On Simulated Annealing and the Construction of Linear Spline Approximations for Scattered Data

Oliver Kreylos* Bernd Hamann*

April 2, 2001

Abstract

We describe a method to create optimal linear spline approximations to arbitrary functions of one or two variables, given as scattered data without known connectivity. We start with an initial approximation consisting of a fixed number of vertices and improve this approximation by choosing different vertices, governed by a simulated annealing algorithm. In the case of one variable, the approximation is defined by line segments; in the case of two variables, the vertices are connected to define a Delaunay triangulation of the selected subset of sites in the plane. In a second version of this algorithm, specifically designed for the bivariate case, we choose vertex sets and also change the triangulation to achieve both optimal vertex placement and optimal triangulation. We then create a hierarchy of linear spline approximations, each one being a superset of all lower-resolution ones.

1 Introduction

In several applications, one is concerned with the representation of complex geometries or complex physical phenomena at multiple levels of resolution.

*Center for Image Processing and Integrated Computing (CIPIC), Department of Computer Science, University of California, Davis, CA 95616-8562, USA; {kreylos, hamann}@cs.ucdavis.edu

In the context of computer graphics and scientific visualization, so-called *multi-resolution methods* are crucial for the analysis of very large numerical data sets, see [2, 3, 4, 5, 6, 7]. Examples include high-resolution terrain data (digital elevation maps), laser scans of mechanical models, see [8, 9], and high-resolution, three-dimensional imaging data (e.g., magnetic resonance imaging data).

We present an approach for the construction of multi-resolution representations of very large scattered data sets using the principle of *simulated annealing*, see [10, 11, 12]. Our goal is the computation of several optimal linear spline approximations to a given scattered data set. This approach is a generalization of data-dependent triangulation algorithms, see, for example, [13].

We assume that the given data sets are samples of a real function of one or two variables[1], with the samples randomly distributed in the function's domain and no known connectivity between the samples. Each individual linear spline approximation is defined by the set of its control points and by the way these points are connected forming a triangulation (simplex mesh).

When representing high-resolution data sets with low-resolution linear spline approximations, one has to be careful where to place the spline's control points and how to connect them in order to achieve a faithful representation of the data set, see Figures 1 and 2.

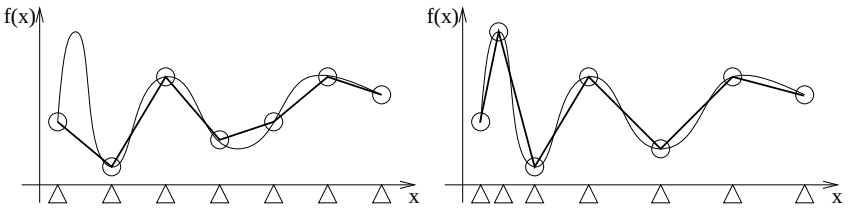

Figure 1: Uniform versus optimal control point placement for univariate data.

To find the optimal linear spline approximation with a given number of control points, we employ an iterative optimization algorithm that attempts to improve the quality, measured by an appropriate error norm, of an initial approximation by moving the spline's control points. This iteration is governed by the simulated annealing algorithm, an optimization technique

[1]The algorithm works for arbitrary numbers of variables, but in this paper we deal with the cases of one and two variables only.

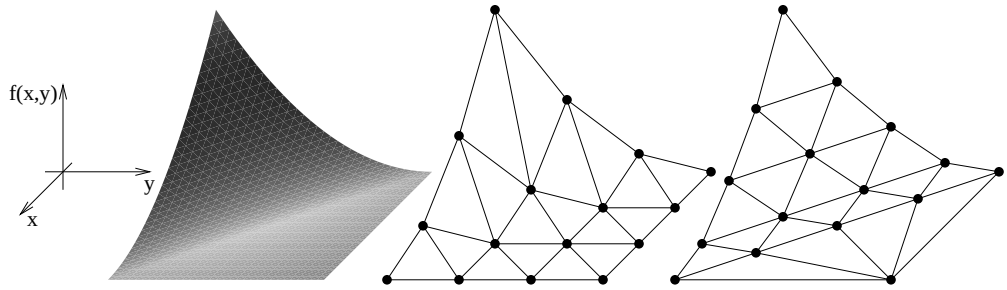

Figure 2: Uniform versus optimal control point placement for bivariate data. Left: graph of function to be approximated; center: uniform control point placement; right: optimal control point placement.

well fit for high-dimensional problems where the desired global minimum is hidden among many, poorer, local minima.

Since the functions we approximate are only defined at discrete, randomly distributed sites in the domain, we will restrict our algorithm in the following way: We only place control points at the given sites and only use the supplied function values at those sites. The main advantage of such an approach lies in the fact that no additional geometrical or function information needs to be stored. The problem of placing an approximating spline's control points is hereby reduced to choosing a subset of samples and connecting them in an appropriate way; and the process of "moving a control point" is really defined by removing one sample from the chosen subset and inserting another one instead. Thus, control points do not "move," we just select different samples.

1.1 Basic Definitions

To make the discussion of our algorithm easier and to allow describing the algorithm for the univariate and multivariate cases together, we define some special terms we use throughout this paper. Some of them originated in the field of computational geometry, while others are special uses of well-known mathematical terms.

- A *site* is a point in a function's domain. Considering functions of n (real) variables, a site is a point $s = (x_1, \ldots, x_n) \in \mathbf{R}^n$.

- The *convex hull* of a set $S \subset \mathbf{R}^n$ of sites is the "smallest" closed, convex subset $CH(S) \subset \mathbf{R}^n$ that contains all of S' sites, i.e., $S \subset CH(S)$.

For all dimensions, $CH(S)$ is bounded by a convex polyhedron whose vertices are a subset of S. We call a site $s \in S$ *interior*, if it is not an element of the vertex set defining $CH(S)$'s bounding polyhedron. In other words, a site s is interior, if and only if its removal would not change the convex hull, i.e., if $CH\big(S \setminus \{s\}\big) = CH(S)$, see Figure 3.

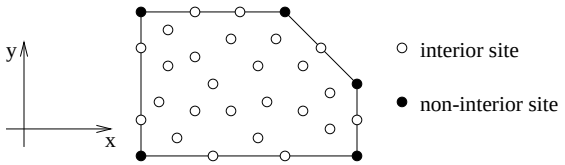

Figure 3: Interior and non-interior sites in the bivariate case.

- A *vertex* is a pair consisting of a site and the corresponding function value. Considering functions $f\colon \mathbf{R}^n \to \mathbf{R}^m$ of n variables and m function values, a vertex is defined by an $(n+m)$-tuple $v = \big(s, f(s)\big) = \big((x_1, \ldots, x_n), (f_1, \ldots, f_m)\big) \in (\mathbf{R}^n \times \mathbf{R}^m)$, see Figure 4. We call a vertex *interior vertex*, if its site is an interior site.

- *Scattered data* define a representation of a function $f\colon \mathbf{R}^n \to \mathbf{R}^m$ by a finite set of vertices, with all vertices having pairwise different sites.

- A *vertex placement* is an ordered set of vertices, with all vertices having pairwise different sites. For N vertices and a function $f\colon \mathbf{R}^n \to \mathbf{R}^m$ it is an element $V = \Big(\big(s_1, f(s_1)\big), \ldots, \big(s_N, f(s_N)\big)\Big) \in \big((\mathbf{R}^n \times \mathbf{R}^m)\big)^N$.

- A *connectivity* is a set of n-dimensional simplices which connect all sites in a vertex placement and overlap only on their boundaries. In the case of one variable, a connectivity is simply a set of adjacent intervals; for two variables it is a *triangulation*; and for three variables it is a *tetrahedral mesh* (or *tetrahedrization*).

  The $(n-1)$-dimensional simplices seperating adjacent simplices in the connectivity will be called *edges*, regardless of dimension. A vertex' *platelet* is the union of all simplices that share this vertex' site. In the case of one variable, a platelet is an interval, and for two variables it is an area bounded by a star polygon, see Figure 4.

- A *configuration* is the pair consisting of a vertex placement and the underlying connectivity.

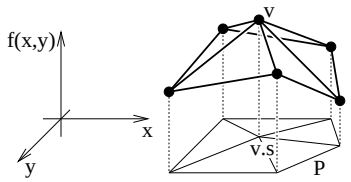

Figure 4: A vertex v, its site $v.s$ and its platelet's boundary polygon P in the bivariate case.

1.2 Visualizing Large Data Sets

When visualizing functions that are discretized in terms of large scattered data sets, one would like to render images of these functions in real time. To achieve this, one has to approximate a given function with a small enough number of *graphical primitives*, i.e., line segments or triangles, to render the function at interactive frame rates. To allow adjusting the approximation quality, a hierarchy of approximations with increasing numbers of primitives is highly desirable.

More specifically, at each hierarchy level k we choose N_k vertices from an original scattered data set, i.e., we only choose sites contained in the original set and only use the original function values at those sites. Furthermore, we ensure that the set of vertices of any hierarchy level $j < k$ is a subset of level k's vertex set. After having decided which vertices to select for a hierarchy level k, that level's vertices are connected in an appropriate way to form a linear spline's control mesh. The resulting linear splines then define a hierarchy of approximations to the function defined by the provided scattered data. An example of such a hierarchy in the univariate case is shown in Figure 5.

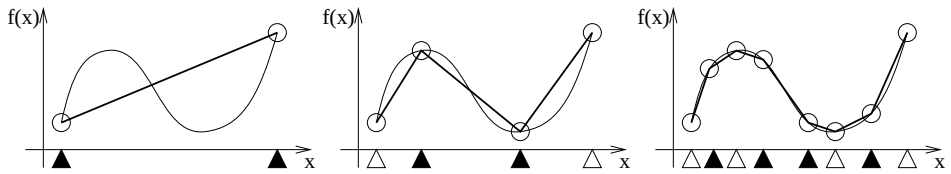

Figure 5: A hierarchy of approximations in the univariate case. New vertices are inserted at the sites marked by solid triangles.

If the number of vertices for an approximation level is prescribed, one has to address two problems:

1. Which vertices should one choose for the approximation, i.e., how should one create the vertex placement?

2. How should one connect the chosen vertices, i.e., how should one create the connectivity?

In the special case of a function of one variable, we only have to address the first problem, since in the univariate case the connectivity is defined by the chosen sites' numerical order.

1.3 Finding Optimal Approximations

Our approach to finding an optimal linear spline approximation for a given, fixed number of vertices N_k is based on an iterative optimization algorithm. First, we create an initial configuration, then we attempt to improve this configuration by changing its vertex placement and its connectivity in every step. Since this optimization problem is high-dimensional and generally involves local minima in abundance, the algorithm of simulated annealing, see section 2.1, is well suited to construct "good" linear spline approximations.

During the iteration process, we will restrict our algorithm to use only vertices that are elements of the original data set. This approach has two major advantages:

1. Since the function to be approximated is only known at discrete sites, we would have to estimate function values for sites that are not included in the original data set, i.e., we would have to define some appropriate scattered data interpolant or approximant.

2. By only using original vertices, each approximation is a subset of the original data set. To represent a particular approximation we only have to maintain a list of indices referring to this superset instead of a list of vertex positions. For the definition of a hierarchy of approximations at different resolutions we only need to store one integer for each original vertex: the integer that indicates the hierarchy level at which a vertex becomes active.

However, in the case of two, or more, variables the quality of a configuration depends on both vertex placement and connectivity. There are two different ways to proceed:

1. One can ignore the connectivity and treat a function of n variables like a univariate function by enforcing a certain type of connectivity throughout the iteration process; in the bivariate case, an obvious candidate is the Delaunay triangulation, see [14, 15]. Under this constraint, a vertex placement implies its connectivity (up to negligible ambiguities[2]), and the algorithm can proceed exactly as in the univariate case.

   Once the iteration process has terminated, one could use the optimal vertex placement as input for a *data-dependent triangulation algorithm*, which attempts to find the best possible connectivity for a given vertex placement, see [13]. The drawback of this two-step algorithm is that the final vertex placement may not be optimal for the final connectivity, since both parts were optimized independently.

2. One can attempt to optimize both parts of the configuration, namely vertex placement and connectivity, in parallel. For example, before each step one could randomly decide which part to change, and then either move a vertex or change the connectivity, which could include *swapping* a common edge of two adjacent triangles. The probability for deciding which part to change could either be constant throughout the algorithm, or it could change to favor connectivity changes in later stages. The drawbacks of this parallel algorithm are twofold: One has to treat the univariate case differently from the general case, and the optimization process will take even longer to finish.

We discuss both ways simultaneously, since the respective algorithms differ only slightly, and we will compare the results, see section 4.2.

2 Background and Related Work

2.1 The Simulated Annealing Algorithm

Simulated annealing is an iterative minimization technique suitable for optimization problems of large scale, especially those where a desired global extremum is "hidden" among many, poorer local extrema, see [11, 12, 13]. Unlike "greedy" iterative optimization techniques, which always take the best

[2]These ambiguities arise when a set of sites is not in *general position* or is *degenerate*, meaning that more than three sites are co-circular [16, 17, 18].

possible step from any configuration, simulated annealing sometimes takes a "bad" step and can hence overcome being stuck in a local minimum, see Figure 6. Thus, the chances of finding a global minimum are increased.

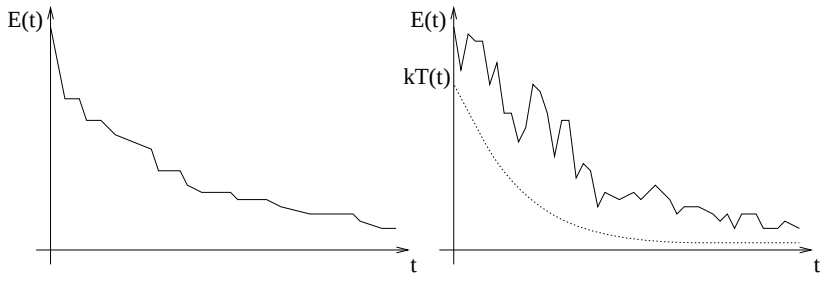

Figure 6: Sketch of error value over time for greedy optimization methods (left) and simulated annealing (right). As the temperature kT decreases, positive changes of E become less probable.

The simulated annealing algorithm was inspired, and received its name, by the process of metal crystallizing from the liquid phase back to the solid phase while its temperature slowly decreases. This process can be viewed as an minimization process: One defines a configuration as an n-tuple of the positions of n atoms and the function to be minimized as the liquid's internal energy. In spite of this being a large-scale optimization problem with local energy minima in abundance, the natural annealing process manages to reach the global energy minimum of a single crystal, as long as the liquid's temperature is lowered slowly enough. This is due to the fact that the natural annealing process allows steps decreasing the liquid's internal energy as well as steps increasing the energy. In fact, the probability prob(ΔE) for an increase ΔE in energy is determined by *Boltzmann's law of thermodynamics*, stating that prob(ΔE) $= e^{-\Delta E/kT}$, where ΔE is the energy increase in Joule (J), T is the liquid's current temperature in Kelvin (K) and $k = 1.38 \times 10^{-23}$ J/K is *Boltzmann's constant*.

The simulated annealing algorithm is an adaption of the natural annealing process for arbitrary optimization problems. In this adaption, a configuration is no longer the n-tuple of atom positions but a variable of arbitrary type; the liquid's internal energy is replaced by the function to be minimized; the movement of atoms is replaced by a relation that transforms one configuration into another one; and the term kT from Boltzmann's law is replaced by an arbitrary (positive, real) number called "temperature." We note that

the temperature is independent from the function to be minimized: The former can have an arbitrary value and defines the probability of accepting an increase in the latter during the optimization process, see Figure 6.

The generic simulated annealing algorithm, also known as the *Metropolis algorithm*, see [12], is shown in Algorithm 1:

Parameters:

- A space X of configurations.
- An initial configuration $C \in X$.
- A (non-deterministic) function *changeConfiguration*: $X \to X$ which transforms one configuration into the next one.
- A function *targetFunction*: $X \to \mathbf{R}$ which is to be minimized.
- An annealing schedule $kT\colon \mathbf{N} \to \mathbf{R}^+$.

Local Variables:

- An integer $n \in \mathbf{N}$.
- A configuration $newC \in X$.
- A number $\Delta E \in \mathbf{R}$.
- A function *prob*: $[0,1] \to \{0,1\}$, $p \mapsto r$, with the property that the probability $P(r = 1) = p$.

Results:

- A final configuration $C \in X$.

$n = 0$; /* iteration counter */
while iteration not finished **do**
  begin
  $newC = changeConfiguration(C)$;
  $\Delta E = targetFunction(newC) - targetFunction(C)$;
  if $\Delta E < 0$ **then**
    $C = newC$; /* accept all good steps */
  else if $prob\left(e^{-\Delta E/kT(n)}\right)$ **then**
    $C = newC$; /* accept some bad steps */
  $n = n + 1$;
  end
return C; /* return final configuration */

Algorithm 1: Generic simulated annealing.

The initial temperature and the decrease of temperature over the course of iteration are described by a decreasing function $kT\colon \mathbf{N}_0 \to \mathbf{R}^+$, called

"annealing schedule." Since the annealing schedule can be an arbitrary decreasing function and since it influences both the speed of optimization and the "optimality" of the result, its choice is another optimization problem: If the temperature is chosen to decrease too quickly, the algorithm might "get stuck" in a local minimum; on the other hand, if the temperature decreases too slowly, the algorithm usually requires a long time to converge to a minimum. Various heuristics have been developed that have proven useful for common problems, see [11, 13].

2.2 An Error Measure for Linear Spline Approximations to Scattered Data

To optimize an approximation to a given function, one has to define some quality measure, which describes how well a current configuration approximates a target function. When both the target function and the approximation are defined analytically, mathematics provides several well-defined function space metrics. However, in our case, the target function is defined by scattered data, which means that its behaviour is unknown between the provided sites. Nevertheless, if one uses a metric which is defined by an integral of some difference measure between the two functions, such as the L^2 metric, given by

$$L^2(f_1, f_2) = \sqrt{\int \big(f_1(x) - f_2(x)\big)^2 \, dx} \quad ,$$

it is possible to calculate a reasonable estimation to this metric.

If we assume that the given sites are randomly distributed in the target function's domain, we can approximate an integral by a variation of *Monte-Carlo-integration* and thus obtain a reasonable quality measure, see [11]. The method to calculate the L^2 distance between an approximation and a target function for the application we discuss in this paper is shown in Algorithm 2:

N = number of original vertices;
A = area of original sites' convex hull;
$dist = 0$; /* distance value */
for each original vertex v **do**
begin
Find simplex s in connectivity that contains v's site $v.s$;
Interpolate the simplex' value $\overline{f}$ at site $v.s$;
$dist = dist + (\overline{f} - v.f)^2$;
end
return $\sqrt{dist \cdot A/N}$; /* return L^2-distance */

Algorithm 2: Distance calculation.

Figure 7 illustrates how vertices contribute to the error value for functions of one and two variables.

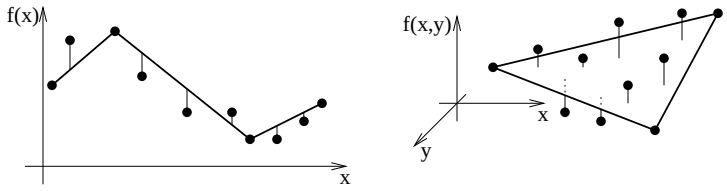

Figure 7: Vertices influencing the error measure.

In the case of one variable there is another appropriate scheme: We interpret the original vertices as control points of a linear spline f and interpret the current vertex placement as control points of a linear spline a. Then one can calculate the L^2 distance between f and a by algebraically integrating the functions' difference over the interval defined by the original sites' convex hull.

3 The Optimization Algorithm

We now describe the individual steps of our algorithm. Algorithm 3 is a high-level description. The subsequent sections describe the important steps in more detail.

Create initial configuration (vertex placement and connectivity);
Determine initial temperature and create annealing schedule;
while iteration is not complete **do**
    begin
    Change configuration;
    Calculate change in error norm;
    Undo iteration if rejected by simulated annealing;
    end
return current configuration;

Algorithm 3: Optimal linear spline approximation.

The algorithm's main loop terminates when "the iteration is complete." Depending on a user's needs, this condition can be specified in two ways:

1. The loop can be terminated after a prescribed number of iterations. This guarantees a certain runtime of the algorithm, but it does not guarantee the final configuration to be a minimum. Using this termination condition, the algorithm can be classified as a Monte-Carlo method.

2. The loop can be terminated when the annealing algorithm can no longer perform any iteration steps. This means the final configuration is a minimum, at least a local one. Since there is no upper bound on the number of iteration steps the algorithm will perform before reaching this condition, it can be classified as a Las-Vegas method.

In our experiments, we decided to use the Monte-Carlo termination condition. (In all examples, the number of iterations performed is stated in the respective error graphs.)

3.1 Creating an Initial Configuration

Our approximations are defined over the original sites' convex hull. Therefore, we include all non-interior vertices, as defined in section 1.1, in all approximations. Other vertices are then added to this set.

In the univariate case, we cover the convex hull by choosing the leftmost and the rightmost original vertices and spread the rest of vertices nearly uniformly between them, always choosing the nearest original site from a

calculated site. The connectivity is naturally defined by sorting the chosen vertices by their sites and connecting adjacent vertices by line segments.

In the multivariate cases we cannot distribute the vertices over a regular grid, because the original sites' convex hull can be arbitrarily shaped. Instead, we select all non-interior original vertices, thereby covering this convex hull, and we choose the rest of vertices randomly from the original data set. The initial connectivity is no longer determined by the initial vertex placement. In the bivariate case, we define the initial connectivity by a Delaunay triangulation of the initial vertices' sites. Since the Delaunay property maximizes the minimum angle in a triangulation, it is a reasonable first choice to approximate unknown functions. Furthermore, if we decide to ignore the problem of connectivity throughout the optimization, as mentioned in section 1.3, the Delaunay property is easy to maintain when moving vertices.

To create a Delaunay triangulation, we employ the iterative algorithm described by Guibas et al., see [19], inserting one site at a time into the current triangulation and restoring the Delaunay property afterwards by swapping diagonals of convex quadrilaterals that violate it. This might not be the fastest possible algorithm, though Knuth points out its expected runtime is $O(n \log n)$, but we only have to do this once and it is fairly easy to implement. Furthermore, most parts of this algorithm can be re-used for moving vertices, which is needed in the iteration algorithm described in section 3.3.

3.2 Creating an Annealing Schedule

There are two steps one has to perform to create the annealing schedule:

1. One has to define the initial temperature.

2. One has to decide how fast to decrease the temperature over the course of iteration.

A reasonable heuristic to define the initial temperature is to apply some (say, 100) steps of the iteration scheme and to calculate the mean error norm increase of all "bad" steps. After that, we define the initial temperature as $(1/\log_e 2)$ times the mean error norm change. Since the probability for accepting a "bad" step is defined as $p = e^{-\Delta E/kT}$, where ΔE is the (positive) error norm change and kT is the current temperature, the annealing algorithm initially accepts an "expected bad" step with a probability of one half.

Next, we lower the temperature in steps, leaving it constant for a fixed number of iterations and scaling it by a fixed factor afterwards, resulting in a geometric decrease in temperature. The number of iterations per temperature step and the scaling factor, smaller than one, can be chosen arbitrarily. These two parameters have an influence on both the algorithm's speed and the final result's quality: If the temperature is lowered too rapidly, the series of configurations might converge towards a far-from-optimal local minimum; if the temperature is lowered too slowly, the series might require a long time to converge.

3.3 Changing the Current Configuration

The simulated annealing algorithm's core is its iteration step. In principle, one can use any method to change the current configuration, but we have found out that the "split" approach, shown in Algorithm 4, works very well.

```
if prob(movePropability) then    /* move a vertex */
    begin
    Choose an interior vertex v;
    Estimate v's contribution vE to the error measure E;
    if vE < errorFactor · E then    /* v is in flat region */
        Move v globally;
    else    /* v is in high-curvature region */
        Move v locally;
    if movePropability = 1 then    /* only moves allowed */
        Restore the Delaunay property;
    end
else    /* rotate an edge */
    begin
    Choose an edge e which is the diagonal of a convex quadrilateral;
    Rotate e;
    end
```

Algorithm 4: Changing the current configuration.

The constant *movePropability* is used to control the behaviour of the optimization process for bivariate functions. If this constant's value is one, the algorithm moves a vertex in every step, and after each vertex movement the current triangulation is updated to satisfy the Delaunay property. Thus, the

algorithm maintains a Delaunay triangulation of the current sites throughout the optimization process, ignoring the problem of optimizing the triangulation and concentrating on finding an optimal vertex placement instead. If *moveProbability* is smaller than one, the algorithm can either move a vertex or swap an edge, thereby optimizing both vertex placement and triangulation simultaneously. In this case, we do no longer enforce the Delaunay property after vertex movements. Section 1.3 discusses both approaches. In the case of *moveProbability* being zero, the algorithm swaps an edge in each step and becomes a data-dependent triangulation algorithm. For univariate functions, *moveProbability* is always one, since the problem of connectivity does not arise.

The variable E holds the current distance between the scattered data set and the approximating spline, and the constant *errorFactor* determines the maximum error contribution which still classifies a vertex as being located in a flat region of the approximated function.

3.3.1 Estimating a vertex' error contribution

To estimate how much the removal of an interior vertex v would increase the current error measure, we estimate the "volume" of v's platelet: We construct an approximating least squares hyperplane H for all vertices surrounding v, and calculate the point p having the same site as v and lying on H. Then we define h as the distance between v and p, and A as the area of v's platelet. Figure 8 illustrates this for the univariate and bivariate cases.

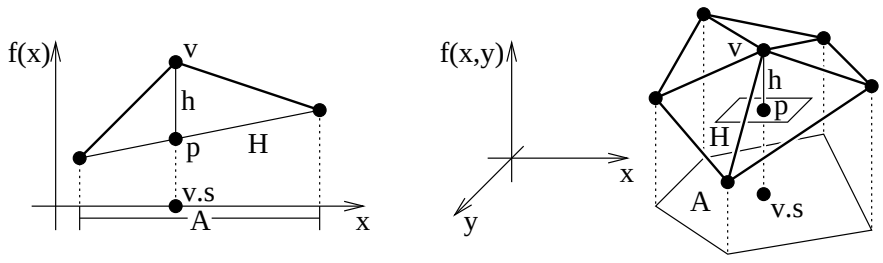

Figure 8: Estimating a vertex' error contribution.

A possible and reasonable definition of v's error contribution is the volume of the "hyperpyramid" having base A and height h. This volume is given by $A \cdot h/2$ in the univariate case and by $A \cdot h/3$ in the bivariate case. In order to make this estimation comparable to our L^2 error measure, we define

a vertex' error contribution vE in the univariate case as $\sqrt{A \cdot h^2/2}$ and in the bivariate case as $\sqrt{A \cdot h^2/3}$.

3.3.2 Global vertex movement

If v's error contribution is smaller than a constant times the current error, we assume that v is currently located in a "flat" region of the function and should be moved away from this region. We move v *globally* to a randomly chosen new site not already being part of the current configuration. By doing this we assure that vertices get driven away from nearly flat regions of a function in early stages of the iteration.

To actually "move" a vertex globally we (1) remove it from the configuration; (2) we fill the resulting hole in the connectivity (in the bivariate case by re-triangulating the vertex' platelet); and (3) we insert the vertex into the configuration at the new site by splitting the simplex that contains this site. If we decided to ignore the optimization of the connectivity by defining *moveProbability* as one, we would have to update the connectivity after moving a vertex by performing edge swapping until the Delaunay condition is satisfied, which is described in [19]. Figure 9 illustrates this process for the bivariate case.

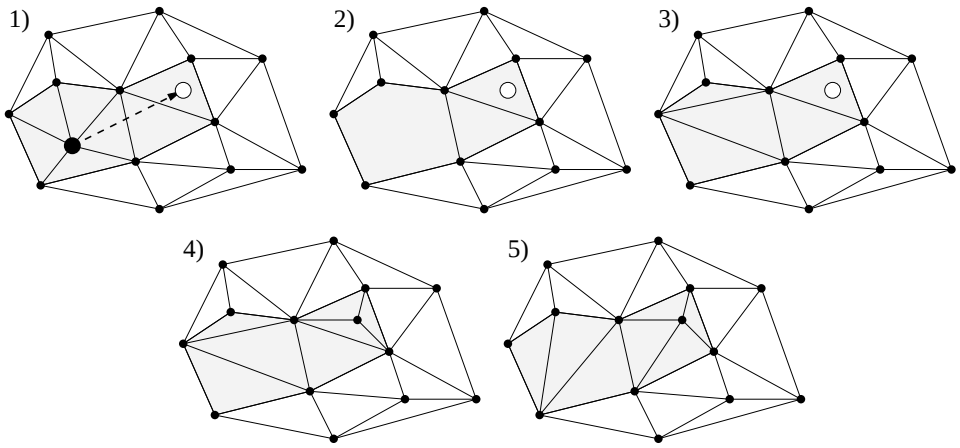

Figure 9: Moving a vertex globally in the bivariate case. 1) initial state; 2) removing the vertex; 3) filling the hole; 4) inserting new vertex; 5) restoring the Delaunay property (optional).

There is one special case to consider when moving a vertex: A new site

lying on an edge that is currently part of the connectivity must be treated differently, since the simplex, or simplices, covering this site cannot be split in the usual way. (Usually, we insert a new vertex by removing the simplex that covers its site and inserting edges connecting the new vertex to all vertices defining that simplex.) Considering the bivariate case, if the edge is part of the boundary of the original sites' convex hull, the triangle covering the new site is split into two triangles. If the edge is *interior*, i.e., it is shared by two triangles, both triangles sharing this edge are split into two triangles. (However, if one decides to use a Delaunay triangulation throughout the optimization process, the latter case can be ignored: One can split any of the two triangles in the usual way, thereby introducing a degenerate triangle, and the edge swapping step will generate the correct result.)

3.3.3 Local vertex movement

When a vertex' error contribution is larger than $errorFactor \cdot E$, we assume it is currently located in an "important," high-curvature region of the target function, and we attempt to find a better site for this vertex by moving it *locally* to a new, unoccupied site in its platelet. The maximum distance a vertex can be moved locally is also bounded by a constant called *localDistance*. This increases the probability of making a "good" step in later stages of the iteration. Since the given vertices are randomly distributed, it would be difficult to first select a subset of "near" vertices and then to choose one of them randomly. Instead, we employ a probabilistic method: We randomly select an original vertex w lying inside the platelet of the vertex v to be moved, calculate the Euclidian distance d between v and w, and accept w with the probability $e^{-(d/localDistance)^2}$. If w is rejected, we choose other vertices until one is accepted or until more than a fixed number of vertices has been rejected. This ensures that the distance a vertex is moved locally is distributed in a bell-shaped manner, with the probability of accepting a small displacement being close to one. We do not use a normal distribution for the distance, because the probability of accepting even a very small displacement would be considerably smaller than one in that case.

However, if we applied the global movement algorithm to a small displacement, the connectivity surrounding the moved vertex could change drastically, which would probably increase the error value. This is due to the fact that the vertex' platelet is re-triangulated before the vertex is inserted again, see Figure 10.

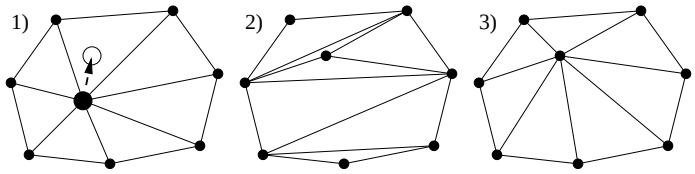

Figure 10: Drawbacks of moving a vertex a small distance by the global method in the bivariate case: 1) initial state; 2) result after global move as shown in Figure 9 (without enforcment of the Delaunay property); 3) desired result.

It is desirable to change the connectivity as little as possible when adjusting a vertex' position by a small displacement, see Figure 10, parts 1 and 3. To achieve this, we use a different method to move a vertex locally. Conceptually, we "slide" the vertex on the line from its old to its new site, dragging the edges connecting it to all surrounding vertices along. Whenever a surrounding simplex becomes degenerate during the vertex' motion, we swap one edge of the affected simplex before moving the vertex any further, see Figure 11. This method of swapping edges while moving the vertex also works for moving a vertex over arbitrary distances, and out of its initial platelet, but it changes the connectivity between the vertex' old and new site, and it can result in the vertex' new platelet containing both the old and the new site.

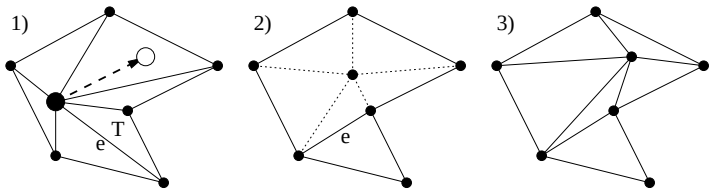

Figure 11: Moving a vertex locally in the bivariate case for a non-convex platelet: 1) initial state; 2) swapping edge e to prevent triangle T from becoming degenerate; 3) resulting state.

4 Examples and Results

4.1 Univariate Scalar-valued Functions

1. The first test case is the function $f(x) = 3\sin(x^2)$, $x \in [0, 4\sqrt{\pi}\,]$, and a linear spline approximation with 18 vertices, see Figure 12. Though it is hard to prove, our algorithm finds an approximation that looks globally optimal.

2. The second test case is the function

$$f(x) = \begin{cases} 2(1-x) & \text{if } x < 1 \\ 4(1-x)(x-2) & \text{if } 1 \le x < 2 \\ 2(x-2) & \text{if } 2 \le x < 3 \\ 2(x-3)^2 & \text{if } 3 \le x \end{cases} \quad , \quad x \in [0, 4],$$

   and a linear spline approximation with 14 vertices, see Figure 13. Our algorithm finds a very good approximation, although the function has discontinuities in both the zeroeth and first derivatives. In the two quadratic sections $[1, 2]$ and $[3, 4]$ the sites are uniformly distributed; we thus assume that the resulting approximation is globally optimal.

3. The third test case is the function

$$f(x) = 4\sum_{n=0}^{3} \frac{\sin\big((2n+1)x\big)}{2n+1} \quad , \quad x \in [0, 4\pi],$$

   the fourth-order Fourier approximation of a square wave, and a linear spline approximation with ten vertices, see Figure 14. The number of vertices is too small to capture all details of the function, but the algorithm still finds a very good approximation.

4. The fourth test case is the same function as in the third, but using a linear spline approximation with 30 vertices, see Figure 15. Now all the function's important features are present in the approximation – the graph of the original function does not show in Figure 15 because it is completely overlayed by the approximating polyline.

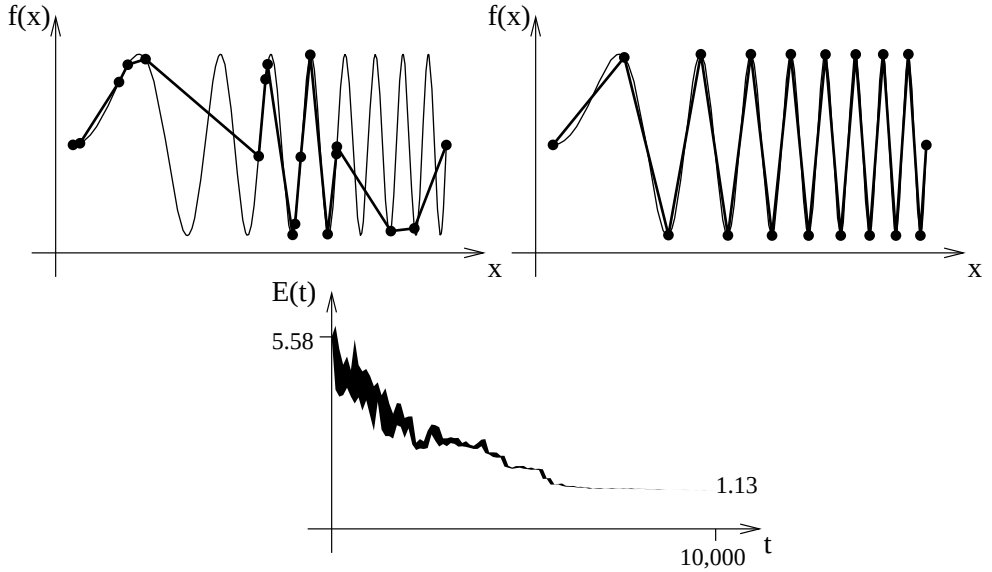

Figure 12: First experiment. Upper-left: initial vertex placement; upper-right: final vertex placement; bottom: error measure over time.

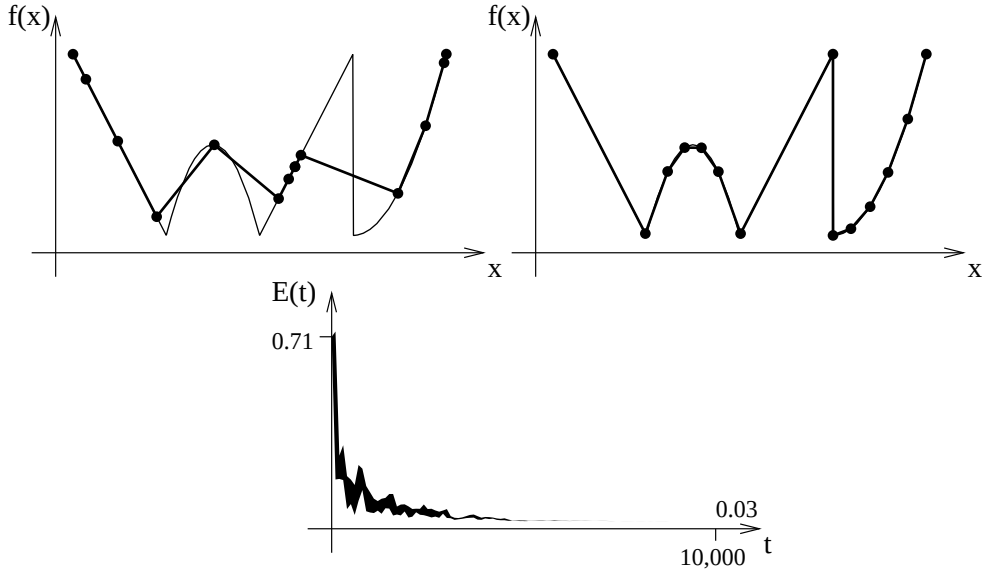

Figure 13: Second experiment. Upper-left: initial vertex placement; upper-right: final vertex placement; bottom: error measure over time.

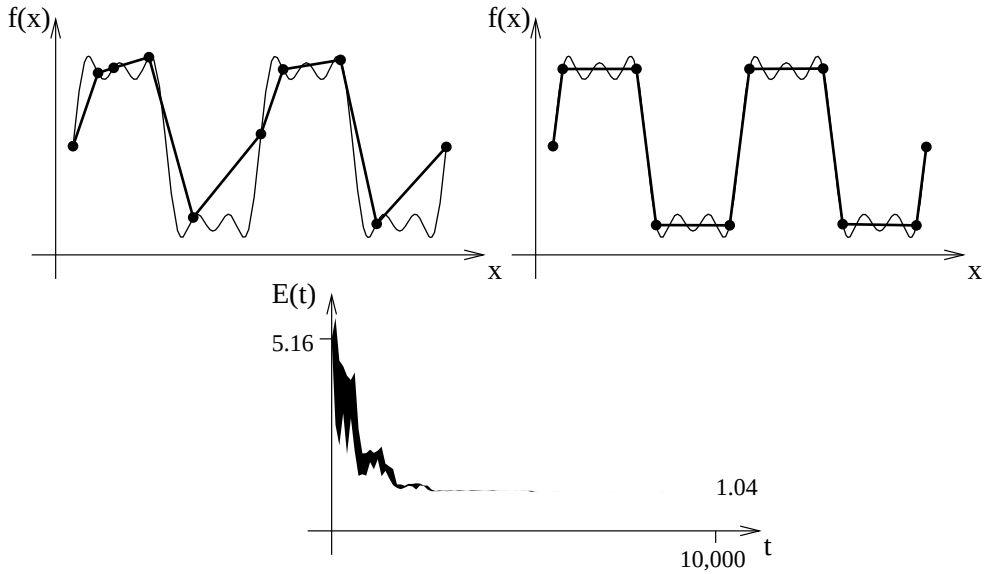

Figure 14: Third experiment. Upper-left: initial vertex placement; upper-right: final vertex placement; bottom: error measure over time.

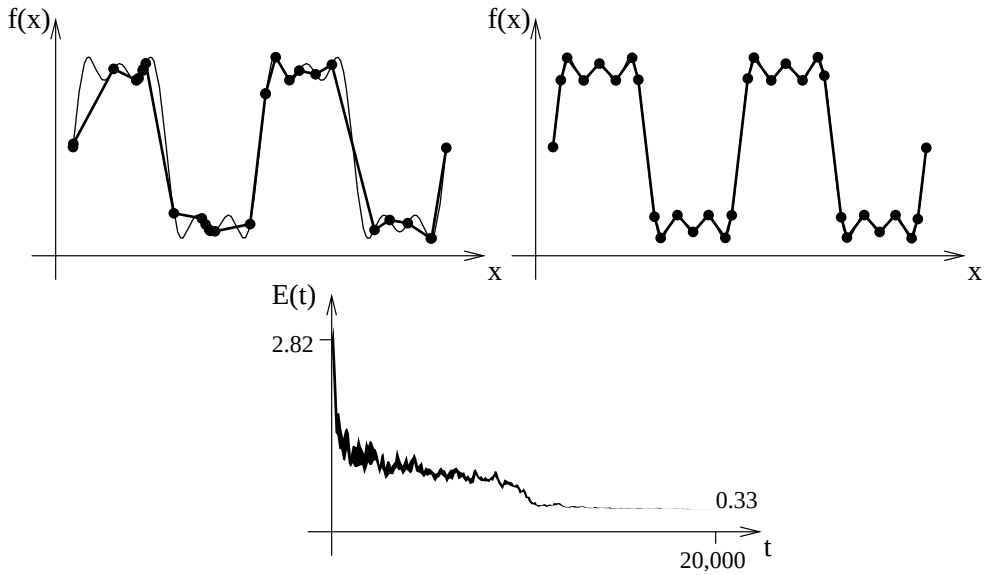

Figure 15: Fourth experiment. Upper-left: initial vertex placement; upper-right: final vertex placement; bottom: error measure over time.

4.2 Bivariate Scalar-valued Functions

5. The fifth test case is the function

$$f(x,y) = \begin{cases} 0.3 & \text{if } x^2+y^2 \le 0.3 \\ -0.5x & \text{if } x^2+y^2 > 0.3 \text{ and } x < 0 \\ x^2 & \text{if } x^2+y^2 > 0.3 \text{ and } 0 \le x \end{cases} \quad , \quad x,y \in [-1,1],$$

and a linear spline approximation with 100 vertices and general triangulation, see Figure 16. Our algorithm finds a very good approximation, although the function has discontinuities in both the zeroeth and first derivatives.

6. The sixth test case is the same function as in the fifth, but this time using a linear spline approximation with 100 vertices and a Delaunay triangulation, see Figure 17. Not only is the final error measure twice as large as for a general triangulation, but the resulting vertex placement is also completely different from the result of experiment eight. This shows that a post-processing step of applying a data-dependent triangulation algorithm, see 1.3, would lead to a sub-optimal result, since the vertex placement cannot be changed by the post-processing step.

7. The seventh test case is the function

$$f(x,y) = 2\sum_{i=0}^{2}\sum_{j=0}^{2} \frac{\sin\big((2i+1)x\big)}{2i+1} \cdot \frac{\sin\big((2j+1)y\big)}{2j+1} \quad , \quad x,y \in [0,2\pi],$$

the third-order Fourier approximation of a bivariate square wave, and a linear spline approximation with 50 vertices and a general triangulation, see Figure 18. The number of vertices is too small to capture all details of the function, but the algorithm still finds a decent approximation.

8. The eigth test case is the same function as the seventh, but this time using a linear spline approximation with 250 vertices and a general triangulation, see Figure 19. Due to the increased number of vertices the approximation takes much longer to converge, but the result captures all details of the target function.

9. The ninth test case is a scattered data set consisting of 37,594 vertices, resulting from a laser scan of a Ski-Doo hood and a linear spline approximation with 1,000 vertices and a general triangulation, see Figure 20.

This case shows that our algorithm can be used in surface reconstruction, as long as the source data can be interpreted as a bivariate, scalar valued function. In the general case of a two-manifold surface, the algorithm can be used to approximate locally functional pieces of a given surface, as described in [8].

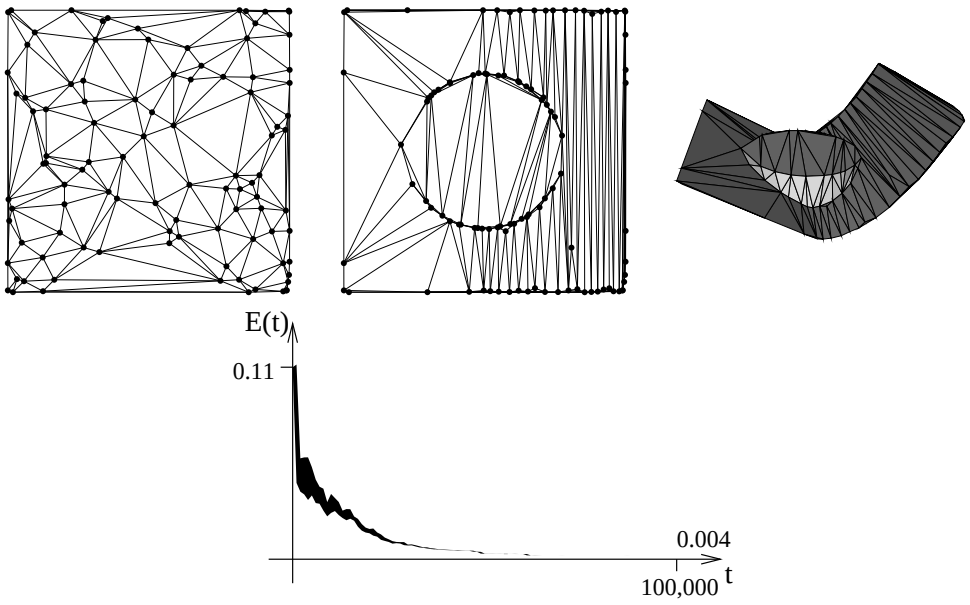

Figure 16: Fifth experiment. Top row: initial and final configurations and flat-shaded rendering; bottom row: error measure over time.

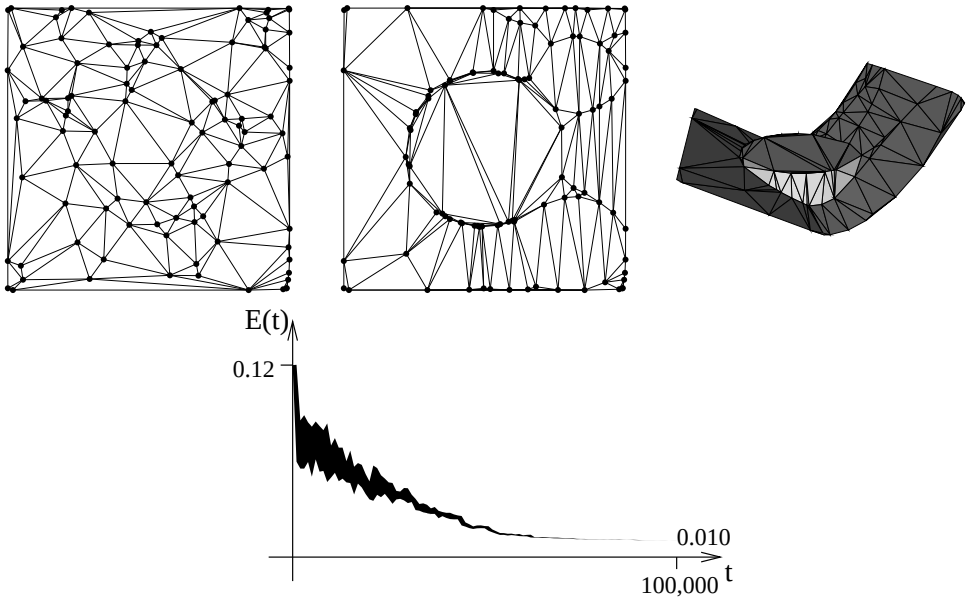

Figure 17: Sixth experiment. Top row: initial and final configurations and flat-shaded rendering; bottom row: error measure over time.

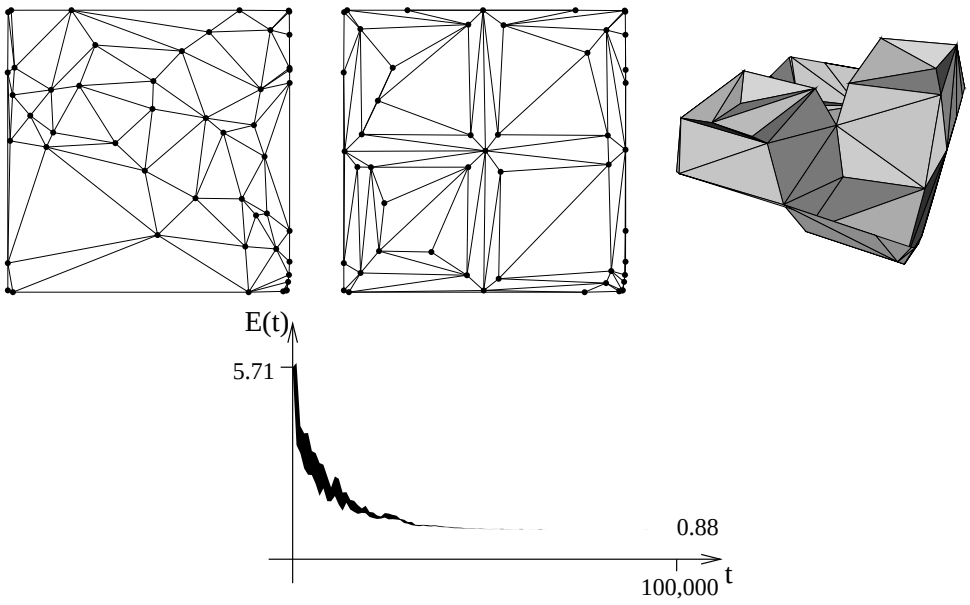

Figure 18: Seventh experiment. Top row: initial and final configurations and flat-shaded rendering; bottom row: error measure over time.

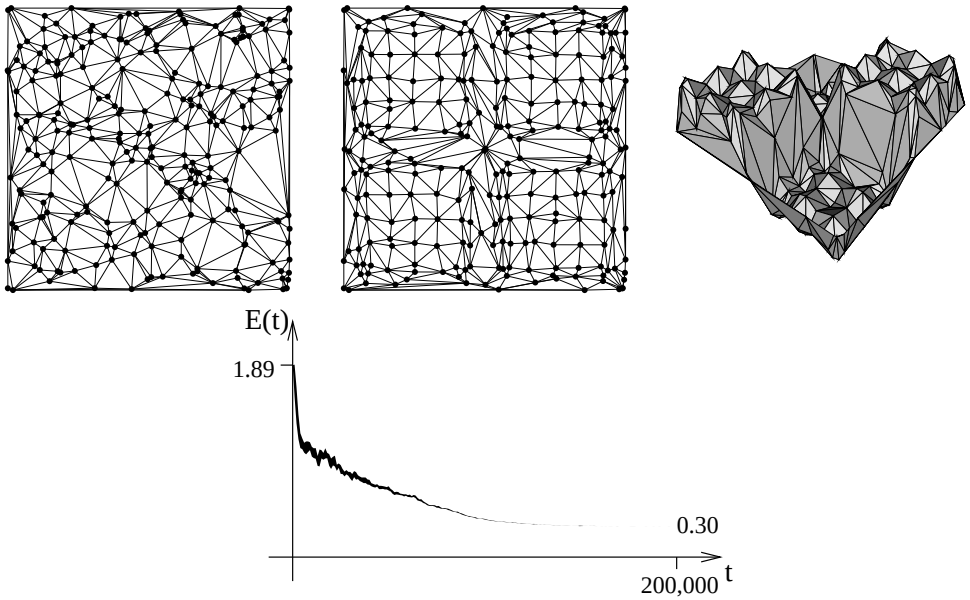

Figure 19: Eigth experiment. Top row: initial and final configurations and flat-shaded rendering; bottom row: error measure over time.

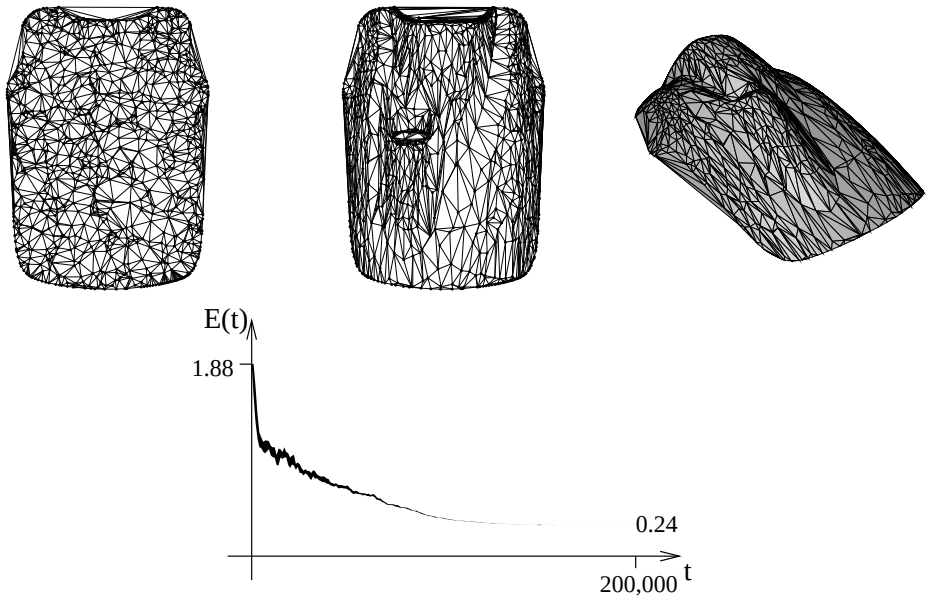

Figure 20: Ninth experiment. Top row: initial and final configurations and flat-shaded rendering; bottom row: error measure over time.

4.3 Bivariate Vector-valued Functions

In this section we apply our method to the approximation of RGB images, interpreting them as bivariate vector-valued functions of the form

$$f\colon \mathbf{R}^2 \to [0,1]^3, \quad (x,y) \mapsto \big(r(x,y), g(x,y), b(x,y)\big) \quad .$$

To judge an approximation's quality, our L^2 error measure algorithm requires a function

$$dist^2\colon \big([0,1]^3\big)^2 \to \mathbf{R}^+ \quad .$$

We define this as the squared Euclidian distance between two color values $c_1 = (r_1, g_1, b_1)$ and $c_2 = (r_2, g_2, b_2)$:

$$dist^2(c_1, c_2) := (r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2 \quad .$$

10. The tenth test case is a photograph of Golden Gate Bridge in San Francisco, resampled to a resolution of 329×222 pixels, see Figure 21, and a linear spline approximation with 400 vertices and a general triangulation, see Figure 22.

11. The eleventh test case is the same function as in the tenth case, but this time approximated by a linear spline with 800 vertices and a general triangulation, see Figure 23. The resulting linear spline is a superset of the result of experiment ten, as defined in section 1.2.

12. The twelvth test case is again the same function, but approximated by a linear spline with 1,600 vertices and a general triangulation, see Figure 24. Again, the resulting linear spline is a superset of the result of experiment eleven. It is hard to see in these low-quality reproductions, but the resulting linear spline approximation is very close to the original image.

Figure 21: A color photograph of Golden Gate Bridge in San Francisco, resampled to a resolution of 329×222 pixels.

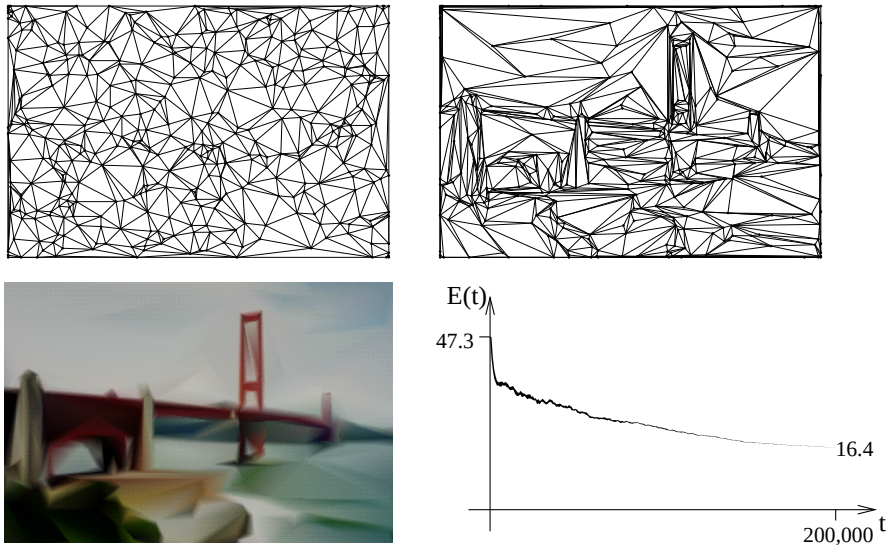

Figure 22: Tenth experiment. Top row: initial and final configurations; bottom row: final approximation and error measure over time.

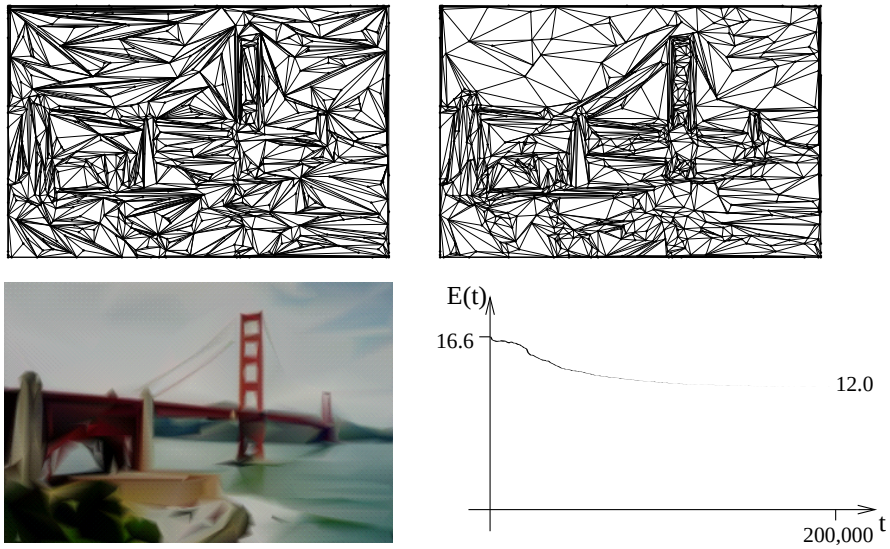

Figure 23: Eleventh experiment. Top row: initial and final configurations; bottom row: final approximation and error measure over time.

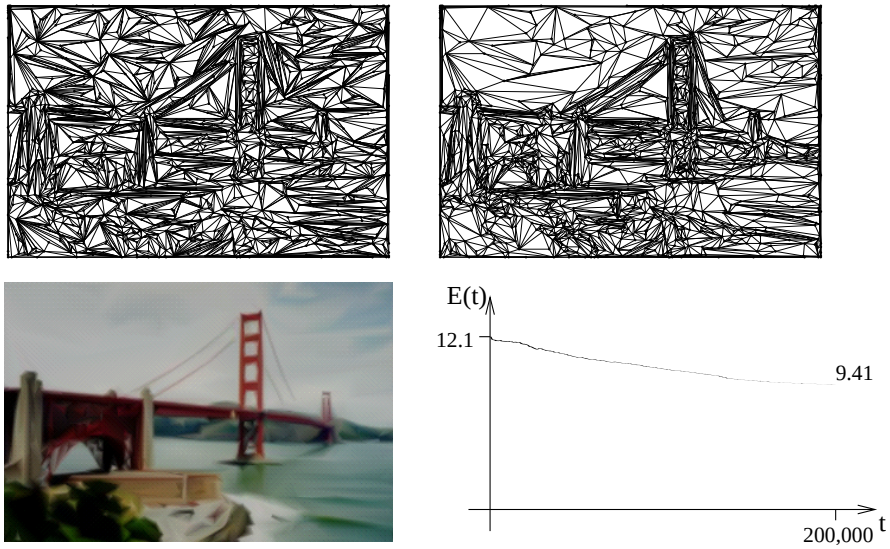

Figure 24: Twelvth experiment. Top row: initial and final configurations; bottom row: final approximation and error measure over time.

5 Conclusions

We have presented a method to calculate optimal linear spline approximations to functions defined by scattered data, using an iterative optimization technique governed by the simulated annealing algorithm. Our method is a generalization of data-dependent triangulation methods.

We have demonstrated that our method performs well for univariate and bivariate scalar-valued functions and for bivariate vector-valued functions. Our method yields good approximations in a reasonable time, and in simple cases it often finds a globally optimal approximation. Furthermore, we have found that our algorithm finds very good approximations to RGB images even when using only a small number of vertices. Our technique provides an interesting alternative way to transform images to a storage-efficient, resolution-independent representation.

6 Future Work

The main areas for future research are the generalization of our algorithm to functions of three and more variables and the application of our method to image and video compression. There is no reason to believe that our method would not work for higher-dimensional functions. The only problems might be representing higher-dimensional linear splines and defining an appropriate iteration step. Even dealing with time-varying d-dimensional data sets is possible; one could either interpret them as $(d+1)$-dimensional functions, or one could calculate independent d-dimensional approximations for the function at discrete times. Taking advantage of time coherence, one could improve the speed of the iteration and the quality of the results by using the final configuration from time t as initial configuration for time $t + \Delta t$. For use of our method in image compression, one would have to investigate methods for efficient storage of linear splines; and one could also research the use of different error measures to achieve more visually pleasing results or special effects like edge enhancement. The remarks about time-varying functions apply to video data as well, and since especially video streams in tele-conferencing exhibit strong frame coherence, our algorithm might lead to a real-time video compression method for this kind of video streams.

7 Acknowledgements

This work was supported by the National Science Foundation under contracts ACI 9624034 and ACI 9983641 (CAREER Awards), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the Office of Naval Research under contract N00014–97–1–0222; the Army Research Office under contract ARO 36598–MA–RIP; the NASA Ames Research Center through an NRA award under contract NAG2–1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level–2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628 awarded to the University of California, Davis. We also acknowledge the support of ALSTOM Schilling Robotics, Chevron, General Atomics, Silicon Graphics, and ST Microelectronics, Inc. We thank the members of the Visualization Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis. This work would not have been possible without the encouragement of Hartmut Prautzsch, Institut für Betriebs- und Dialogsysteme, Fakultät für Informatik, Universität Karlsruhe (TH), Karlsruhe, Germany, and without the generous grant from the Rational Solutions company.

References

[1] Kreylos, O. and Hamann, B. *On simulated annealing and the construction of linear spline approximations for scattered data*, in: Gröller, E., Löffelmann, H. and Ribarsky, W., eds., Proc. "EUROGRAPHICS-IEEE TCCG Symposium on Visualization", Data Visualization '99 (1999), Springer Verlag, Vienna, Austria, pp. 189–198

[2] Bonneau, G.-P., Hahmann, S. and Nielson, G. M., *BLaC-wavelets: A multi-resolution analysis with non-nested spaces*, in: Yagel, R. and Nielson, G. M., eds., Visualization '96 (1996), IEEE Computer Society Press, Los Alamitos, CA, pp. 43–48

[3] Eck, M., DeRose, A. D., Duchamp, T., Hoppe, H., Lounsbery, M. and Stuetzle, W., *Multiresolution analysis of arbitrary meshes*, in: Cook, R., ed., Proc. SIGGRAPH 1995, ACM Press, New York, NY, pp. 173–182

[4] Gieng, T. S., Hamann, B., Joy, K. I., Schussman, G. L. and Trotts, I. J., *Constructing hierarchies for triangle meshes*, IEEE Transactions on Visualization and Computer Graphics 4(2) (1998), pp. 145–161

[5] Hamann, B., *A data reduction scheme for triangulated surfaces*, Computer Aided Geometric Design 11(2) (1994), pp. 197–214

[6] Trotts, I. J., Hamann, B., Joy, K. I. and Wiley, D. F., *Simplification of tetrahedral meshes*, in: Ebert, D. S., Hagen, H. and Rushmeier, H. E., eds., Proc. Visualization '98, IEEE Computer Society Press, Los Alamitos, CA, pp. 287–295

[7] Hamann, B., Jordan, B. W. and Wiley, D. F., *On a construction of a hierarchy of best linear spline approximations using repeated bisection*, IEEE Transactions on Visualization and Computer Graphics 5(1) (1999), pp. 30–46

[8] Hamann, B., Kreylos, O., Monno, G. and Uva, A. E., *Optimal linear spline approximation of digitized models*, in: Proc. "International Conference on Information Visualization '99 (IV '99) – Computer Aided Geometric Design Symposium" (1999), IEEE Computer Society Press, Los Alamitos, CA, pp. 244–249

[9] Heckel, B., Uva, A. E. and Hamann, B., *Clustering-based generation of hierarchical surface models*, in: Wittenbrink, C. M. and Varshney, A., eds., Proc. IEEE Visualization '98 – Late Breaking Hot Topics (1998), IEEE Computer Society Press, Los Alamitos, CA, pp. 41–44

[10] Nielson, G. M., *Scattered data modeling*, IEEE Computer Graphics and Applications 13(1) (1993), pp. 60–70

[11] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C*, 2nd ed. (1992), Cambridge University Press, Cambridge, MA

[12] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E., *Equations of state calculations by fast computing machines*, Journal of Chemical Physics 21 (1953), pp. 1087–1092

[13] Schumaker, L. L. *Computing Optimal Triangulations Using Simulated Annealing*, Computer Aided Geometric Design 10 (1993), pp. 329–345

[14] Delaunay, B. *Sur la sphere vide*, Otdelenie Matematicheskii i Estestvennyka Nauk 7 (1934), Izv. Akad. Nauk SSSR, pp. 793–800

[15] de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O., *Computational Geometry* (1990), Springer-Verlag, New York, NY

[16] Edelsbrunner, H., *Algorithms in Combinatorial Geometry* (1987), Springer-Verlag, New York, NY

[17] Edelsbrunner, H., and Seidel, R., *Voronoï diagrams and arrangements*, Discrete Computational Geometry 1 (1986), 25–44

[18] Preparata, F. P., Shamos, M. I., *Computational Geometry*, third printing (1990), Springer-Verlag, New York, NY

[19] Guibas, L. J., Knuth, D. E., and Sharir, M. *Randomized incremental construction of Delaunay and Voronoï diagrams*, in: Proc. 17th Int. Colloq.—Automata, Languages and Programming, Lecture Notes in Computer Science (LNCS) 443 (1990), Springer Verlag, Berlin, pp. 414–431