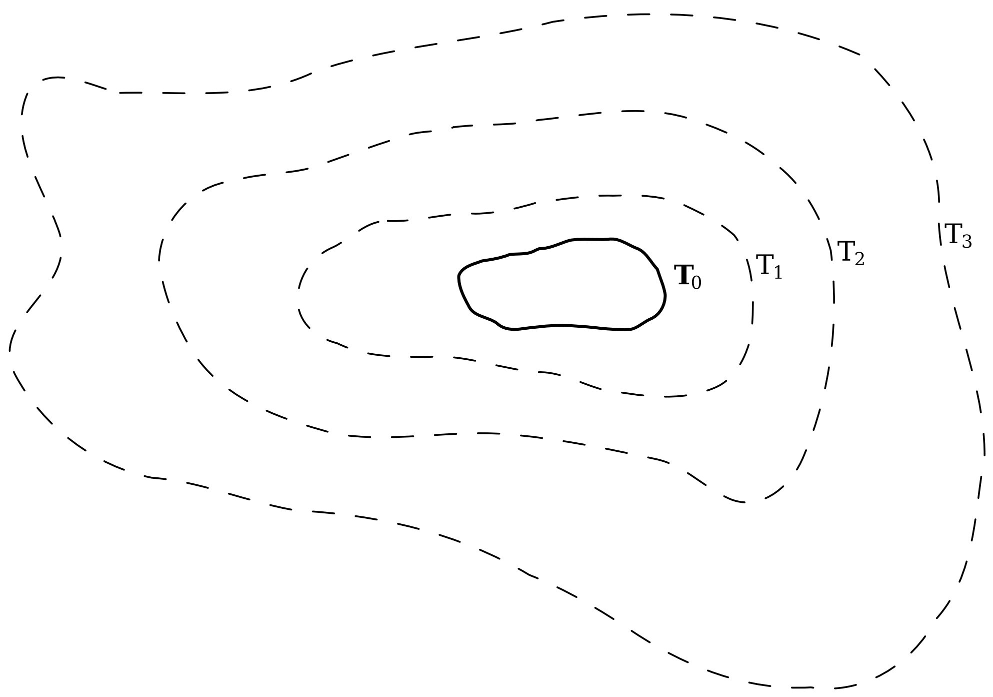


Figure 2.3: An example wavefront propagating over a surface. T_0 is the initial location of the wavefront. The other curves represent the progress of the wavefront at time step T_1, T_2, etc.

would be generated by the motion of the front over the surface of the data. By giving the wavefront an appropriate velocity through the data, one can produce a wavefront whose arrival times at each data point correspond to some solution at that point. The propagation of a wavefront to create an arrival surface can be seen in Figure 2.3. The boundary value formulation states that the front must always propagate outwards, and can never backtrack. This means that information must be propagated "one way," in the upwind direction, from smaller values of T to larger values of T. This causal relationship between consecutive points in the upwind direction is exploited to create a Dijkstra-like algorithm.

We now need to be able to describe the motion of the wavefront through time. Unfortunately, this motion may not be well defined for all parts of the front, as the curve that represents the front may not be differentiable everywhere.

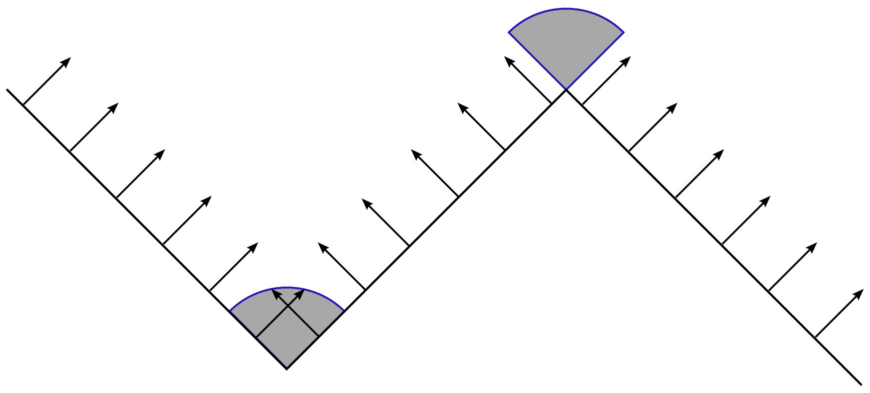


Figure 2.4: A shock is shown on the left. If every point on the wavefront is moved forward by its velocity for the next time step, it would run into itself. A rarefaction is shown on the right. If every point on the wavefront is moved forward by its velocity for the next time step, it would pull itself apart.

Sethian's solution is to use upwind finite difference equations to find "viscosity solutions" that conform to hyperbolic conservation laws. These "weak solutions" approximate the motion of the front well and can handle shocks and rarefactions. Shocks and rarefactions are described in Figure 2.4. We refer the reader to [5] for a more thorough overview of these concepts.

At a high level, the fast marching algorithm is, indeed, very Dijkstra-like. The algorithm given below is for a two-dimensional grid, like our DEM data; though Sethian also describes how it can be modified for a triangle mesh [3].

1. With start point p within a grid cell, compute the distance from p to the four vertices of the cell and label those vertices with their cost. Add those vertices to the set F. Set F makes up a fringe of trial vertices. The cost labels of these vertices are still subject to change.

2. Add all other vertices to set O and label them with cost ∞. Set O contains vertices that have not been seen yet.

3. Loop until all vertices are in set I. Vertices in set I are considered final, and their cost labels will not change.

    (a) Pick a vertex, v from set F with the smallest cost label. Move v from F to I.

    (b) For each neighbor of v in O or F, update its cost label using the upwind gradient approximation, but only if this cost is less than its current cost label.

    (c) For each neighbor of v in O, move v from O to F.

The real key to making the fast marching algorithm work well is the gradient approximation used in step 3b. Sethian proposes the following upwind gradient approximation.

$$|\nabla T| \approx (max(D^{-x}T, -D^{+x}T, 0)^2 + max(D^{-y}T, -D^{+y}T, 0)^2)^{\frac{1}{2}} = F \qquad (2.9)$$

Where D^{-x}, D^{+x}, D^{-y}, and D^{+y} are forwards and backwards operators representing the change in T in a specific grid direction.

For a point being updated, we calculate its label by solving the above approximation for each of its neighbors in set I, and also each pair of its neighbors in set I, and then picking the smallest solution. When using fast marching to find discrete geodesics, Equation 2.9 simplifies greatly. When trying a pair of neighbor points from set I, we just solve for T_u in the quadratic Equation 2.10. When trying a single neighbor from set I, we solve for T_u in the degenerate quadratic Equation 2.11.

$$(T_u - T_x)^2 + (T_u - T_y)^2 = 1 \qquad (2.10)$$

$$(T_u - T_n)^2 = 1 \qquad (2.11)$$

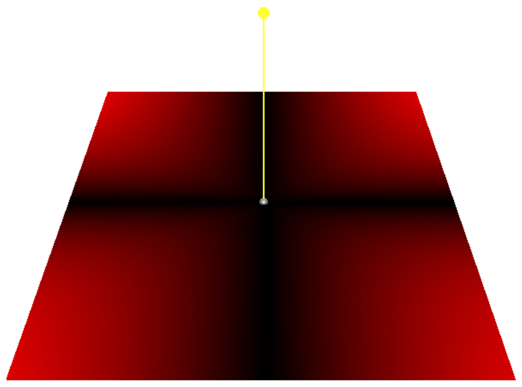


Figure 2.5: Error when running fast marching using Sethian's gradient approximation from Equation 2.9. Traveling through a point in the X direction has a cost of 1, while traveling through the same point in the Y direction has a cost of 2. Black is 0 error, and full red is an error of 35. For this test, the maximum error was 34.4, and the average error was 32.3.

We chose the fast marching algorithm for our work because, one, it is not much more difficult to understand and implement than Dijkstra's shortest path algorithm. Also, due to its use of a priority queue in step 3a, it has a similar running time to Dijkstra's algorithm — $O(NlogN)$. Yet, it can attain a better accuracy than Dijkstra's algorithm, as geodesics are not constrained to be on mesh edges. It should also be noted that there are implementations, such as [21], that improve running time or space requirements; although one could find similar modifications to Dijkstra's algorithm. The accuracy of the fast marching algorithm is dependent on the accuracy of the finite difference equation used to approximate the gradient. One could use Equation 2.9 from above, or use a higher order approximation as Sethian also suggest in [5]. The authors of [22] describe a cost update procedure that is more accurate when the initial boundary is a single point.

Modifying the fast marching algorithm to produce discrete energetic geodesics then becomes a matter of picking an appropriate gradient approximation method.

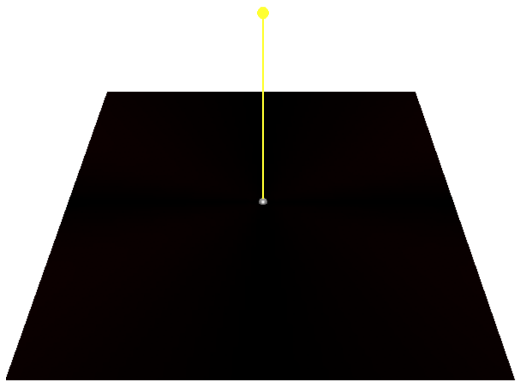


Figure 2.6: Error when running fast marching using the update step from [23], Equations 2.12 and 2.13. This is otherwise the same test as in Figure 2.5 above. The maximum error for this test was 1.7, and the average error was 1.2.

Unfortunately, we cannot use Equation 2.9, or any of the other previously mentioned gradient approximations, because they assume that ∇T is isotropic. When working with energetic costs, $\nabla T(p)$ is dependent on the slope at point p, and the slope through a point is dependent on the direction of travel, making ∇T anisotropic. Tsitsiklis acknowledges this deficiency when working with anisotropic costs in his own work [20]. The error that comes from running fast marching on an anisotropic cost field using a typical gradient approximation method can be seen in Figure 2.5.

Luckily, this problem has already been addressed. The authors of [23] also needed to run fast marching on an anisotropic cost field. They use the following finite difference approximation for updating a point's cost. If updating from a pair of neighbor points in set I, solve for T_u in the quadratic Equation 2.12. If updating from a single neighbor in set I, solve for T_u in the degenerate quadratic Equation 2.13.

$$(\frac{T_u - T_x}{C_{x\to u}})^2 + (\frac{T_u - T_y}{C_{y\to u}})^2 = 1 \tag{2.12}$$

$$(\frac{T_u - T_n}{C_{n\rightarrow u}})^2 = 1 \tag{2.13}$$

In these equations, $C_{n\rightarrow u}$ is the cost of traversing from the neighbor point n to the point we are updating. The authors of [23] claim that this gradient approximation correctly handles the anisotropy in the cost field given the discretization. Our own test, seen in Figure 2.6, shows that this update step does handle the anisotropy well. We therefore use this finite difference approximation in our own fast marching update step.

2.3 Domain Decomposition Parallelization for Fast Marching

As will be described in later sections, our implementation of fast marching is rather slow for large multiple-gigabyte data sets, even taking most of a day to run on the large Oregon data set that we will use to evaluate our tools. This drove us to develop a faster fast marching algorithm that is based on the domain decomposition fast marching algorithm presented in [24]. Our work towards this algorithm will be described in Section 3.2, and this section will provide an overview of the domain decomposition method that it is based on.

The goal for the author of [24] was to develop various ways in which the fast marching algorithm, described above in Section 2.2.3, could be adapted to run on and utilize a parallel architecture. The fast marching algorithm is not an inherently parallelizable algorithm due to step 3a, which relies on the picking of a *global* minimum point at each iteration.

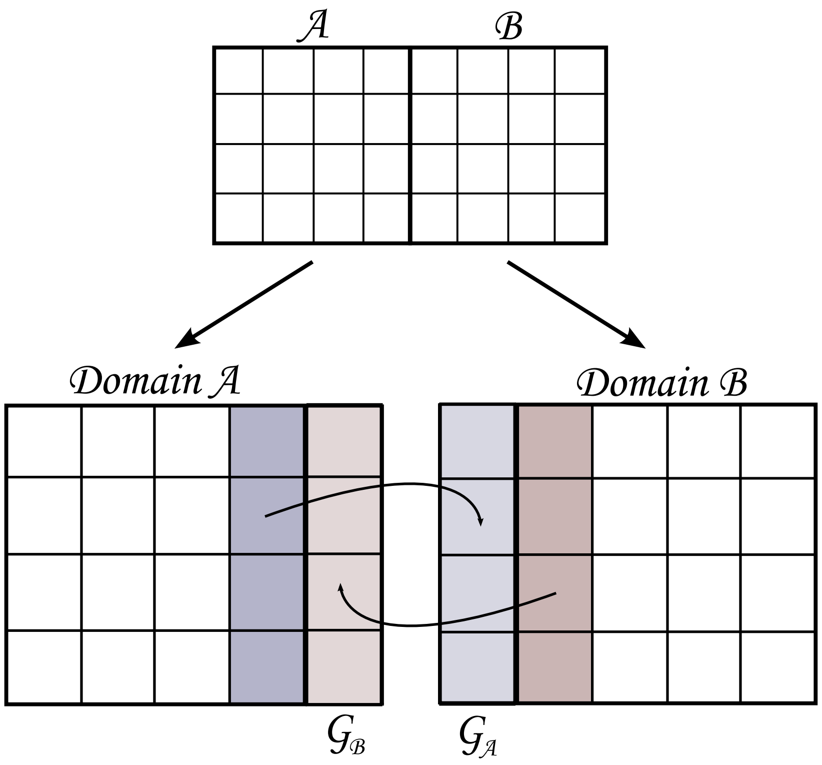


Figure 2.7: The layout of the "ghost nodes" used for synchronization of parallelized fast marching in domains *A* and *B*.

The solution developed by the author of [24] is to break the full data set into multiple smaller domains. Different threads of execution then run the fast marching algorithm within each domain, picking the *local* minimum point when reaching step 3a. This works fine for some of the points if these local minimum points have the same value that they would have had if they were selected as a global minimum point. However, we must address the dependencies between domains, as there are points that should be affected by information propagating across domain borders. This is done by requiring threads to synchronize access to shared "ghost points" that exist on the borders between domains. Figure 2.7 shows the layout of domains with bordering ghost points. If a change is made to a ghost point by one thread, it must notify the neighboring thread. That thread then removes any points from set I, the final set, that have a larger cost than the ghost point, and it moves fast marching back to that ghost point. This rollback procedure only needs to be done with points that have larger cost values because, as was discussed earlier, information only propagates from smaller points to larger points (in the upwind direction). The modified fast marching algorithm that is run by each thread within a domain is given below.

1. Perform steps 1 and 2 from the standard fast marching algorithm.

2. Loop until all vertices are in set I.

    (a) Check the ghost points along the borders that are currently in set I. If any ghost point g has moved to set I in another domain and now has a smaller cost label, rollback fast marching by moving all points in set I with cost label greater than or equal to the cost label of g back to set F.

   (b) Pick a vertex, v from set F with the locally smallest cost label. Move v from F to I.

   (c) For each neighbor of v in O or F, update its cost label using the upwind gradient approximation, but only if this cost is less than its current cost label.

   (d) For each neighbor of v in O, move v from O to F.

   (e) If v is a ghost point for another domain, communicate the change to the thread handling that domain.

3. Wait for all domains to complete or a ghost point g to be given a smaller cost label and moved to set I in another domain. If a g is modified, perform the rollback from step 2a and restart the algorithm at step 2b.

Chapter 3

Implementation

Our solution is implemented as a number of simple tools, i.e. a tool for viewing the data, a tool for running fast marching, a tool for creating paths, etc. We wanted our tools to be able to handle multiple gigabytes of digital elevation data by keeping only the data we need in memory and swapping the rest to disk. Rather than trying to write our own code for managing large amounts of varying data, we went with an existing database solution — the Berkeley DB [25]. The Berkeley DB is very simple, robust, and customizable. It is not a relational database. Rather, it provides a set of APIs for doing lower-level database work such as organizing arbitrary chunks of bytes into hashes, b-trees, and lists. It handles caching, sorting, storing and retrieving your data, and it is a linked-in library, which makes it fast. It also has more advanced features such as transaction support, locking support for multithreaded applications, and recovery support via logs. These more advanced features were not needed in this project, though, so they were turned off.

3.1 Continuous Energetic Path Tools

Our first tool, called cepmakedb, creates the databases used by other tools and populates them with data from ArcInfo ASCII DEM files. Large DEMs typically come broken up into numerous smaller DEMs, so the input to this program is a text file with the file names of the smaller DEMs arranged in a grid as the DEMs themselves should be arranged when stitched back together. As the tool processes the DEM data it collects the data together and writes it to the databases in squares of nearest data points. This takes some advantage of the spatial locality inherent in the data, so that a database page is likely to contain points that are near to each other. This approach is better than writing the data to the database in raster order, which would only take advantage of locality in one dimension.

The next tool, called cepviewer, displays the visualizations of the DEM data created by other tools. The default visualization, created by cepmakedb, is of the elevation data. The colors in this visualization correspond to elevation. These visualizations are all top-down orthographic projections of the data. The user can switch between the available visualizations by pressing the space bar. The user can also zoom in on areas of the data using their mouse. Left clicking in a single spot creates and moves start and end points used by other tools when performing fast marching and creating paths. If a path has been created, it is also displayed by cepviewer. The cepviewer can be seen in Figures 3.1 and 3.2.

The cepcolor tool can be used for resetting the elevation visualization. It can also be used to switch between the natural and rainbow color gradients seen in Figures 3.1 and 3.2. The rainbow gradient, though less pleasant on the eyes, is able to show more detail.

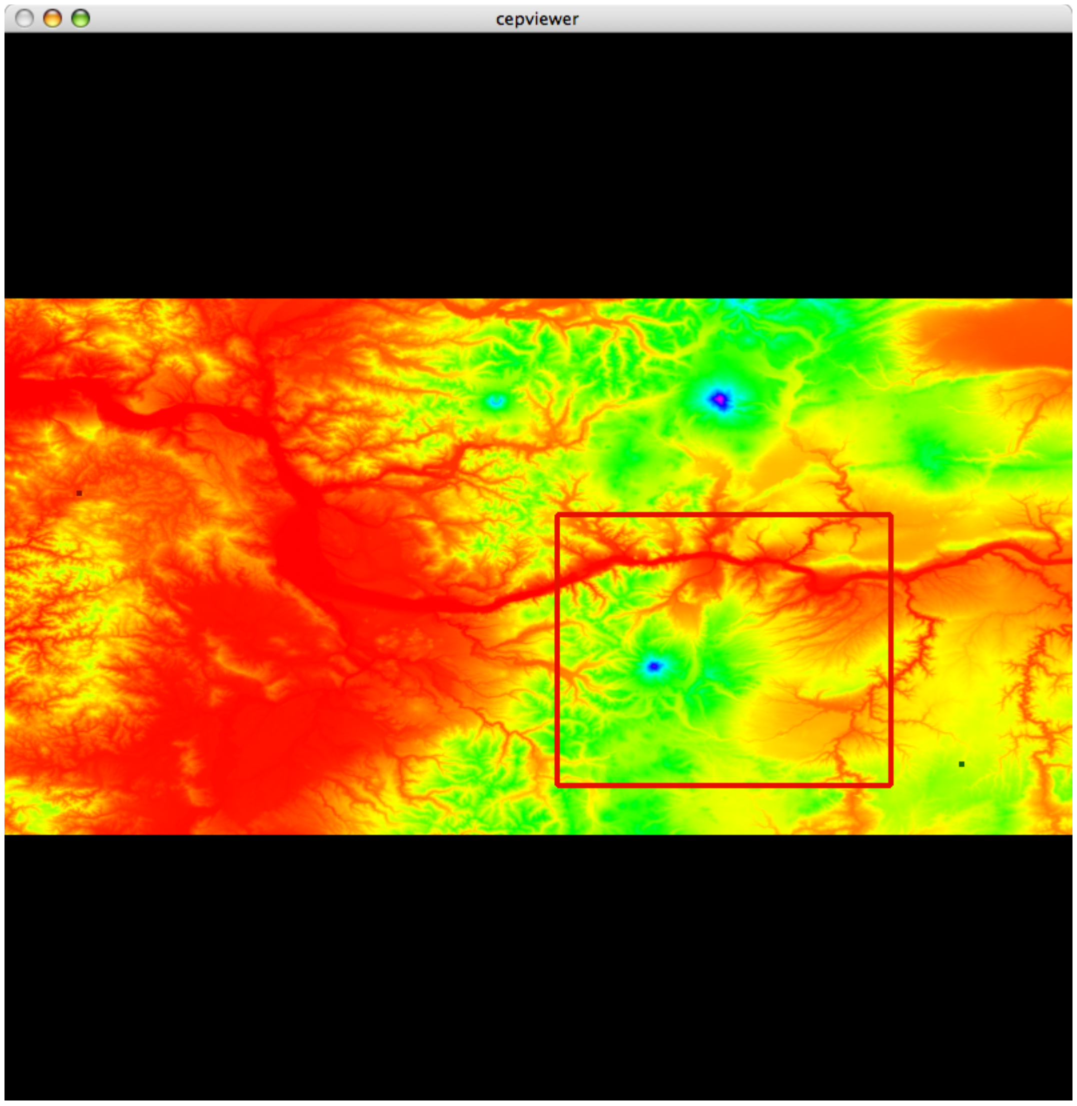


Figure 3.1: A screenshot of the **cepviewer** application. It is currently displaying elevations using a rainbow gradient. Red corresponds to low elevations. As elevations increase, colors change from red to yellow to green to blue to violet. This image also shows the outline of a region selected for zooming.

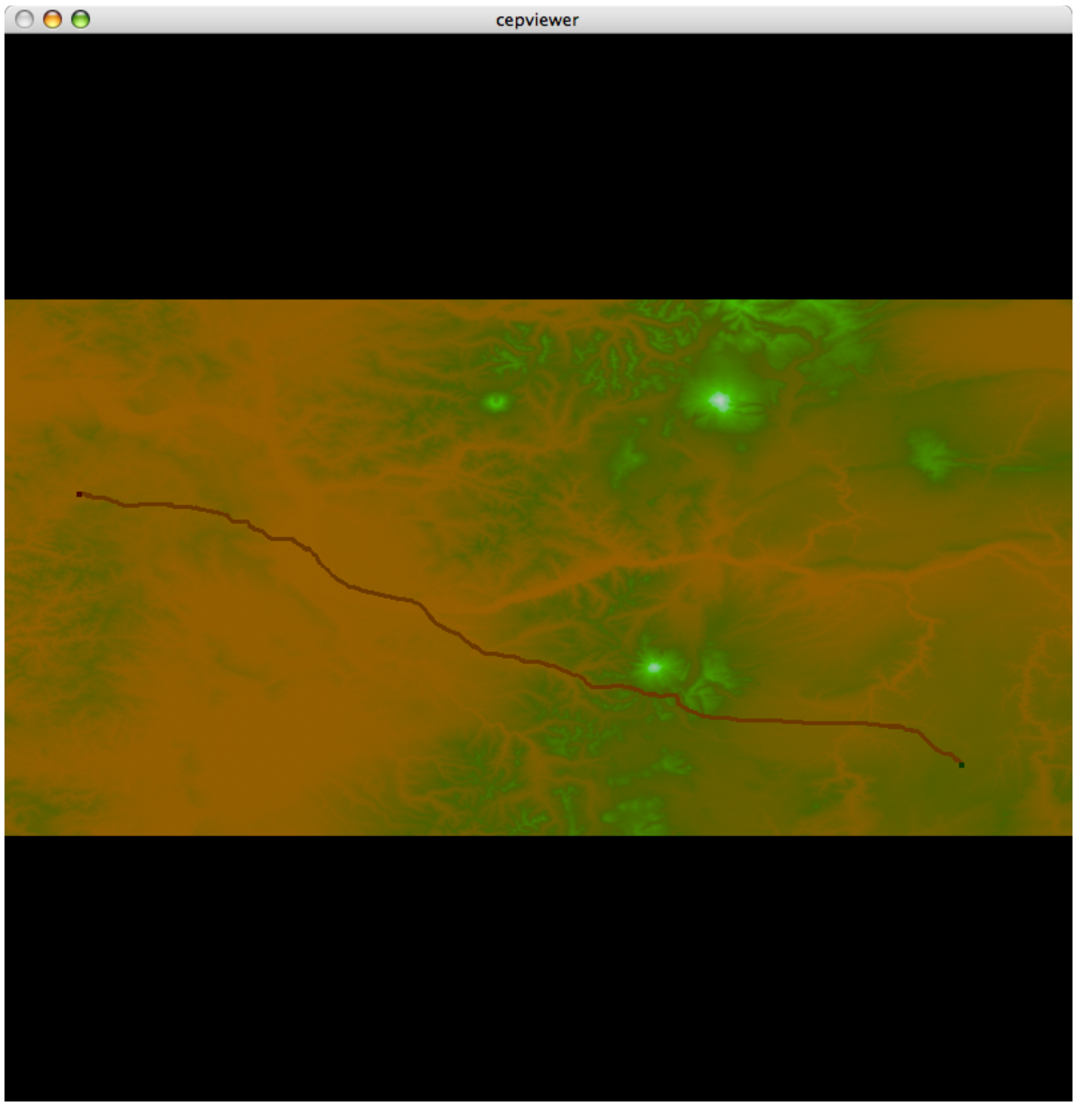


Figure 3.2: Another screenshot of the cepviewer application. This time it is displaying elevations with a more natural color scheme, with brown for lowlands, green for higher terrain, and white for very high peaks. A path is also being displayed from the start point on the right to the end point on the left.

Energetic analysis of the DEM data is performed using the cepfm tool. It checks for the presence of a starting point, set using the cepviewer application, and performs fast marching out from it. Fast marching is performed as described in Section 2.2.3, and the finite difference approximation to the gradient is used from [23], as is described in Equations 2.12 and 2.13. Caloric cost is calculated as it was in Energetic Analyst [1, 2], which is described in Section 2.1, Equations 2.1 through 2.7. Once fast marching has completed, cepfm creates two caloric visualizations. One represents caloric cost with isocontours — curves of the same color that are the same caloric cost from the start point. The other uses a color gradient, currently the rainbow, to represent a more continuous caloric difficulty terrain from the start point. These visualizations can be seen in Figures 3.3 and 3.4. There are actually two versions of the cepfm tool. One performs fast marching in a standard way, and the other uses a modified domain decomposition method to run faster. Section 3.2 contains more information regarding these two tools.

The ceppath tool uses the start and end points set by cepviewer and the caloric data created by cepfm to create a least-caloric path between the two points that can be displayed by the viewer application. The correct way to do this would be to follow the gradient of T from the end point back to the start point. This can be done by solving the differential equation seen in Equation 3.1 [3].

$$\frac{dX(s)}{ds} = -\nabla T \tag{3.1}$$

The path is then traced by $X(s)$. However, this approach for finding the geodesic is not trivial. Therefore, we opted for a simpler solution. We create a path by adding the end point to the path, picking its lowest cost neighbor, adding it to the path, and proceeding in that manner until we reach the start point. As pointed out by the authors of [26], this is guaranteed to be correct since T is

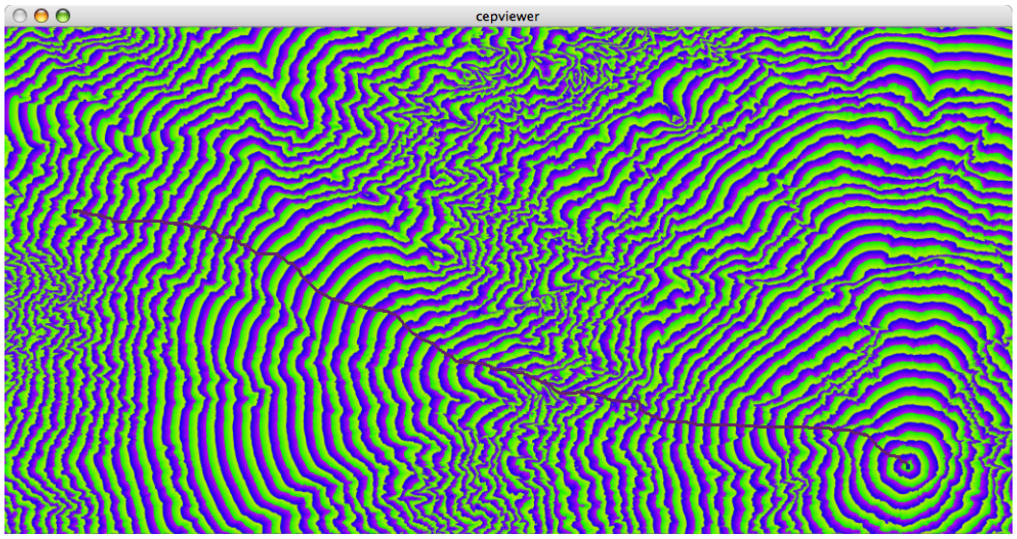


Figure 3.3: A screenshot of the **cepviewer** application displaying caloric isocontours as generated by **cepfm**. Curves of the same color are the same caloric cost from the start point.

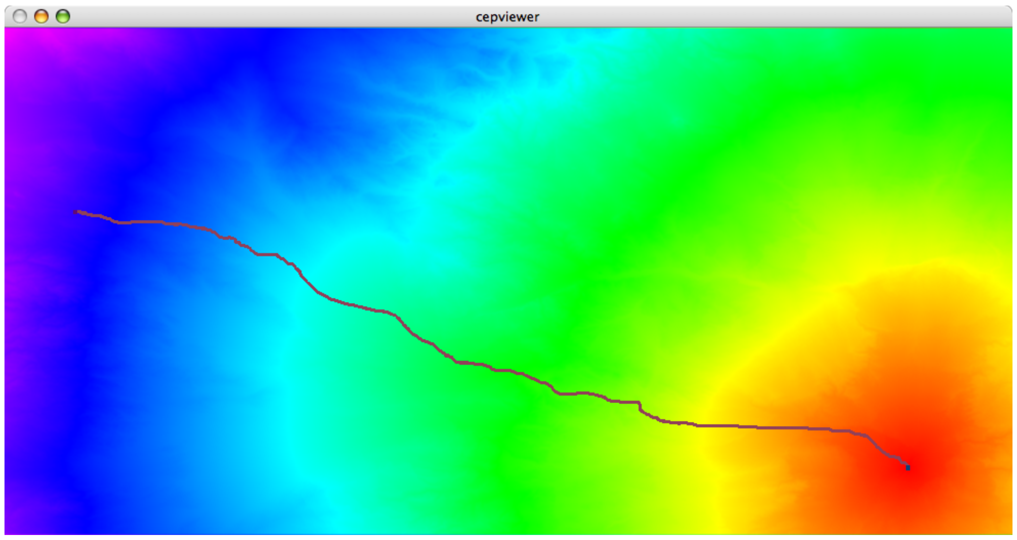


Figure 3.4: A screenshot of the **cepviewer** application displaying the caloric gradient as generated by **cepfm**. The gradient is currently set to be the rainbow, with red at low caloric cost and violet at high caloric cost.

defined increasingly from our start point. Future work might modify ceppath to create paths by following the gradient in order to get a truer and less jagged representation of the path.

3.2 A Domain Decomposition-Based Algorithm for Fast Marching

Two versions of the cepfm tool were created. The first one does fast marching in the normal way as described earlier. However, this tool could take multiple hours to run on a very large DEM data set, even taking most of a day to run on the large Oregon data set that will be used later for evaluating this work. There are two likely factors contributing to this slowness. One is that we are incurring a non-trivial amount of overhead when using the Berkeley DB to get and retrieve single elements of elevation and calorie costs data. The second is that, while running fast marching, we have numerous database cache misses and often have to access the disk to get the required database pages. Figure 3.5 demonstrates how we can get a cache miss with a very large fringe.

We wrote a second version of cepfm to address these issues. The algorithm that it uses is based off of the domain decomposition method for parallelized fast marching described by Herrmann in [24], which we reviewed in Section 2.3. In our version of the algorithm, all of the data required to run the algorithm are broken up into "chunks." Herrmann calls these domains. The fast marching algorithm is run within one chunk at a time, and the whole chunk of data resides in memory, making access to it fast. When we are done with one chunk, it is written back to the database and the next chunk is loaded. Reading and writing large chunks

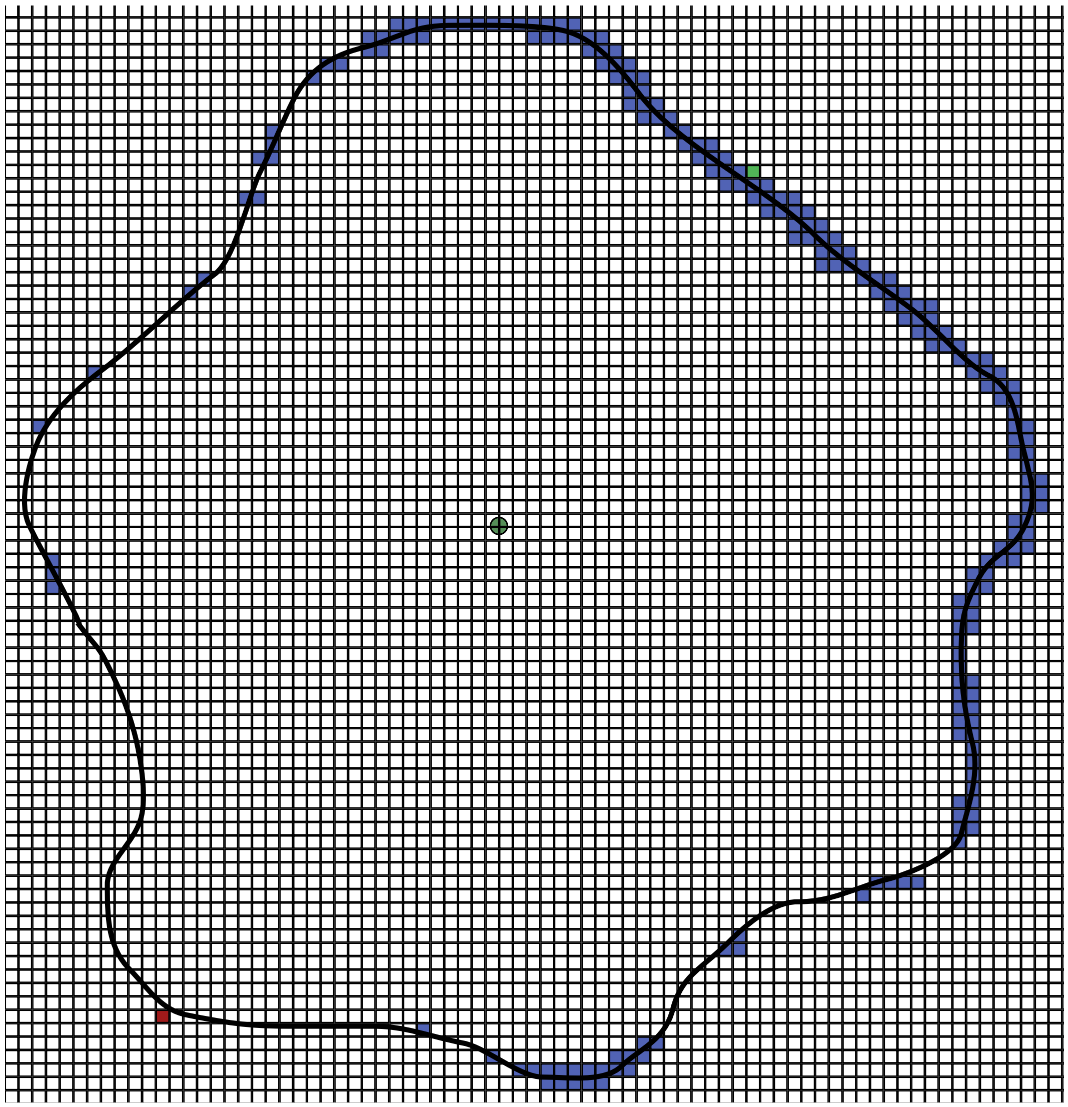


Figure 3.5: An illustration of how a very large fringe can cause a database cache miss during fast marching. Each grid square represents a collection of data points in the same database page. Blue pages are in the cache. The algorithm just finished working with data in the green page. The red page is needed for the next fast marching step, but is not in the cache.

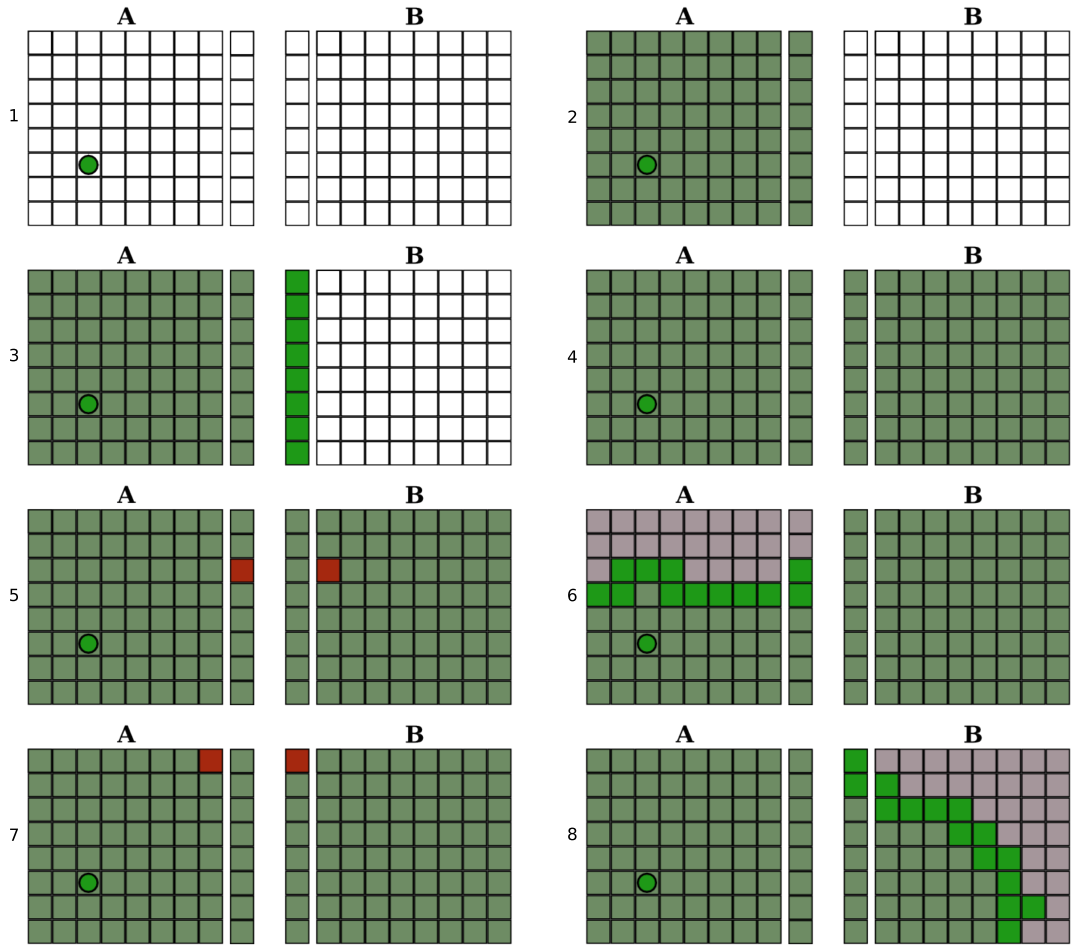


Figure 3.6: A demonstration of domain decomposition-based fast marching on a DEM divided into two chunks. (2) Fast marching is run within the chunk containing the start point, A. (3) An initial fringe is created in the bordering chunk, B, using the border values from chunk A. (4) Fast marching is run within chunk B. (5) Border points in B are compared with ghost points in A, and a value is found that is smaller by more than the threshold amount. (6) Values larger than the modified ghost value are rolled back in chunk A. Fast marching is rerun within chunk A using the modified ghost value. (7) Border points in A are compared with ghost points in B, and a value is found that is smaller by more than the threshold amount. (8) Values larger than the modified ghost value are rolled back in chunk B. Fast marching is rerun within chunk B using the modified ghost value.

of data like this should hide some of the overhead of using the Berkeley DB, as we make fewer accesses to it. Keeping the entire chunk of data in memory avoids the issue of cache misses.

Our modification to the domain decomposition algorithm from [24] is intended for a single thread of execution, so synchronization at the border nodes is not necessary. However, we do employ the use of ghost nodes at the borders between chunks as we still need to catch and handle dependencies between chunks. Our algorithm went through a number of iterations before we settled on the version that we present results for in Section 4.1. The first version of the algorithm was simple and very similar to the domain decomposition algorithm of [24].

Figure 3.6 provides a visual overview of how this initial algorithm works. It starts by running fast marching over all points within the chunk containing the start point. We then run fast marching over all points in each chunk outward from the starting chunk. Each successive chunk creates its initial fringe set, F, from the ghost nodes on borders with completed chunks. After fast marching a single chunk, we run a correction step based on the rollback step from [24]. The details of this correction step, with the rest of our initial algorithm, are provided below.

1. Load the chunk containing the start point. Run the standard fast marching algorithm, as was described in Section 2.2.3, within this chunk.

2. Add the neighboring chunks to a chunk priority queue, with the cost determined by the cost of their smallest ghost node. Loop until there are no more chunks in the priority queue.

   (a) Load the chunk from the front of the chunk priority queue.

(b) For each ghost point g on the border with a completed chunk, add g to set F.

(c) Run the standard fast marching algorithm starting at the loop — step 3 from Section 2.2.3.

(d) Compare each border point b with its associated ghost point g in the neighboring chunks. If any b's cost label is less than g's cost label by more than **THRESHOLD** amount, add the neighboring chunk to a correction priority queue, with its value determined by the cost of the b. If the chunk is already in the correction priority queue, update its cost if b has a smaller cost.

(e) Run the correction algorithm below.

(f) Add any bordering chunks that haven't been seen before to the chunk priority queue. If necessary, update the cost of bordering chunks that are already in the chunk priority queue.

The variable in the above algorithm, **THRESHOLD**, is a tunable value than can significantly affect both the running time and the correctness of the algorithm. The higher the threshold value, the less likely it is that the correction algorithm will be run, which should affect the correctness of the result. The lower the threshold value, the more likely it is that the correction algorithm will be run, which will increase the running time of the algorithm. The correction algorithm itself is described below.

1. Remove a completed chunk C from the correction priority queue.

   (a) Pick the changed ghost node with the smallest cost label, s. Update its value from the associated border point in the bordering chunk B.

(b) For each point or ghost p in C with cost label less than or equal to the cost label of s, move p from set I to set F.

(c) For each point or ghost p in C with cost label greater than the cost label of s, move p from set I to set O.

(d) Run the standard fast marching algorithm within chunk C starting at the loop — step 3 from Section 2.2.3. Whenever the algorithm checks the value of a point that is on the border with B, check to see if B has a smaller value. If so, copy the smaller value to the border point and use it.

(e) Compare each border point b in chunk C with its associated ghost point g in the neighboring chunks. If any b's cost label is less than g's cost label by more than **THRESHOLD** amount, add the neighboring chunk to a correction priority queue, with its value determined by the cost of the b. If the chunk is already in the correction priority queue, update its cost if b has a smaller cost.

Steps 1b and 1c are effectively the rollback step from [24]. This correction algorithm runs until all completed chunks agree on their borders by at least **THRESHOLD** amount. Also, it is important to note that the correction algorithm tries to run "smaller" corrections first. This is to better conform to the upwind nature of the fast marching algorithm. Making corrections arbitrarily can adversely affect the correctness of the algorithm.

Unfortunately, early tests showed less than ideal results from this algorithm. In fact, we often found that tuning **THRESHOLD** to a lower value would actually increase the amount of error in our results. This was counterintuitive as tuning it down leads to more corrections, which should lead to less error.

Our hypothesis for why we are getting so much error in domain decomposition-based fast marching is that the fast marching method is highly reliant on the correct upwind propagation of information for correctness. Correct upwind propagation is typically enforced by step 3a of the fast marching algorithm — picking a globally smallest point. Domain decomposition fast marching picks locally smallest points in each domain. Synchronization of access to the ghost nodes on the borders, as well as the concurrent running of fast marching in each domain, prevents any one domain from making too much progress based on an incorrect smallest point. Small corrections can then be made when inconsistencies are detected at the borders between domains. However, in our single processor version of the algorithm, fast marching is allowed to make as much progress as possible during both the initial run and in subsequent corrections. Hence, we are propagating out a great deal of incorrect caloric data. Some of this cost data seems to even be a *smaller* value than it should be, which can allow it to be picked up by the correction algorithm at borders and propagated into neighboring chunks. This would explain why much of the error that we saw was negative — meaning that the caloric cost was less than it should have been.

To test our hypothesis we wrote a version of the domain decomposition-based algorithm that found caloric costs using Dijkstra's shortest path algorithm instead of the fast marching method. Dijkstra's shortest path algorithm is less complex than the fast marching algorithm, and the way information propagates is much simpler. A point's value is only dependent on the value of its smallest neighbor. If any of its neighbors change values, it is a simple task to make a correction. This change then propagates out over the data in a straightforward way. Our tests of the domain decomposition-based Dijkstra's algorithm gave us the expected results. Lowering the value of **THRESHOLD** lowers the error. Also, the error is bounded by the value of **THRESHOLD**.

The simplified problem with our domain decomposition-based fast marching algorithm, then, seemed to be that fast marching was making too much progress from incorrect data. Our solution is to limit how far information can propagate within a chunk. We do this by using another tunable value, **PROPAGATE**, which is the "distance" in caloric cost that fast marching is allowed to make during the initial run and subsequent corrections in a chunk. The updated domain decomposition-based fast marching algorithm is shown below.

1. Load the chunk containing the start point. Run the standard fast marching algorithm, as was described in Section 2.2.3, within this chunk. Terminate fast marching when it accepts a point with a cost label greater than or equal to **PROPAGATE**.

2. Add the chunk to a chunk priority queue, with its cost determined by its smallest fringe value. If fast marching reached any border points, add the associated neighboring chunks to the chunk priority queue, with the cost determined by the cost of their smallest ghost node.

   (a) Load the chunk from the front of the chunk priority queue.

   (b) If this is the first time that the chunk has been seen, then for each ghost point g on the border with a completed chunk, add g to set F. This creates its initial fringe. It should otherwise have an existing fringe from its last run.

   (c) Run the standard fast marching algorithm starting at the loop — step 3 from Section 2.2.3. Terminate fast marching when it accepts a point that is more than **PROPAGATE** calories greater than the first point accepted in this run.

(d) Compare each border point b in set I with its associated ghost point g in the neighboring chunks. If any b's cost label is less than g's cost label by more than **THRESHOLD** amount, add the neighboring chunk to a correction priority queue, with its value determined by the cost of the b. If the chunk is already in the correction priority queue, update its cost if b has a smaller cost.

(e) If fast marching didn't finish in the chunk, then add the chunk back to the chunk priority queue, with its cost determined by its smallest fringe value. If fast marching reached border points on borders with chunks that haven't been seen yet, add the associated neighboring chunks to the chunk priority queue, with the cost determined by the cost of their smallest ghost node.

The updated correction algorithm doesn't change much. It only needs to limit the propagation of a correction. It is described below.

1. Remove a completed chunk C from the correction priority queue.

   (a) Pick the changed ghost node with the smallest cost label, s. Update its value from the associated border point in the bordering chunk B.

   (b) For each point or ghost p in C with cost label less than or equal to the cost label of s, move p from set I to set F.

   (c) For each point or ghost p in C with cost label greater than the cost label of s, move p from set I to set O.

(d) Run the standard fast marching algorithm within chunk C starting at the loop — step 3 from Section 2.2.3. Whenever the algorithm checks the value of a point that is on the border with B, check to see if B has a smaller value. If so, copy the smaller value to the border point and use it. Terminate fast marching when it accepts a point that is more than **PROPAGATE** calories greater than the first point accepted in this run.

(e) Compare each border point b in chunk C with its associated ghost point g in the neighboring chunks. If any b's cost label is less than g's cost label by more than **THRESHOLD** amount, add the neighboring chunk to a correction priority queue, with its value determined by the cost of the b. If the chunk is already in the correction priority queue, update its cost if b has a smaller cost.

This algorithm does produce significantly less error than its predecessor. Tuning **PROPAGATE** down will decrease the error as expected. Also, tuning **THRESHOLD** down when **PROPAGATE** is low will decrease error. Unfortunately, the error is still not bound by **THRESHOLD**.

Chapter 4

Results

The previous work by Wood presented results and arguments as to the effectiveness of caloric cost over other metrics, such as straightest path [1]. Therefore, we will primarily focus on presenting an evaluation of our domain decomposition-based fast marching algorithm presented in Section 3.2. We will also use our tools to analyze a large portion of digital elevation data from Oregon in order to evaluate the effectiveness of our tools. The results of this evaluation will be used to make proposals for future work.

4.1 Error and Performance Results for Domain Decomposition-Based Fast Marching

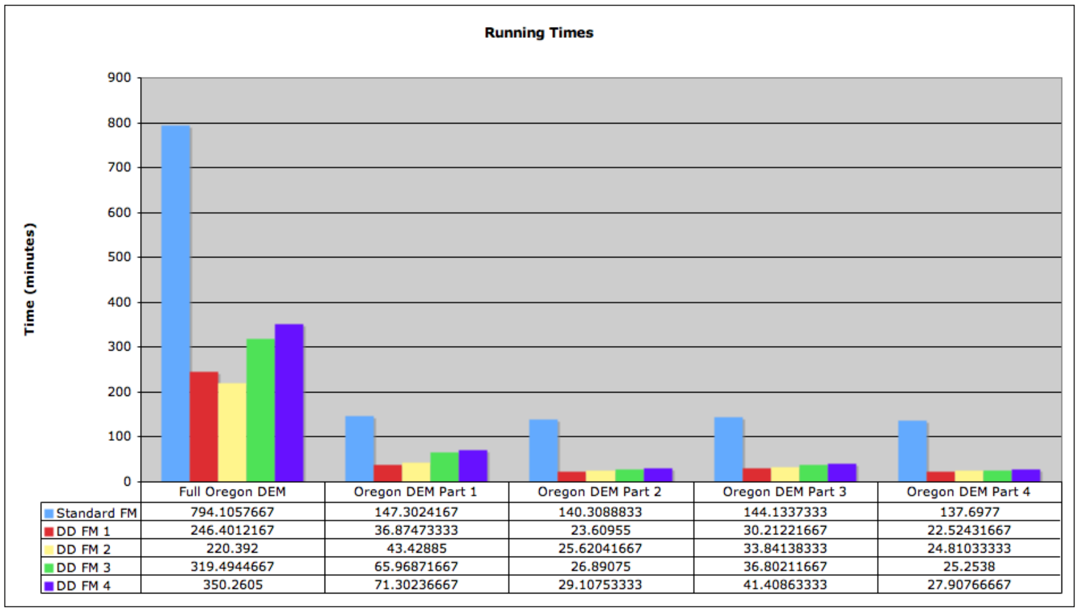
Our goal for the modified domain decomposition fast marching tool was to be able to run fast marching over very large data sets in less time than our standard fast marching tool. This would hopefully be at a minimal cost to the accuracy of the results. We performed our tests on five different data sources.

- The **Full Oregon DEM** data set is the large DEM used in Section 4.2.
- **Oregon DEM Part 1** though **Oregon DEM Part 4** are each a quarter of the **Full Oregon DEM** data set.
  - **Oregon DEM Part 1** and **Oregon DEM Part 2** both use the same quarter of the data, but have different starting points.
  - **Oregon DEM Part 3** and **Oregon DEM Part 4** use different portions of the data and have different starting points.

On each data set we performed standard fast marching and four different runs of the domain decomposition-based fast marching.

- **DD FM 1** uses a **PROPAGATE** value of 500 calories and a **THRESHOLD** value of 5 calories.
- **DD FM 2** uses a **PROPAGATE** value of 500 calories and a **THRESHOLD** value of 0.5 calories.
- **DD FM 3** uses a **PROPAGATE** value of 200 calories and a **THRESHOLD** value of 5 calories.
- **DD FM 4** uses a **PROPAGATE** value of 200 calories and a **THRESHOLD** value of 0.5 calories.

All of these tests were performed on a 2.33 GHz Intel Core 2 Duo machine with 2 GB of 667 MHz DDR2 SDRAM. The database cache was set to a size of 512 MB. For domain decomposition, chunk sizes were set to a maximum of 1024 x 1024 data points. For the timing tests we report wall clock time.



	Full Oregon DEM	Oregon DEM Part 1	Oregon DEM Part 2	Oregon DEM Part 3	Oregon DEM Part 4
🟦 Standard FM	794.1057667	147.3024167	140.3088833	144.1337333	137.6977
🟥 DD FM 1	246.4012167	36.87473333	23.60955	30.21221667	22.52431667
🟨 DD FM 2	220.392	43.42885	25.62041667	33.84138333	24.81033333
🟩 DD FM 3	319.4944667	65.96871667	26.89075	36.80211667	25.2538
🟪 DD FM 4	350.2605	71.30236667	29.10753333	41.40863333	27.90766667

Figure 4.1: A chart of the timing results for each test. Below it is a table of the actual timing values. DD FM tests take about a third of the time of standard fast marching.

Figure 4.1 shows the running times for each data set and each fast marching run. The domain decomposition-based fast marching method takes about a third of the time to run when compared to the standard fast marching method for all of our tests. Also, the time that domain decomposition-based fast marching takes typically increases as **PROPAGATE** and **THRESHOLD** are tuned down. This is expected because tuning **THRESHOLD** down causes more corrections to be made, and tuning **PROPAGATE** down limits fast marching in each chunk, which in turn will cause more jumping around between chunks. Moving between chunks more often means more loads and stores of chunk data from and to the database.

With the variable values that we chose, we typically saw error that was less than 200 calories in all of our tests, with the majority of points having negligible error. Interestingly, this was true of the **Full Oregon DEM** as well as the

	Full DEM	DEM Part 1	DEM Part 2	DEM Part 3	DEM Part 4
DD FM 1					
Ave Error	4.657903	0.667706	1.037725	0.653417	0.138821
Max Error	190.174835	148.558502	182.657288	148.489273	127.015396
DD FM 2					
Ave Error	1.072422	0.220335	10.422381	0.216483	0.232800
Max Error	254.070190	114.325211	155.879028	214.480530	76.809616
DD FM 3					
Ave Error	2.624293	0.620319	2.111316	0.581078	0.190194
Max Error	129.605209	63.288265	73.330414	71.396088	62.605042
DD FM 4					
Ave Error	2.477251	0.065483	0.278318	0.321646	0.078015
Max Error	90.212952	99.823235	76.997459	109.255867	48.214836

Table 4.1: A table of the average and maximum error of each run of the domain decomposition-based fast marching algorithm on each data set. Averages and maximums are calculated using the absolute value of the errors, so that negative and positive errors are treated alike.

DEM parts, even though the **Full Oregon DEM** contained four times as many points. To put these results into perspective, paths of any interesting length in out test data are typically tens of thousands of calories long. Table 4.1 shows the maximum and average caloric error for each domain decomposition-based fast marching run when compared to the results of the standard fast marching method. Table 4.2 shows the distribution of error for each domain decomposition-based fast marching run as averaged over the data sets. These results show that a large majority of data points are only off by 0 to 5 calories when using domain decomposition-based fast marching. For a more complete set of error results for domain decomposition-based fast marching, please visit Appendix A. Finally, Figure 4.2 plots the error distribution from Table 4.2.

The error that we see from our updated domain decomposition-based fast marching algorithm is acceptable. What's more, the timing results show us that

	DD FM 1	DD FM 2	DD FM 3	DD FM 4
$[-25,-20)$	1.856×10^{-5}	0	0	5.949×10^{-8}
$[-20,-15)$	3.51×10^{-5}	2.381×10^{-5}	1.0113×10^{-6}	5.017×10^{-6}
$[-15,-10)$	2.239×10^{-5}	5.849×10^{-5}	5.5125×10^{-6}	0.000119881
$[-10,-5)$	4.588×10^{-5}	0.000377471	0.000789052	0.000400839
$[-5,0)$	11.18941863	16.65672127	6.174735998	11.4792977
$[0,5)$	79.35249341	71.55092589	87.84330668	82.75579442
$[5,10)$	6.046916197	0.882269741	3.335486835	4.154831922
$[10,15)$	1.493236341	0.558730123	2.235106031	1.479255248
$[15,20)$	0.447463753	8.243355836	0.36090583	0.09208019
$[20,25)$	0.427769657	1.202950647	0.044258657	0.008795539
$[25,30)$	0.749078264	0.490183319	0.002067759	0.004038844
$[30,35)$	0.184073969	0.306134008	0.001283625	0.002098869
$[35,40)$	0.041366121	0.029478125	0.000994654	0.020019196
$[40,45)$	0.01255795	0.034328968	0.000386499	0.002224071
$[45,50)$	0.005119227	0.009055742	0.000221208	0.000530247
$[50,55)$	0.006156722	0.006173034	0.000135113	0.000153631
$[55,60)$	0.010385611	0.014223209	9.956×10^{-5}	9.871×10^{-5}
$[60,65)$	0.0053265	0.002824148	8.175×10^{-5}	7.753×10^{-5}
$[65,70)$	0.001951592	0.002315669	7.376×10^{-5}	5.847×10^{-5}
$[70,75)$	0.021723551	0.002948758	1.644×10^{-5}	4.29×10^{-5}
$[75,80)$	0.000865403	0.001810657	1.071×10^{-5}	3.214×10^{-5}
$[80,85)$	0.000730965	0.00095987	9.102×10^{-6}	2.072×10^{-5}
$[85,90)$	0.000694364	0.000797001	6.13×10^{-6}	1.239×10^{-5}
$[90,95)$	0.000519493	0.000612814	4.164×10^{-6}	7.277×10^{-6}
$[95,100)$	0.000531474	0.000519997	3.212×10^{-6}	3.331×10^{-6}
$[100,105)$	0.000305995	0.000508456	2.737×10^{-6}	2.776×10^{-7}
$[105,110)$	0.00026023	0.000380782	2.558×10^{-6}	5.552×10^{-7}
$[110,115)$	0.000202846	0.000344536	1.785×10^{-6}	0
$[115,120)$	0.000160571	0.000212083	2.082×10^{-6}	0
$[120,125)$	0.000131978	0.00012815	1.071×10^{-6}	0
$[125,130)$	8.772×10^{-5}	9.518×10^{-5}	4.759×10^{-7}	0
$[130,135)$	7.065×10^{-5}	7.241×10^{-5}	0	0

Table 4.2: A partial table of the data visualized in Figure 4.2. The left column lists caloric error ranges. The other four columns list the percentages of points that fall into those error ranges. This data is averaged over the data sets. For a more complete set of error results, please see Appendix A.

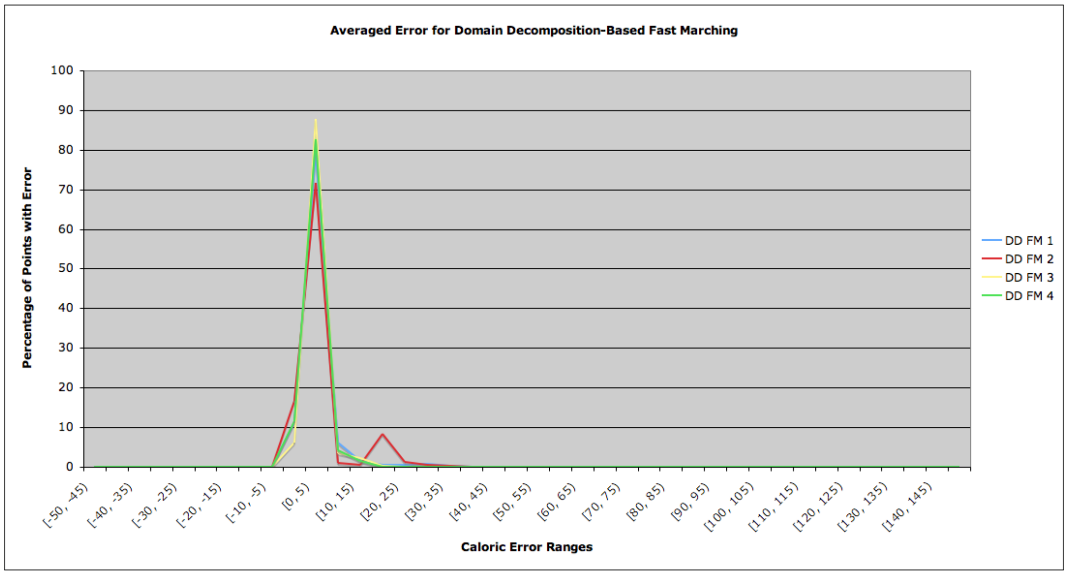


Figure 4.2: A graph of the percentage of points in the data sets that fall into the given caloric error ranges. This data is averaged from all data sets. This makes it clear that the majority of the points are only 0 to 5 calories off from the correct value. This is a visualization of the data in Table 4.2.

we still have leeway to tune the values of **PROPAGATE** and **THRESHOLD** even further down and get even better accuracy. Finally, Figure 4.3 shows a visualization of the error for a single test run. All other visualizations show a similar pattern. Error in these images can be seen as streaks that move outward from the start point along the upwind direction, showing how information was propagated in the data.

Finally, we should note that least caloric paths generated from our data were often the same or similar enough as to be indistinguishable by the naked eye. Few paths in the domain decomposition-based fast marching data differed visually from the same path in the normal fast marching data, and even these were only off by a small amount. Figure 4.4 and 4.5 illustrate the similarity between a path generated from standard fast marching data and a path with the same

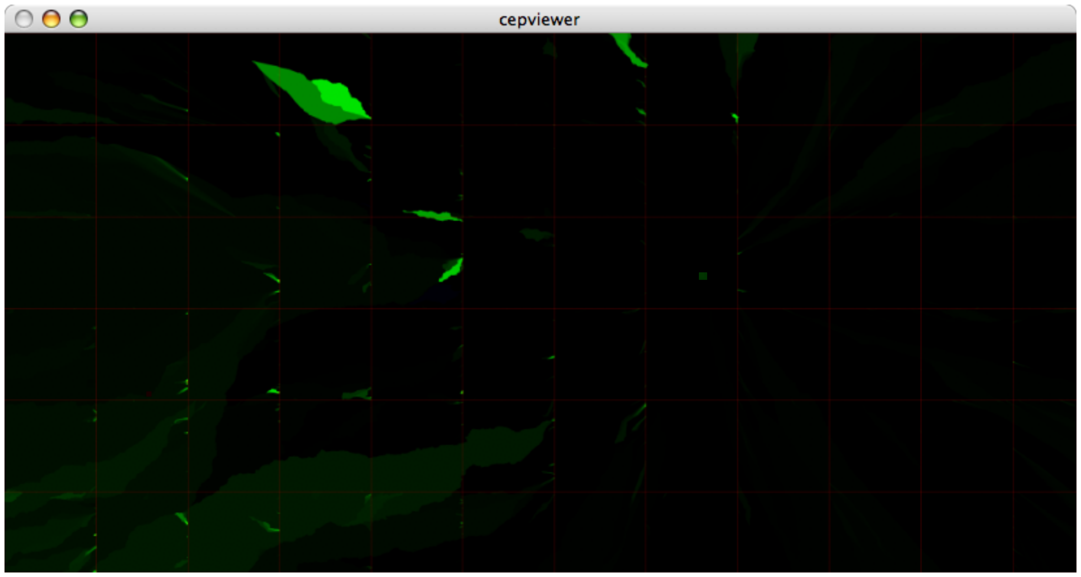


Figure 4.3: A visualization of the propagation of caloric error. Lighter green indicates more error than darker green. Notice that the error propagates outward in “streaks” from the start point, following the propagation of information in the data.

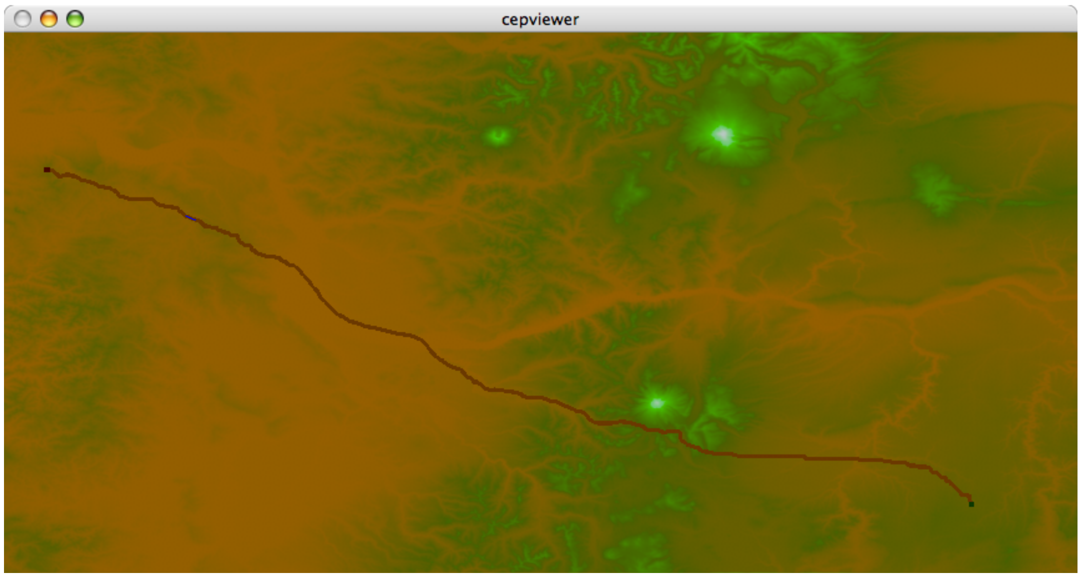


Figure 4.4: An energetically optimal trail found using the data produced by the standard fast marching method.

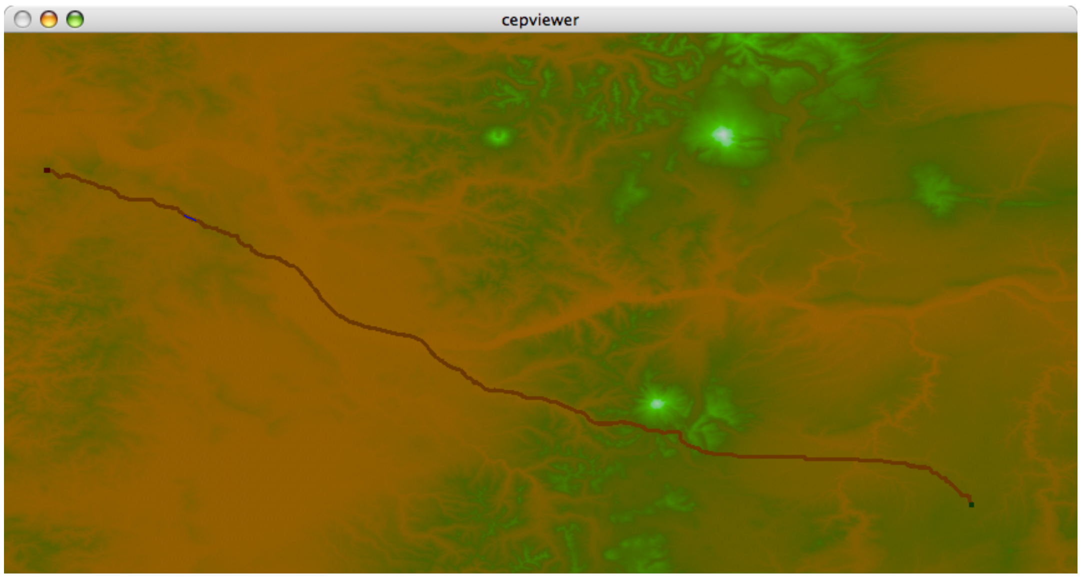


Figure 4.5: An energetically optimal trail found using the data produced by the domain decomposition-based fast marching method. This image uses the same DEM and the same start and end points as in Figure 4.4. The trails are so similar that we have to highlight the section of the trails that are not the same to make it more noticeable.

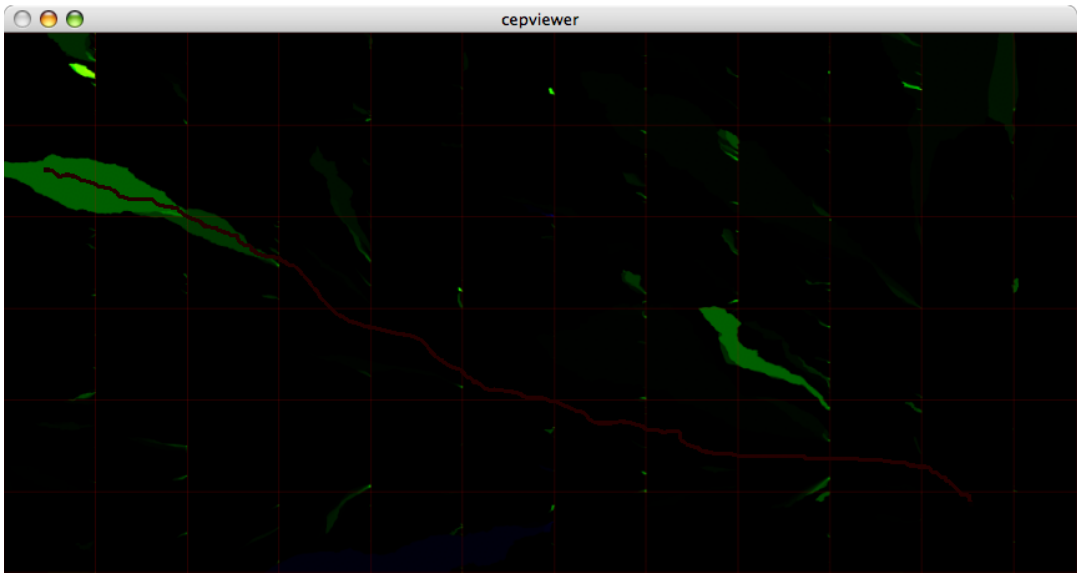


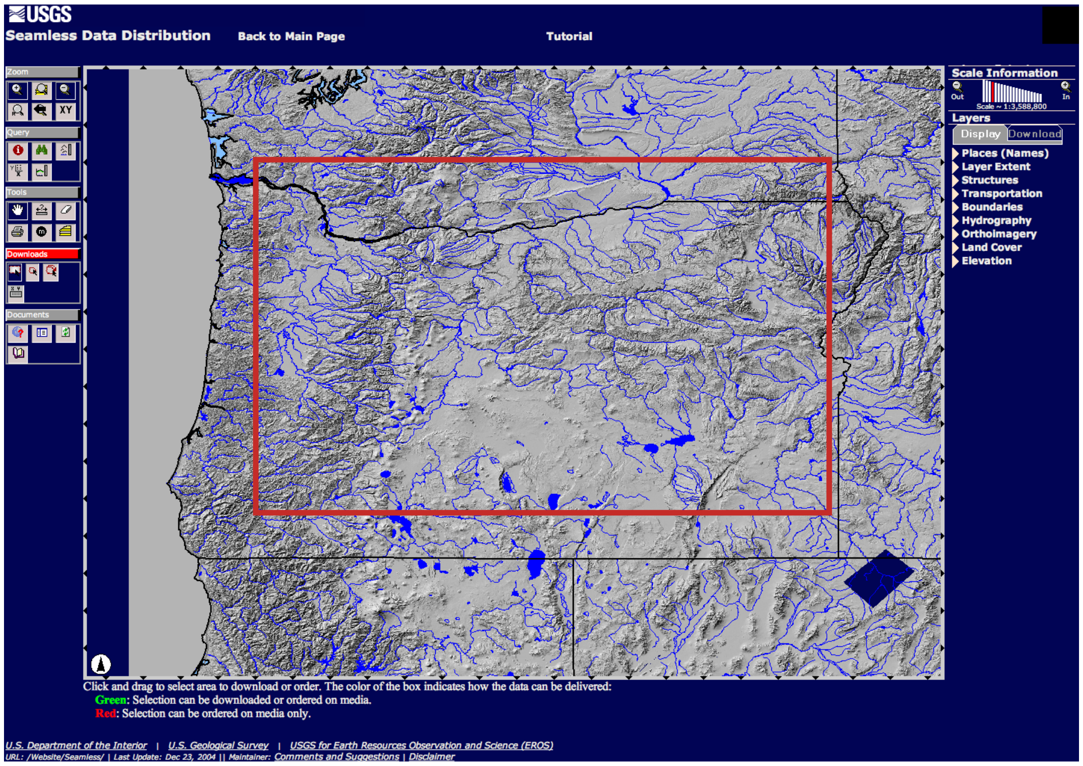
Figure 4.6: The same trail and data as from Figure 4.5, but showing a visualization of the error in the caloric cost data. The section where this trail differs from the trail in Figure 4.4 lies in an area of high error.

start and end point generated from domain decomposition-based fast marching data. To generate these figures, we had to find a path that would traverse an area of high error, which can be seen in Figure 4.6. The length of the trail in the standard fast marching data is 23,459.66 calories. The length in the domain decomposition-based fast marching data is 23,469.04 calories. Differences in least caloric paths should become even less noticeable as the values of **PROPAGATE** and **THRESHOLD** are tuned down.

4.2 Analysis of Oregon DEM

To evaluate the effectiveness of our tools, we downloaded a large portion of digital elevation data, mostly covering Oregon, from the USGS Seamless Data Distribution website [27]. A screenshot of this website showing the approximate extents of our digital elevation data is shown in Figure 4.7. This data came as fourteen smaller DEMs which, when stitched back together, contained 23951 x 14037 data points at a 30 meter resolution. The particular region of Oregon was chosen with the hopes that we could compare a lowest energetic cost path with the Oregon Trail [28], the historical trail used by Western pioneers in the 19th century.

Figures 4.8 through 4.10 show multiple views of the digital elevation data, caloric costs, and the least caloric path given a start point and end point that approximate the start and end points of the Oregon Trail in the state of Oregon. The caloric length of this trail is 44,006.45 calories. Overlaid on these images is an approximate tracing of the real Oregon Trail, including a common cutoff, terminating at Oregon City [29]. The figures show that both routes travel through the central-Northern part of the state and avoid mountain ranges. The energeti-



Click and drag to select area to download or order. The color of the box indicates how the data can be delivered:
Green: Selection can be downloaded or ordered on media.
Red: Selection can be ordered on media only.

U.S. Department of the Interior | U.S. Geological Survey | USGS for Earth Resources Observation and Science (EROS)
URL: /Website/Seamless/ | Last Update: Dec 23, 2004 || Maintainer: Comments and Suggestions | Disclaimer

Figure 4.7: A screenshot of the USGS Seamless Data Distribution website used to download our large Oregon data set. The red outline shows the approximate extents of this data.

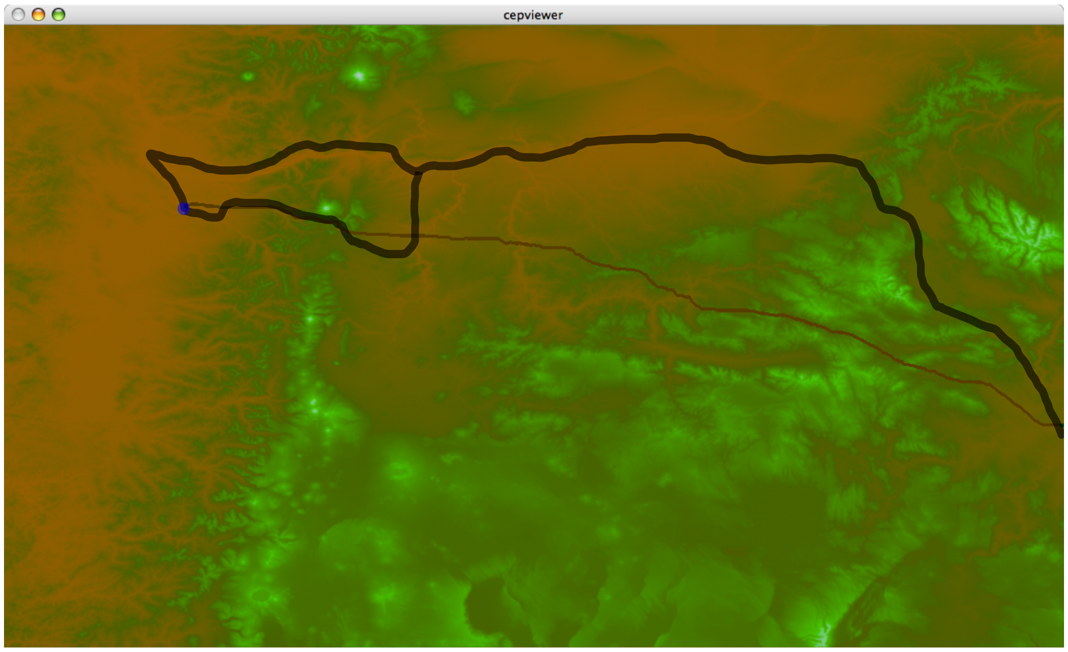


Figure 4.8: The elevation view of our Oregon data showing the least caloric path between the start and end points of the Oregon Trail in Oregon. The trail to the North is an approximate tracing of the real Oregon Trail [29].

cally optimal path does vary from the historical route at the first portion of the trail. In Figures 4.11 and 4.12 we calculate an energetically optimal trail from the start point to a way-point on the Oregon Trail, and from the way-point to the end point. These figures show a much stronger correlation with the historical trail. The caloric length of the path through the way-point is approximately 45,870 calories — less than 2,000 calories longer than the energetically optimal trail. These examples do much to demonstrate the validity of our algorithm and methods. However, there is a disparity between the paths, and we would like to suggest a number of possible factors contributing to it and offer recommendations for future changes to address these issues.

First, the Oregon Trail is not necessarily going to be a lowest human caloric path. The trail was commonly traveled by wagon, so a better metric might have

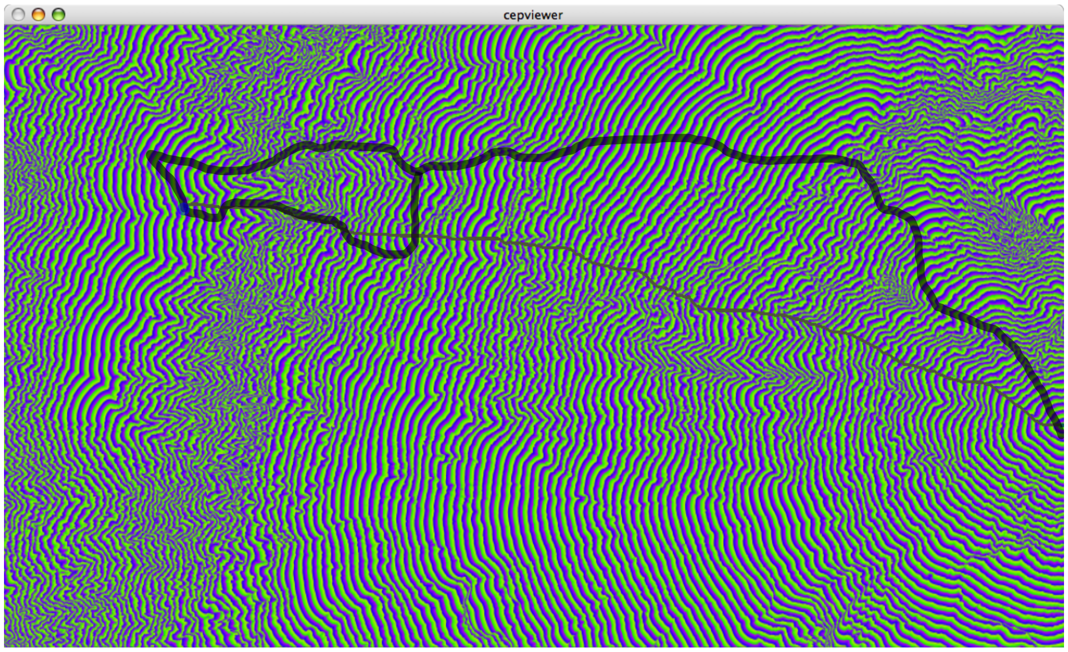


Figure 4.9: The energetic isocontour view of our Oregon data. Otherwise the same as Figure 4.8.

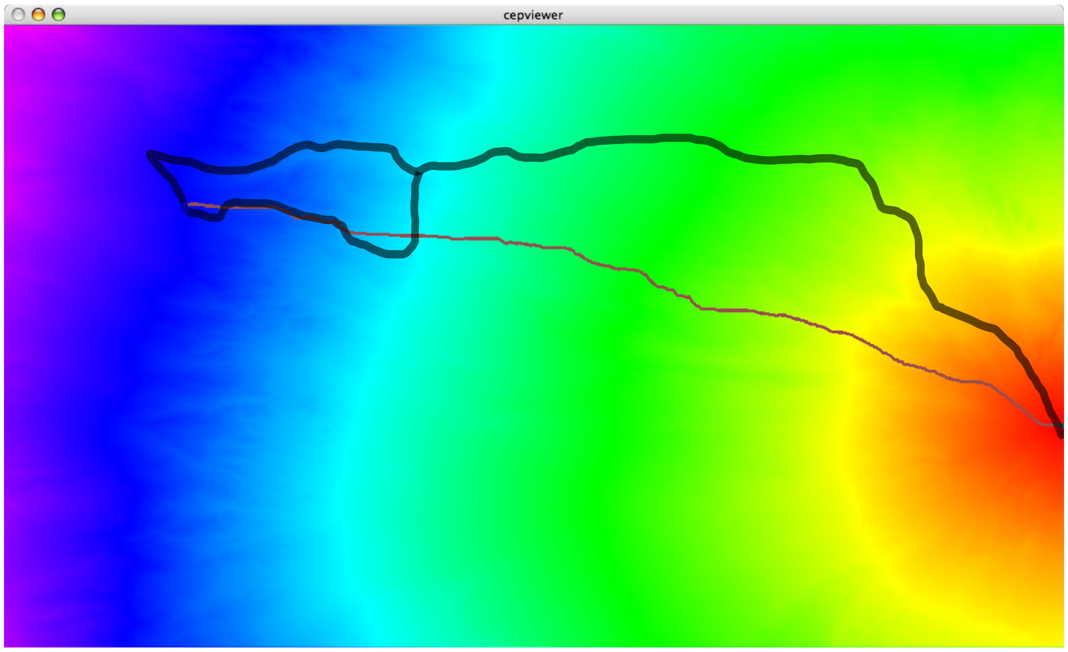


Figure 4.10: The energetic gradient view of our Oregon data. Otherwise the same as Figure 4.8.

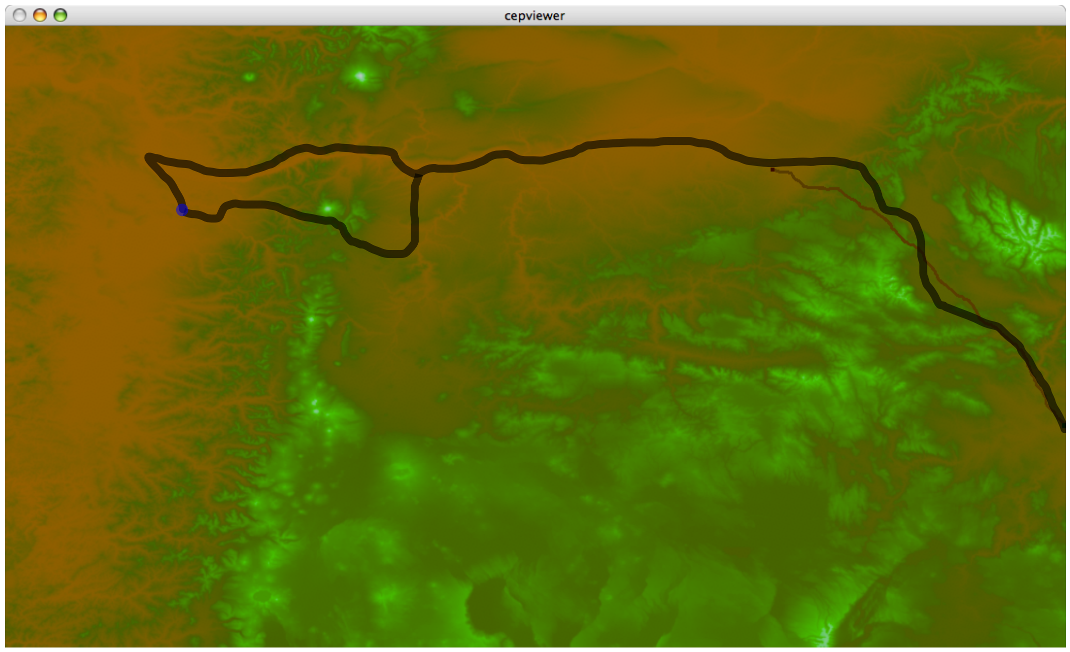


Figure 4.11: The energetically optimal path from the start point to a nearby point on the Oregon Trail. Overlaid on this image is an approximate tracing of the real Oregon Trail [29].

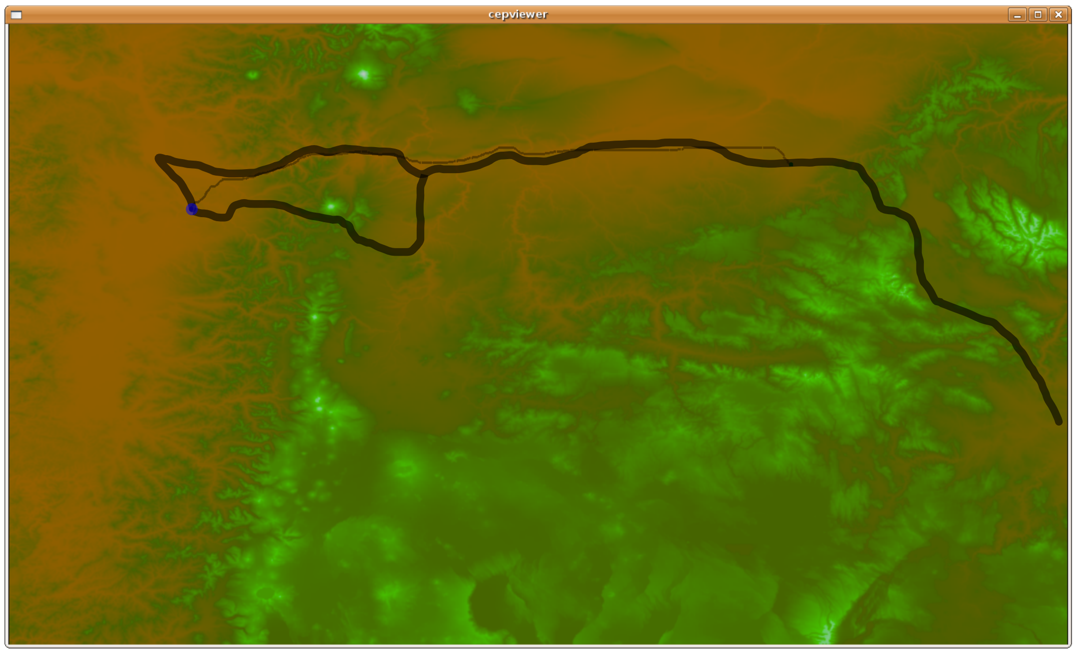


Figure 4.12: The energetically optimal path from a point near to the actual Oregon Trail to the end point of the trail. This comparison is similar to that of Figure 4.11.

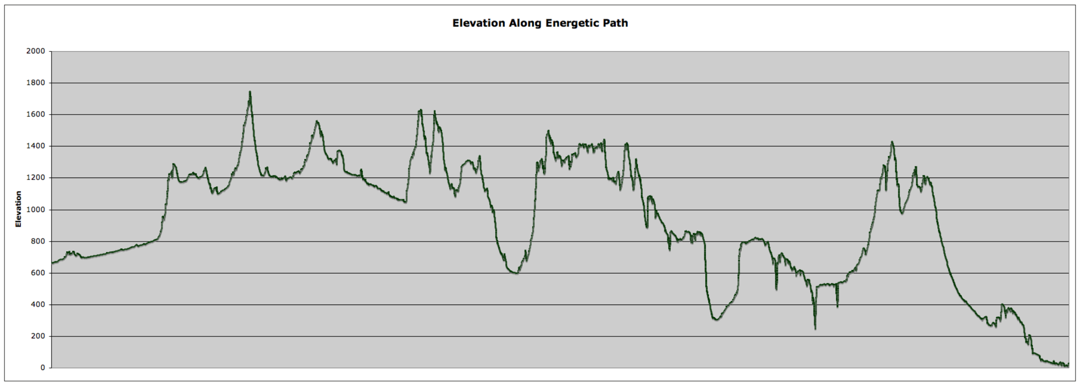


Figure 4.13: A graph of the elevations, in meters, along the energetically optimal path shown in Figures 4.8 through 4.10.

something to do with how easy it would be to navigate with a wagon, such as flatness. Figure 4.13 shows a graph of the elevations along the least caloric path shown in Figures 4.8 through 4.10. The maximum elevation along this path is 1,749.08 meters, and the minimum is 9.02 meters. The path starts at 665.37 meters and ends at 27.10 meters. The maximum and minimum elevations in the entire digital elevation model are 3,746.36 meters and -0.95 meters, respectively. We can see that the least caloric path is not necessarily going to be very flat, but may involve many elevation changes. Figures 4.8 through 4.12 show that energetically optimal paths are more direct than the historical trail. They travel through elevation changes that the real Oregon Trail avoids. A future version of the analysis tool might give the user a choice of a multitude of metrics, such as energetic cost, flatness, or straightness, allowing them to mix and match varying amounts of these metrics.

Second, the caloric cost is currently being calculated everywhere using the terrain factor for a treadmill (see Equations 2.1 through 2.4). A more accurate version of the tool might read in terrain types — such as rocky, grassy, and wetland — from some other GIS data source, and use that data to set appropriate terrain factors where possible.

Another form of impedance that is currently ignored is the presence of streams, rivers, and bodies of water. As we can see in Figure 4.7, this form of GIS data is readily available. One approach that a future version of the analysis tool might take would be to read in GIS data for bodies of water and use that information to set the terrain factor artificially high for those points. This would restrict the crossing of streams, rivers, and bodies of water. However, this is not exactly what we want. We might want to discourage the crossing of bodies of water with a moderate increase in the terrain factor, but what we are really trying to stop is traveling of rivers and large lakes as if they were lowest calorie shortcuts. A smarter approach would be to use an anisotropic terrain factor where the magnitude of the terrain factor in a given direction is proportional to the distance one would have to travel in that direction in order to reach land again. An anisotropic terrain factor would not affect our algorithm, as it already expects anisotropy from the slope of the terrain.

Finally, there are a handful of other impedance factors that are currently ignored, such as temperature, the presence of eatable flora and fauna, and the presence of snow. These factors may or may not have GIS data that could be used, and the translation to terrain factor may not be easily defined. Therefore, it might be worthwhile to give the user the ability to hand-edit the terrain factor data, or possibly some other impedance factor data, to give the user the ability to make certain relevant areas harder or easier to traverse. It would not be hard to modify the cepviewer application in its current form to give it a terrain factor "painting" mode, where the user would use his mouse to paint higher or lower terrain factors onto the terrain data.

Chapter 5

Conclusions and Future Work

The work presented here is another step toward what could be a powerful set of tools for archaeologists, anthropologists, and historians. It further develops and improves upon prior work for the analysis of digital elevation data using a caloric cost metric. Our tools can handle multiple gigabytes of digital elevation data and can create more accurate discrete energetic geodesics through the use of the fast marching method. We have also attempted to address performance issues by developing a modified domain decomposition-based fast marching algorithm that is able to run in significantly less time at an acceptable cost to accuracy.

Tools providing human-centered metrics for analyzing nearness on a terrain, where these metrics might include energetically optimal travel as well as other metrics such as straightness or flatness, could provide for intriguing research in early human travel and better concepts of geographic nearness. It is important that these tools be able to handle realistically large amounts of data; particularly if, in the future, these tools must analyze more than just digital elevation data. Our tools work towards this goal by using a database library to help handle gigabytes of data. Future revisions of these tools might use a custom data library that can take better advantage of spatial locality and could do some form of prefetching[1].

Our goal of supporting larger quantities of data also drove us to use a more accurate algorithm for finding caloric costs — fast marching. We developed a domain decomposition-based fast marching algorithm to try to address the performance aspect of working with large quantities of data. Being that this algorithm is based off of work for parallelized fast marching [24], future work might see an even faster *parallelized* version of the cepfm tool. This parallelization would have to be across multiple machines, or some other non-shared memory solution, as it is currently memory that is our bottleneck. The multi-core shared memory machine that we performed our tests on, for example, would not benefit from parallelization across multiple threads, as each thread would be fighting the other for use of memory. Also, a goal that we never reached for the domain decomposition-based fast marching algorithm was guaranteed error bounds. However, each improvement of the algorithm delivered more accurate results. Further research and work on this algorithm could see a version of the algorithm that allows one to guarantee error bounds.

Finally, we initially attempted to develop a viewer application capable of displaying our data in a three dimensional perspective view. Developing this application quickly became more work than we had time for, so we scrapped it and went to a top-down orthographic view. A future viewing application might come back to the initial Google Earth-like concept [30], as it would have been a more interesting, intuitive, and fun way to view the data.

[1]There are a number of places in our tools where we simply iterate over all of the data, performing some operation on it. A simple prefetching mechanism would boost performance in these places. We find it doubtful that the Berkeley DB uses any such prefetching mechanisms.

Bibliography

[1] B. Wood, "Energetic analyst: Software for the visualization and analysis of pedestrian travel over three dimensional terrain," Master's thesis, California Polytechnic State University San Luis Obispo, 2004.

[2] B. Wood and Z. Wood, "Energetically optimal travel across terrain: visualizations and a new metric of geographic distance with archaeological applications," in *Proceedings of SPIE Electronic Imaging*, vol. 6060, 2006.

[3] R. Kimmel and J. A. Sethian, "Computing geodesic paths on manifolds," *Proceedings of the National Academy of Science*, vol. 95, pp. 8431–8435, Jul. 1998.

[4] J. A. Sethian, "Fast marching methods," *SIAM Review*, vol. 41, no. 2, pp. 199–235, 1999. [Online]. Available: http://link.aip.org/link/?SIR/41/199/1

[5] J. Sethian, *Level Set Methods and Fast Marching Methods*, 2nd ed. Cambridge, United Kingdom: Cambridge University Press, 1999.

[6] S. Kapoor, "Efficient computation of geodesic shortest paths," in *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1999, pp. 770–779.

[7] A. Duggan and M. Haisman, "Prediction of the metabolic cost of walking with and without loads," *Ergonomics*, vol. 35, pp. 417–426, 1992.

[8] J. Harris and F. Benedict, *A Biometric Study of Basal Metabolism in Man.* Carnegie Institute of Washington, Washington, 1919.

[9] K. Pandolf, B. Givoni, and R. Goldman, "Predicting energy expenditure with loads while standing or walking slowly," *Journal of Applied Physiology*, vol. 43, no. 4, pp. 577–581, 1977.

[10] W. R. Santee, W. F. Allison, L. Blanchard, and M. Small, "A proposed model for load carriage on sloped terrain," *Aviation, Space and Environmental Medicine*, vol. 72, no. 6, pp. 562–566, 2001.

[11] W. Tobler, "Three presentations on geographical analysis and modeling: Non- isotropic geographic modeling; speculations on the geometry of geography; and global spatial analysis," National Center for Geographic Information and Analysis, Tech. Rep. 93-1, February 1993. [Online]. Available: http://www.ncgia.ucsb.edu/Publications/Tech_Reports/93/93-1.PDF

[12] U.S. Department of the Interior. (2007, May) U.S. geological survey. [Online]. Available: http://www.usgs.gov

[13] U.S. Department of Commerce. (2007, May) National geophysical data center home page. [Online]. Available: http://www.ngdc.noaa.gov

[14] ESRI. (2007, May) ESRI - the GIS software leader. [Online]. Available: http://www.esri.com

[15] SGI. (2007, May) OpenGL. [Online]. Available: http://www.opengl.org

[16] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," *SIAM J. Comput.*, vol. 16, no. 4, pp. 647–668, 1987.

[17] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe, "Fast exact and approximate geodesics on meshes," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 553–560, 2005.

[18] R. Johnsonbaugh, *Discrete Mathematics*, 5th ed. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[19] R. Kimmel and J. A. Sethian, "Optimal algorithm for shape from shading and path planning," *J. Math. Imaging Vis.*, vol. 14, no. 3, pp. 237–244, 2001.

[20] J. N. Tsitsiklis, "Efficient algorithms for globally optimal trajectories," *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1528–1538, Sep. 1995.

[21] L. Yatziv, A. Bartesaghi, and G. Sapiro, "O(n) implementation of the fast marching algorithm," *Journal of Computational Physics*, vol. 212, no. 2, pp. 393–399, 2006.

[22] M. Novotni and R. Klein, "Computing geodesic distances on triangular meshes," in *The 10-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2002 (WSCG'2002)*, Feb. 2002.

[23] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers.* New York, NY, USA: ACM Press, 2006, pp. 1160–1168.

[24] M. Herrmann, "A domain decomposition parallelization of the fast marching method," *CTR Annual Research Briefs*, pp. 213–225, Dec. 2003.

[25] Oracle. (2007, May) Oracle Berkeley DB. [Online]. Available: http://www.oracle.com/technology/products/berkeley-db/db/index.html

[26] D. Martinez, L. Velho, and P. C. Carvalho, "Geodesic paths on triangular meshes," *sibgrapi*, pp. 210–217, 2004.

[27] U.S. Department of the Interior. (2007, June) USGS seamless data distribution. [Online]. Available: http://seamless.usgs.gov/website/seamless/viewer.php

[28] Oregon Department of Transportation. (2007, May) Support services oregon trail. [Online]. Available: http://www.oregon.gov/ODOT/CS/SSB/Oregon_Trail.shtml

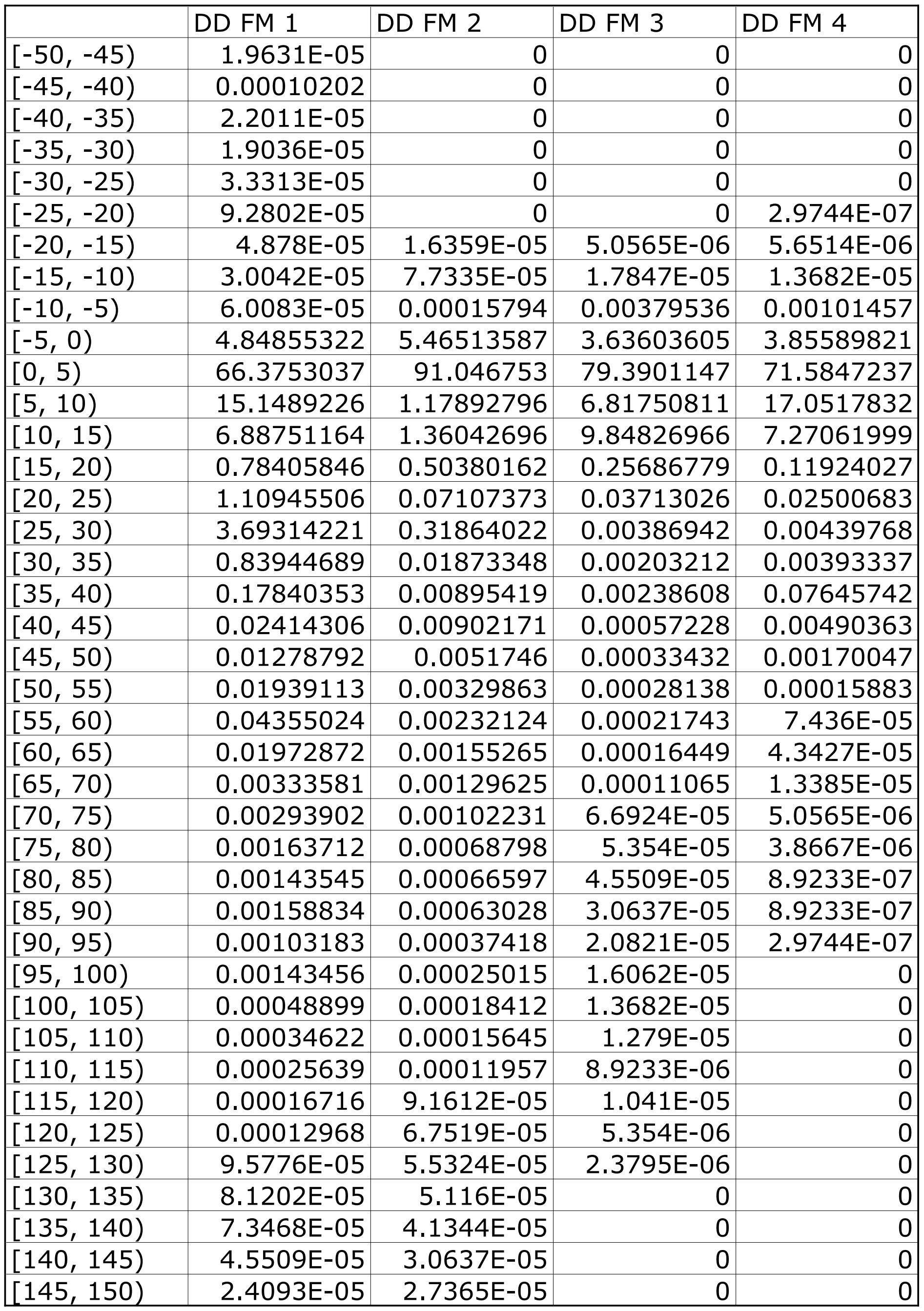
[29] End of the Oregon Trail Interpretive Center. (2007, June) The oregon trail in oregon. [Online]. Available: http://www.endoftheoregontrail.org/maplibrary/oregon.html

[30] Google. (2007, June) Google earth. [Online]. Available: http://earth.google.com

Appendix A

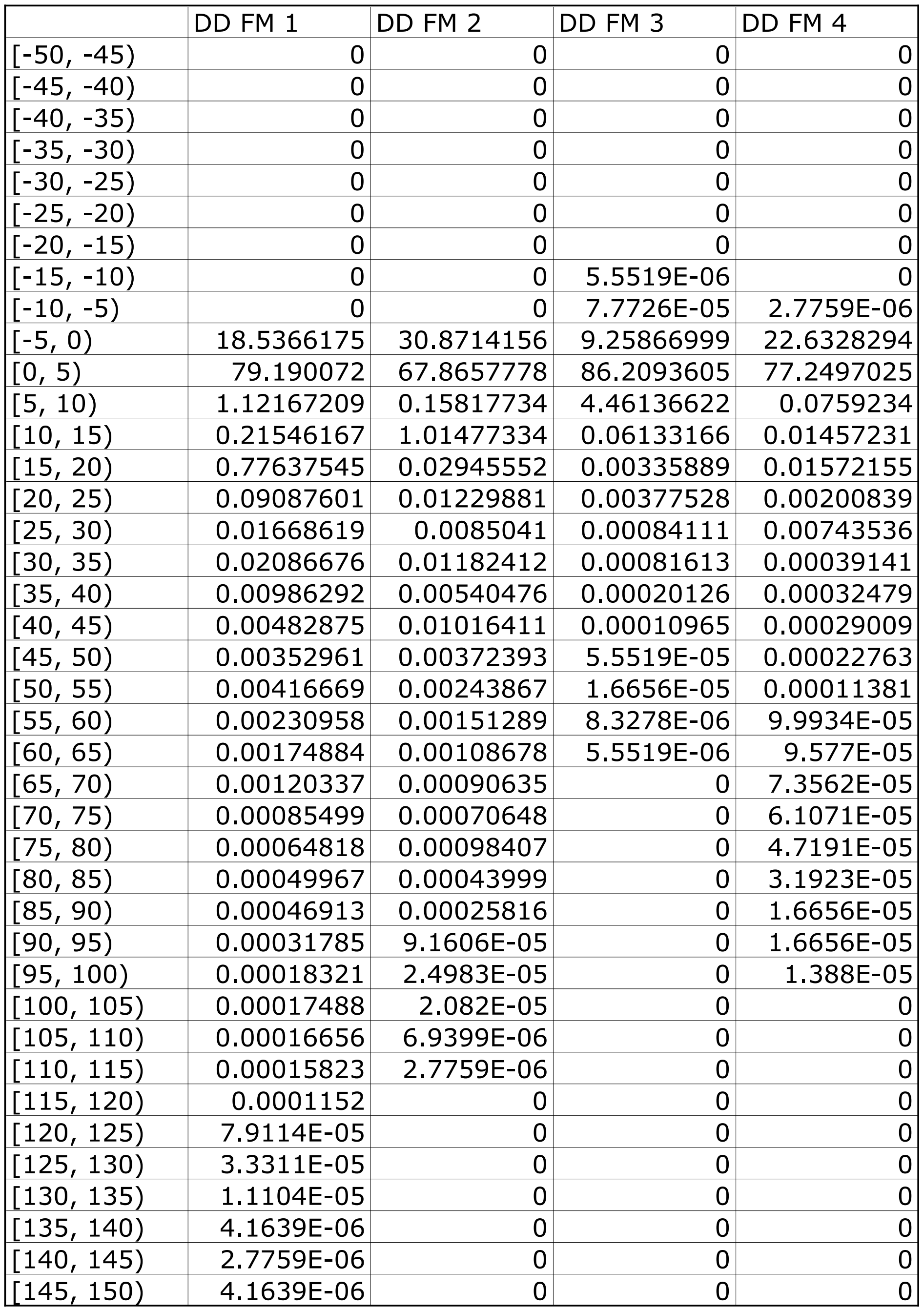
More Caloric Error Results for Domain Decomposition-Based Fast Marching

The tables in Figures A.1 through A.5 list the caloric error for individual runs of the domain decomposition-based fast marching algorithms on each data set. Figure A.6 shows the full table of averaged error for all data sets.



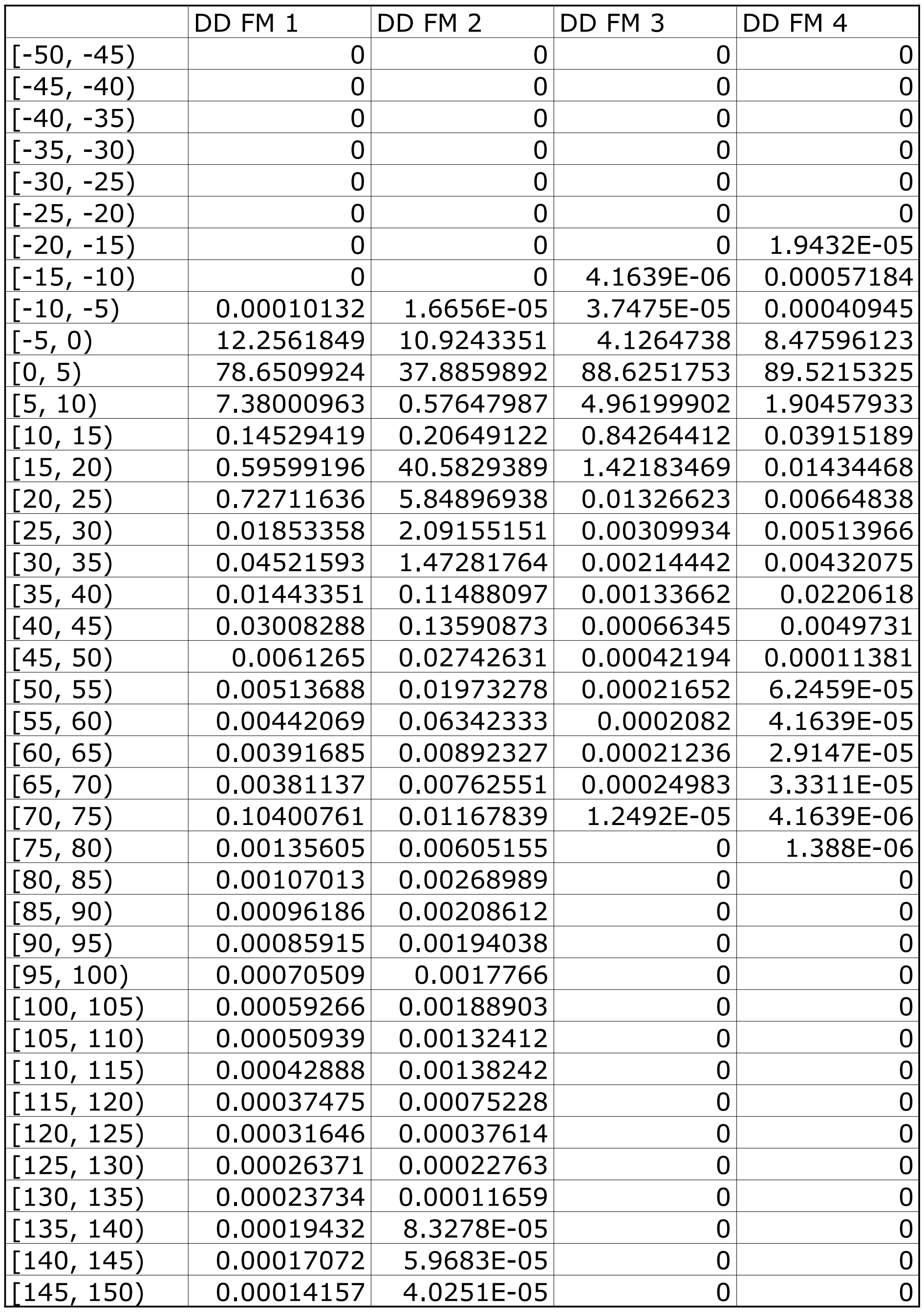
	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	1.9631E-05	0	0	0
[-45, -40)	0.00010202	0	0	0
[-40, -35)	2.2011E-05	0	0	0
[-35, -30)	1.9036E-05	0	0	0
[-30, -25)	3.3313E-05	0	0	0
[-25, -20)	9.2802E-05	0	0	2.9744E-07
[-20, -15)	4.878E-05	1.6359E-05	5.0565E-06	5.6514E-06
[-15, -10)	3.0042E-05	7.7335E-05	1.7847E-05	1.3682E-05
[-10, -5)	6.0083E-05	0.00015794	0.00379536	0.00101457
[-5, 0)	4.84855322	5.46513587	3.63603605	3.85589821
[0, 5)	66.3753037	91.046753	79.3901147	71.5847237
[5, 10)	15.1489226	1.17892796	6.81750811	17.0517832
[10, 15)	6.88751164	1.36042696	9.84826966	7.27061999
[15, 20)	0.78405846	0.50380162	0.25686779	0.11924027
[20, 25)	1.10945506	0.07107373	0.03713026	0.02500683
[25, 30)	3.69314221	0.31864022	0.00386942	0.00439768
[30, 35)	0.83944689	0.01873348	0.00203212	0.00393337
[35, 40)	0.17840353	0.00895419	0.00238608	0.07645742
[40, 45)	0.02414306	0.00902171	0.00057228	0.00490363
[45, 50)	0.01278792	0.0051746	0.00033432	0.00170047
[50, 55)	0.01939113	0.00329863	0.00028138	0.00015883
[55, 60)	0.04355024	0.00232124	0.00021743	7.436E-05
[60, 65)	0.01972872	0.00155265	0.00016449	4.3427E-05
[65, 70)	0.00333581	0.00129625	0.00011065	1.3385E-05
[70, 75)	0.00293902	0.00102231	6.6924E-05	5.0565E-06
[75, 80)	0.00163712	0.00068798	5.354E-05	3.8667E-06
[80, 85)	0.00143545	0.00066597	4.5509E-05	8.9233E-07
[85, 90)	0.00158834	0.00063028	3.0637E-05	8.9233E-07
[90, 95)	0.00103183	0.00037418	2.0821E-05	2.9744E-07
[95, 100)	0.00143456	0.00025015	1.6062E-05	0
[100, 105)	0.00048899	0.00018412	1.3682E-05	0
[105, 110)	0.00034622	0.00015645	1.279E-05	0
[110, 115)	0.00025639	0.00011957	8.9233E-06	0
[115, 120)	0.00016716	9.1612E-05	1.041E-05	0
[120, 125)	0.00012968	6.7519E-05	5.354E-06	0
[125, 130)	9.5776E-05	5.5324E-05	2.3795E-06	0
[130, 135)	8.1202E-05	5.116E-05	0	0
[135, 140)	7.3468E-05	4.1344E-05	0	0
[140, 145)	4.5509E-05	3.0637E-05	0	0
[145, 150)	2.4093E-05	2.7365E-05	0	0

Figure A.1: The caloric error results for the Full Oregon DEM data set. The data represents percentage of points in a given caloric error range.



	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	0	0	0	0
[-45, -40)	0	0	0	0
[-40, -35)	0	0	0	0
[-35, -30)	0	0	0	0
[-30, -25)	0	0	0	0
[-25, -20)	0	0	0	0
[-20, -15)	0	0	0	0
[-15, -10)	0	0	5.5519E-06	0
[-10, -5)	0	0	7.7726E-05	2.7759E-06
[-5, 0)	18.5366175	30.8714156	9.25866999	22.6328294
[0, 5)	79.190072	67.8657778	86.2093605	77.2497025
[5, 10)	1.12167209	0.15817734	4.46136622	0.0759234
[10, 15)	0.21546167	1.01477334	0.06133166	0.01457231
[15, 20)	0.77637545	0.02945552	0.00335889	0.01572155
[20, 25)	0.09087601	0.01229881	0.00377528	0.00200839
[25, 30)	0.01668619	0.0085041	0.00084111	0.00743536
[30, 35)	0.02086676	0.01182412	0.00081613	0.00039141
[35, 40)	0.00986292	0.00540476	0.00020126	0.00032479
[40, 45)	0.00482875	0.01016411	0.00010965	0.00029009
[45, 50)	0.00352961	0.00372393	5.5519E-05	0.00022763
[50, 55)	0.00416669	0.00243867	1.6656E-05	0.00011381
[55, 60)	0.00230958	0.00151289	8.3278E-06	9.9934E-05
[60, 65)	0.00174884	0.00108678	5.5519E-06	9.577E-05
[65, 70)	0.00120337	0.00090635	0	7.3562E-05
[70, 75)	0.00085499	0.00070648	0	6.1071E-05
[75, 80)	0.00064818	0.00098407	0	4.7191E-05
[80, 85)	0.00049967	0.00043999	0	3.1923E-05
[85, 90)	0.00046913	0.00025816	0	1.6656E-05
[90, 95)	0.00031785	9.1606E-05	0	1.6656E-05
[95, 100)	0.00018321	2.4983E-05	0	1.388E-05
[100, 105)	0.00017488	2.082E-05	0	0
[105, 110)	0.00016656	6.9399E-06	0	0
[110, 115)	0.00015823	2.7759E-06	0	0
[115, 120)	0.0001152	0	0	0
[120, 125)	7.9114E-05	0	0	0
[125, 130)	3.3311E-05	0	0	0
[130, 135)	1.1104E-05	0	0	0
[135, 140)	4.1639E-06	0	0	0
[140, 145)	2.7759E-06	0	0	0
[145, 150)	4.1639E-06	0	0	0

Figure A.2: The caloric error results for the Oregon DEM Part 1 data set. The data represents percentage of points in a given caloric error range.



	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	0	0	0	0
[-45, -40)	0	0	0	0
[-40, -35)	0	0	0	0
[-35, -30)	0	0	0	0
[-30, -25)	0	0	0	0
[-25, -20)	0	0	0	0
[-20, -15)	0	0	0	1.9432E-05
[-15, -10)	0	0	4.1639E-06	0.00057184
[-10, -5)	0.00010132	1.6656E-05	3.7475E-05	0.00040945
[-5, 0)	12.2561849	10.9243351	4.1264738	8.47596123
[0, 5)	78.6509924	37.8859892	88.6251753	89.5215325
[5, 10)	7.38000963	0.57647987	4.96199902	1.90457933
[10, 15)	0.14529419	0.20649122	0.84264412	0.03915189
[15, 20)	0.59599196	40.5829389	1.42183469	0.01434468
[20, 25)	0.72711636	5.84896938	0.01326623	0.00664838
[25, 30)	0.01853358	2.09155151	0.00309934	0.00513966
[30, 35)	0.04521593	1.47281764	0.00214442	0.00432075
[35, 40)	0.01443351	0.11488097	0.00133662	0.0220618
[40, 45)	0.03008288	0.13590873	0.00066345	0.0049731
[45, 50)	0.0061265	0.02742631	0.00042194	0.00011381
[50, 55)	0.00513688	0.01973278	0.00021652	6.2459E-05
[55, 60)	0.00442069	0.06342333	0.0002082	4.1639E-05
[60, 65)	0.00391685	0.00892327	0.00021236	2.9147E-05
[65, 70)	0.00381137	0.00762551	0.00024983	3.3311E-05
[70, 75)	0.10400761	0.01167839	1.2492E-05	4.1639E-06
[75, 80)	0.00135605	0.00605155	0	1.388E-06
[80, 85)	0.00107013	0.00268989	0	0
[85, 90)	0.00096186	0.00208612	0	0
[90, 95)	0.00085915	0.00194038	0	0
[95, 100)	0.00070509	0.0017766	0	0
[100, 105)	0.00059266	0.00188903	0	0
[105, 110)	0.00050939	0.00132412	0	0
[110, 115)	0.00042888	0.00138242	0	0
[115, 120)	0.00037475	0.00075228	0	0
[120, 125)	0.00031646	0.00037614	0	0
[125, 130)	0.00026371	0.00022763	0	0
[130, 135)	0.00023734	0.00011659	0	0
[135, 140)	0.00019432	8.3278E-05	0	0
[140, 145)	0.00017072	5.9683E-05	0	0
[145, 150)	0.00014157	4.0251E-05	0	0

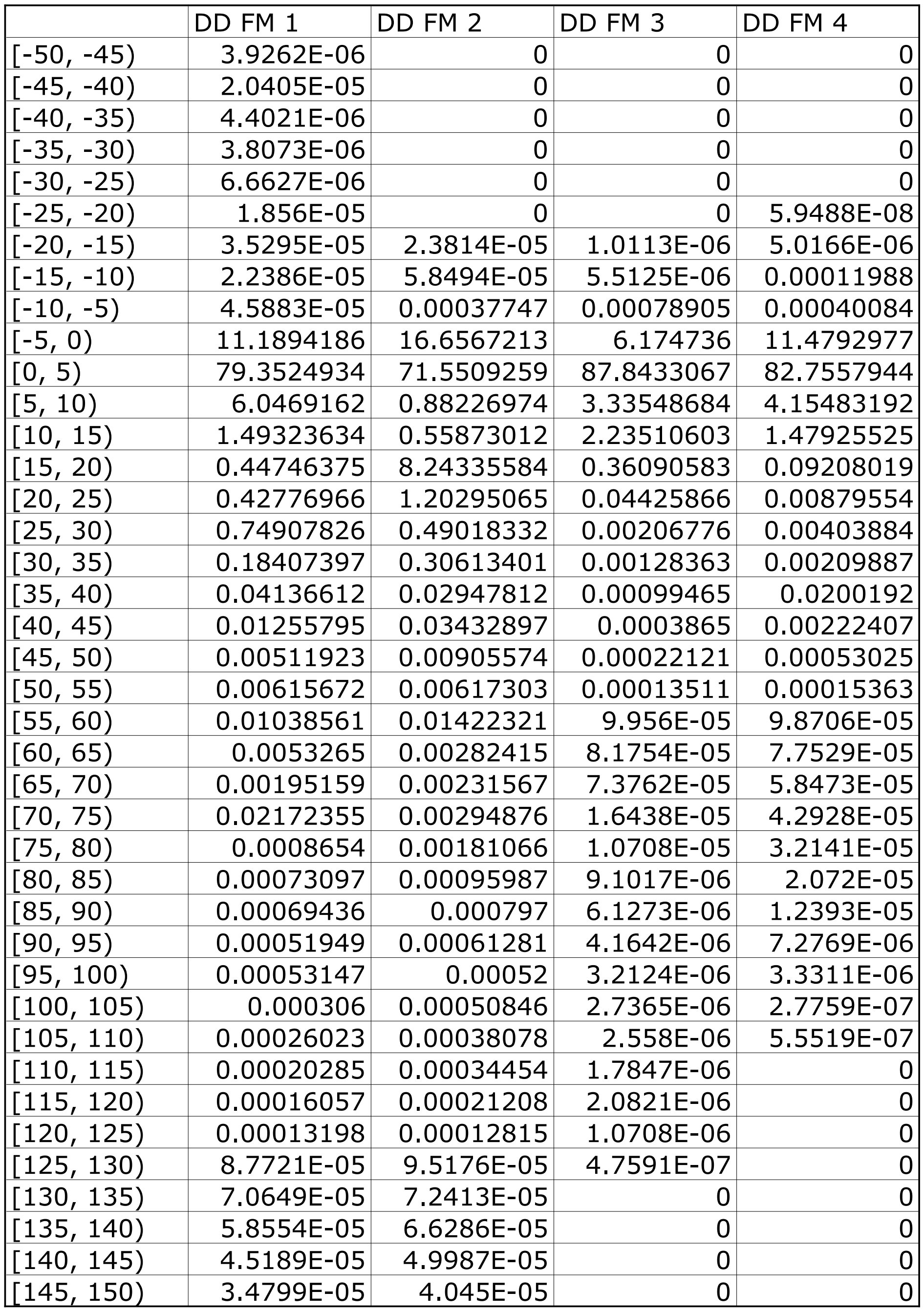
Figure A.3: The caloric error results for the Oregon DEM Part 2 data set. The data represents percentage of points in a given caloric error range.

	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	0	0	0	0
[-45, -40)	0	0	0	0
[-40, -35)	0	0	0	0
[-35, -30)	0	0	0	0
[-30, -25)	0	0	0	0
[-25, -20)	0	0	0	0
[-20, -15)	0.00012769	0.00010271	0	0
[-15, -10)	8.189E-05	0.00021514	0	1.388E-05
[-10, -5)	6.8011E-05	0.00171276	3.4699E-05	0.0005774
[-5, 0)	10.8006211	21.6311807	6.96897729	10.1164249
[0, 5)	82.5395222	77.5017178	92.1988397	87.7894017
[5, 10)	6.21413899	0.4351289	0.12617489	1.70093484
[10, 15)	0.11835367	0.15426603	0.41417748	0.06640747
[15, 20)	0.07413431	0.08842763	0.12059525	0.30880827
[20, 25)	0.20680351	0.07592062	0.16598884	0.00936186
[25, 30)	0.01301362	0.02726947	0.00205559	0.00270238
[30, 35)	0.01378672	0.02319855	0.00123946	0.00144349
[35, 40)	0.00363787	0.01565354	0.00088969	0.00099518
[40, 45)	0.00338804	0.01507059	0.00052327	0.00084944
[45, 50)	0.0028842	0.00796695	0.00027065	0.00058295
[50, 55)	0.00186543	0.00484402	0.00014435	0.00043305
[55, 60)	0.00144904	0.00354765	5.5519E-05	0.00027759
[60, 65)	0.00110482	0.00238037	2.2208E-05	0.0002193
[65, 70)	0.0012811	0.00171553	8.3278E-06	0.00017211
[70, 75)	0.00069537	0.00132829	2.7759E-06	0.00014435
[75, 80)	0.00059405	0.00132829	0	0.00010826
[80, 85)	0.00056074	0.0010035	0	7.0787E-05
[85, 90)	0.00037753	0.00101044	0	4.4415E-05
[90, 95)	0.00031923	0.0006579	0	1.9432E-05
[95, 100)	0.00026927	0.00054825	0	2.7759E-06
[100, 105)	0.00021514	0.00044831	0	1.388E-06
[105, 110)	0.00023179	0.00041639	0	2.7759E-06
[110, 115)	0.00013463	0.00021791	0	0
[115, 120)	0.00012075	0.00021652	0	0
[120, 125)	0.00012214	0.00019709	0	0
[125, 130)	4.1639E-05	0.00019293	0	0
[130, 135)	2.3596E-05	0.00019432	0	0
[135, 140)	2.082E-05	0.00020681	0	0
[140, 145)	6.9399E-06	0.00015962	0	0
[145, 150)	4.1639E-06	0.00013463	0	0

Figure A.4: The caloric error results for the Oregon DEM Part 3 data set. The data represents percentage of points in a given caloric error range.

	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	0	0	0	0
[-45, -40)	0	0	0	0
[-40, -35)	0	0	0	0
[-35, -30)	0	0	0	0
[-30, -25)	0	0	0	0
[-25, -20)	0	0	0	0
[-20, -15)	0	0	0	0
[-15, -10)	0	0	0	0
[-10, -5)	0	0	0	0
[-5, 0)	9.50511649	14.3915391	6.88352285	12.3153747
[0, 5)	90.0065768	83.4543916	92.7930432	87.6336117
[5, 10)	0.3698377	2.06263464	0.31038594	0.04093885
[10, 15)	0.09956053	0.05769306	0.00910724	0.00552459
[15, 20)	0.0067586	0.01215548	0.00187253	0.00228618
[20, 25)	0.00459734	0.00649069	0.00113268	0.00095223
[25, 30)	0.00401574	0.00495131	0.00047334	0.00051914
[30, 35)	0.00105356	0.00409624	0.000186	0.00040532
[35, 40)	0.00049277	0.00249717	0.00015963	0.0002568
[40, 45)	0.00034702	0.0014797	6.3852E-05	0.00010411
[45, 50)	0.0002679	0.00098693	2.3597E-05	2.6374E-05
[50, 55)	0.00022348	0.00055107	1.6657E-05	0
[55, 60)	0.0001985	0.00031093	8.3285E-06	0
[60, 65)	0.00013326	0.00017768	4.1643E-06	0
[65, 70)	0.00012632	3.4702E-05	0	0
[70, 75)	0.00012076	8.3285E-06	0	0
[75, 80)	9.1614E-05	1.3881E-06	0	0
[80, 85)	8.8838E-05	0	0	0
[85, 90)	7.4957E-05	0	0	0
[90, 95)	6.9404E-05	0	0	0
[95, 100)	6.524E-05	0	0	0
[100, 105)	5.83E-05	0	0	0
[105, 110)	4.7195E-05	0	0	0
[110, 115)	3.609E-05	0	0	0
[115, 120)	2.4986E-05	0	0	0
[120, 125)	1.2493E-05	0	0	0
[125, 130)	4.1643E-06	0	0	0
[130, 135)	0	0	0	0
[135, 140)	0	0	0	0
[140, 145)	0	0	0	0
[145, 150)	0	0	0	0

Figure A.5: The caloric error results for the Oregon DEM Part 4 data set. The data represents percentage of points in a given caloric error range.



	DD FM 1	DD FM 2	DD FM 3	DD FM 4
[-50, -45)	3.9262E-06	0	0	0
[-45, -40)	2.0405E-05	0	0	0
[-40, -35)	4.4021E-06	0	0	0
[-35, -30)	3.8073E-06	0	0	0
[-30, -25)	6.6627E-06	0	0	0
[-25, -20)	1.856E-05	0	0	5.9488E-08
[-20, -15)	3.5295E-05	2.3814E-05	1.0113E-06	5.0166E-06
[-15, -10)	2.2386E-05	5.8494E-05	5.5125E-06	0.00011988
[-10, -5)	4.5883E-05	0.00037747	0.00078905	0.00040084
[-5, 0)	11.1894186	16.6567213	6.174736	11.4792977
[0, 5)	79.3524934	71.5509259	87.8433067	82.7557944
[5, 10)	6.0469162	0.88226974	3.33548684	4.15483192
[10, 15)	1.49323634	0.55873012	2.23510603	1.47925525
[15, 20)	0.44746375	8.24335584	0.36090583	0.09208019
[20, 25)	0.42776966	1.20295065	0.04425866	0.00879554
[25, 30)	0.74907826	0.49018332	0.00206776	0.00403884
[30, 35)	0.18407397	0.30613401	0.00128363	0.00209887
[35, 40)	0.04136612	0.02947812	0.00099465	0.0200192
[40, 45)	0.01255795	0.03432897	0.0003865	0.00222407
[45, 50)	0.00511923	0.00905574	0.00022121	0.00053025
[50, 55)	0.00615672	0.00617303	0.00013511	0.00015363
[55, 60)	0.01038561	0.01422321	9.956E-05	9.8706E-05
[60, 65)	0.0053265	0.00282415	8.1754E-05	7.7529E-05
[65, 70)	0.00195159	0.00231567	7.3762E-05	5.8473E-05
[70, 75)	0.02172355	0.00294876	1.6438E-05	4.2928E-05
[75, 80)	0.0008654	0.00181066	1.0708E-05	3.2141E-05
[80, 85)	0.00073097	0.00095987	9.1017E-06	2.072E-05
[85, 90)	0.00069436	0.000797	6.1273E-06	1.2393E-05
[90, 95)	0.00051949	0.00061281	4.1642E-06	7.2769E-06
[95, 100)	0.00053147	0.00052	3.2124E-06	3.3311E-06
[100, 105)	0.000306	0.00050846	2.7365E-06	2.7759E-07
[105, 110)	0.00026023	0.00038078	2.558E-06	5.5519E-07
[110, 115)	0.00020285	0.00034454	1.7847E-06	0
[115, 120)	0.00016057	0.00021208	2.0821E-06	0
[120, 125)	0.00013198	0.00012815	1.0708E-06	0
[125, 130)	8.7721E-05	9.5176E-05	4.7591E-07	0
[130, 135)	7.0649E-05	7.2413E-05	0	0
[135, 140)	5.8554E-05	6.6286E-05	0	0
[140, 145)	4.5189E-05	4.9987E-05	0	0
[145, 150)	3.4799E-05	4.045E-05	0	0

Figure A.6: The caloric error results averaged over all data sets. The data represents percentage of points in a given caloric error range.