Algorithm XXX: DELAUNAYSPARSE: Interpolation via a Sparse Subset of the Delaunay Triangulation in Medium to High Dimensions

TYLER H. CHANG, LAYNE T. WATSON, THOMAS C.H. LUX, ALI R. BUTT,
KIRK W. CAMERON, AND YILI HONG
Virginia Polytechnic Institute and State University

DELAUNAYSPARSE contains both serial and parallel codes written in Fortran 2003 (with OpenMP) for performing medium- to high-dimensional interpolation via the Delaunay triangulation. To accommodate the exponential growth in the size of the Delaunay triangulation in high dimensions, DELAUNAYSPARSE computes only a sparse subset of the complete Delaunay triangulation, as necessary for performing interpolation at the user specified points. This paper includes algorithm and implementation details, complexity and sensitivity analyses, usage information, and a brief performance study.

Categories and Subject Descriptors: G.1.1 [**Numerical Analysis**]: Interpolation — Spline and piecewise polynomial interpolation; J.2 [**Computer Applications**]: Physical Science and Engineering — *Mathematics*; G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms, Computational Geometry, Documentation

Additional Key Words and Phrases: Delaunay triangulation, interpolation, multivariate approximation, high-dimensional data

1. INTRODUCTION

The Delaunay triangulation is an unstructured simplicial mesh that is widely studied in the field of computational geometry. Due to its many favorable properties, the Delaunay triangulation finds wide use as a mesh for multivariate interpolation in the fields of geographic information systems (GIS), civil engineering, physics, and computer graphics. See Section 9.6 of [de Berg et al. 2008] for a brief discussion of the usefulness of Delaunay triangulations. The viability of Delaunay triangulations as a means for interpolating arbitrary nonlinear functions in the context of data science and machine learning has been explored by Omohundro [1990] and more recently by Belkin et al. [2018]. Other recent works have shown

This work was supported in part by NSF Grants DGE-154362, CNS-1565314, and CNS-1838271.
Authors' addresses: T.H. Chang, T.C.H. Lux, A.R. Butt, K.W. Cameron, Department of Computer Science, L.T. Watson, Departments of Computer Science, Mathematics, and Aerospace and Ocean Engineering, Y. Hong, Department of Statistics, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061; e-mail: `thchang@vt.edu`.

Delaunay triangulations to be effective for interpolating real-world computer system data [Chang et al. 2018a and Lux et al. 2018], outperforming several common multivariate interpolation and approximation techniques.

In this work, the main problem of interest is the multivariate interpolation problem. Interpolation via meshes such as triangulations and other tessellations is a classic practice. Delaunay triangulations are widely considered optimal simplicial meshes for many meshing applications, including interpolation. See the first chapter of Cheng et al. [2012] for an overview of Delaunay meshing applications and theory, and see Rajan [1994] for specific theorems on the optimality of Delaunay triangulations in arbitrary dimension.

In two dimensions, the Delaunay triangulation of n points can be efficiently computed in $\mathcal{O}\big(n \log n\big)$ time [Su et al. 1995]. After the Delaunay triangulation has been computed, the cost of evaluating each interpolation point is reduced to the cost of point location. In two dimensions, point location can be performed in $\mathcal{O}\big(n^{\frac{1}{3}}\big)$ time [Mücke et al. 1999], so the total cost of interpolating at m points in two dimensions is $\mathcal{O}\big(n \log n + n^{\frac{1}{3}} m\big)$. However, Klee [1980] showed that in $\mathbf{R}^d$, the worst case size of the Delaunay triangulation is $\Omega\big(n^{\lceil d/2 \rceil}\big)$. In practice, the size of Delaunay triangulations can grow much slower for favorable data sets. In particular, for *well-spaced* point sets, the size of the Delaunay triangulation depends only linearly on n [Sheehy et al. 2008] but may still depend exponentially on d, a phenomenon often associated with *the curse of dimensionality.*

Despite the complexity of high-dimensional Delaunay triangulations, there are currently a wide variety of algorithms for computing them. The first algorithm capable of computing Delaunay triangulations in arbitrary dimension was proposed independently by both Bowyer [1981] and Watson [1981]. Perhaps the most widely used algorithm for computing Delaunay triangulations in arbitrary dimension is the Quickhull algorithm proposed by Barber et al. [1996]. Quickhull is a time-efficient algorithm running in $\Theta\big(n \log n + \kappa\big)$ time, where κ denotes the size of the Delaunay triangulation. Quickhull also boasts a highly optimized numerically stable implementation. An alternative to Quickhull is the graph based algorithm proposed by Boissonnat et al. [2009]. An implementation of this algorithm, contained in the Computational Geometry Algorithms Library (CGAL), stores the Delaunay triangulation in a memory efficient graph structure, at the cost of a slightly greater compute time. One final algorithm of interest is the DeWall algorithm, proposed by Cignoni et al. [1998]. The DeWall algorithm, though not in widespread use, features a unique divide-and-conquer paradigm, and was a major inspiration behind this work.

Due to the exponential growth of Delaunay triangulations in high dimensions, none of the above mentioned algorithms are intended to scale past six or seven dimensions. In fact, this failure to scale to high dimensions is suffered by nearly every mesh based approximation. Consequently, high-dimensional approximation is generally dominated by mesh free methods such as multivariate polynomials, radial basis functions, low order splines, inverse distance weightings, kernel methods, and machine learning techniques such as support vector regressors and artificial neural networks (see [Cheney et al. 2009]). By leveraging the sparse nature of the interpolation problem, the DELAUNAYSPARSE package aims to add the high-dimensional Delaunay mesh based interpolant to the numerical analyst's toolbox.

The rest of this paper is organized as follows. Section 2 describes the Delaunay interpolant in greater detail and outlines a novel algorithm for computing it. Section 3 details the computational aspects of the algorithm, with an emphasis on numerical stability and efficiency. Section 4 describes the serial implementation of the algorithm and its additional features. Section 5 describes the parallel implementation, which uses OpenMP in a shared memory paradigm. Section 6 contains usage information and package organization details. Section 7 shows performance statistics, demonstrating the scalability of the algorithm. For additional information on the properties of Delaunay triangulations and algorithms for computing them, two excellent references are [Aurenhammer et al. 2013] and [Cheng et al. 2012].

2. INTERPOLATION VIA THE DELAUNAY TRIANGULATION

Let P be a set of $n > d$ data points in $\mathbf{R}^d$. A d-dimensional triangulation is defined as a mesh of d-simplices that (1) are disjoint except on their shared boundaries, (2) whose set of vertices is P, and (3) whose union is the convex hull of P, denoted $CH(P)$. The interpolation problem is: given values $f(p)$ for all points $p \in P$ where $f : \mathbf{R}^d \to \mathbf{R}^l$, find an approximation $\hat{f} \approx f$ such that $\hat{f}(p) = f(p)$ for all $p \in P$, where $\hat{f}$ has support in $CH(P)$.

Let $T(P)$ be a d-dimensional triangulation of P. To define an interpolant in terms of $T(P)$, let $q \in CH(P)$ be an interpolation point, and let S be a simplex in $T(P)$ with vertices $s_1, \ldots, s_{d+1}$ such that $q \in S$. Then there exist weights $w_1, \ldots, w_{d+1}$ such that $q = \sum_{i=1}^{d+1} w_i s_i$, $\sum_{i=1}^{d+1} w_i = 1$, and $w_i \geq 0$ for $i = 1, \ldots, d+1$, and the interpolant $\hat{f}_T$ is given by

$$\hat{f}_T(q) = w_1 f(s_1) + w_2 f(s_2) + \ldots + w_{d+1} f(s_{d+1}). \tag{1}$$

In DELAUNAYSPARSE, the interpolant $\hat{f}_{DT}$ is computed, where $DT(P)$ denotes a Delaunay triangulation of P. The Delaunay triangulation is often defined as the geometric dual of the Voronoi diagram, also called the Dirichlet tessellation. Here the following equivalent definition is preferred. For a d-simplex S, let B_S denote the open ball whose center and radius are given by the $(d-1)$-sphere circumscribing S. Then a Delaunay triangulation $DT(P)$ of a finite set of points $P \subset \mathbf{R}^d$ is any triangulation of P such that for each $S \in DT(P)$, B_S satisfies $B_S \cap P = \emptyset$. Remarks 2.1–3 below describe several key properties of a Delaunay triangulation.

Remark 2.1. Given a set of $d + 1$ vertices in P that define a d-simplex S, the condition that $B_S \cap P = \emptyset$ is not only necessary, but also sufficient to conclude that $S \in DT(P)$ for *some* Delaunay triangulation $DT(P)$.

Remark 2.2. Let F be a facet of a simplex $S \in DT(P)$. Let H denote any halfspace whose boundary is the hyperplane containing F. Let $p_1, \ldots, p_\ell$ be a sequence of points that are in $P \cap H$. Define the open circumballs $B_1, \ldots, B_\ell$ such that each B_i circumscribes F and p_i. Assume $p_1, \ldots, p_\ell$ satisfies $p_i \in B_{i+1}$ for all $1 \le i < \ell$. Then $B_1 \cap H \subset B_2 \cap H \subset \cdots \subset B_\ell \cap H$.

Remark 2.3. In randomly generated data, the cases where $DT(P)$ does not exist or is not unique occur with probability zero. Therefore, for algorithmic analysis, it is common to make the simplifying assumption that P is in *general position*, meaning $DT(P)$ exists and is unique. Furthermore, note that the case where $DT(P)$ does not exist occurs only if all the points in P are contained in some lower-dimensional linear manifold. In the context of interpolation, this corresponds to an over parameterization of the underlying function and can be resolved with dimension reduction techniques. The case where $DT(P)$ is not unique can occur in real-world problems and will be addressed in the implementation, discussed in Section 3.

Note that given a set of m interpolation points Q, at most m simplices in $DT(P)$ are needed to compute $\hat{f}_{DT}(q)$ for all $q \in Q$. Therefore, for this particular problem, it is possible to "cheat" the curse of dimensionality when m is significantly less than the size of $DT(P)$. To do so, it suffices to compute a sparse subset of $DT(P)$ such that Q is contained in the subset.

As previously mentioned, one of the major inspirations behind this work was the DeWall algorithm proposed by Cignoni et al. [1998]. The DeWall algorithm features a divide-and-conquer paradigm where construction of each Delaunay simplex is carefully guided so as to construct a "wall" of simplices, bisecting the data set. Each successive simplex is completed from a facet of a previously constructed simplex, using the same methodology as the classic gift-wrapping approach (described in Section 5.6 of [Cheng et al. 2012]).

The two key components of the DeWall and gift-wrapping algorithms that are used in DELAUNAYSPARSE are the growth of the seed simplex and the completion of an open Delaunay facet. After iteratively constructing a seed simplex, the idea is to perform a *visibility walk* to the simplex containing each interpolation point, as described by Devillers et al. [2001]. Since the complete triangulation is never computed, each step of the walk is performed by completing the Delaunay facet designated by the visibility walk protocol. Once each interpolation point q has been located, each response value $\hat{f}_{DT}(q)$ can be computed using (1). A detailed description and analysis of this algorithm is in Section 3 of [Chang et al. 2018b]. Pseudocode for interpolating at a single point follows.

```
Algorithm 1
$P$ contains $n$ $d$-dimensional data points;
$q \in CH(P)$ is the interpolation point;
$S$ denotes the current $d$-simplex;
$F$ denotes the facet of $S$ from which $q$ is visible.
begin Grow an initial seed $d$-simplex $S$, as described in Section 3.1.
```

```
    while $q \notin S$ do
        Select the facet $F$ of $S$ from which $q$ is visible as described in Section 3.2;
        complete a new $d$-simplex $S^*$ from the facet $F$ as described in Section 3.3;
        update $S \leftarrow S^*$.
    enddo
    Since the loop has terminated, $q \in S$. Compute $\hat{f}_{DT}(q)$ using (1).
```

The advantages of this technique are maximized when the interpolation points are sparse with respect to the size of the triangulation. In both expectation and practice, the number of simplices constructed during the walk to each interpolation point is a polynomial function of d and often seemingly independent of n. Given the exponential nature of the problem, this makes for an effective sparse solution, particularly in high dimensions.

2.1 Relationship to Linear Programming

For those experienced in linear programming, Algorithm 1 may seem reminiscent of the Dantzig simplex method for solving linear programs. In fact, computing the Delaunay simplex containing an interpolation point can be connected to linear programming.

Let $P = \{p_1, \ldots, p_n\}$, and let

$$\tilde{A} = \begin{bmatrix} -p_1^T & 1 \\ -p_2^T & 1 \\ \vdots & \vdots \\ -p_n^T & 1 \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} \|p_1\|_2^2 \\ \|p_2\|_2^2 \\ \vdots \\ \|p_n\|_2^2 \end{bmatrix}, \text{ and } \quad \tilde{c} = \begin{bmatrix} -q \\ 1 \end{bmatrix}.$$

Consider the primal and dual linear programs

$$\max_{\tilde{x}} \ \tilde{c}^T \tilde{x} \text{ such that } \tilde{A}\tilde{x} \leq \tilde{b}, \ \tilde{x} \text{ free}, \qquad \min_{\tilde{y}} \ \tilde{b}^T \tilde{y} \text{ such that } \tilde{A}^T \tilde{y} = \tilde{c}, \ \tilde{y} \geq 0.$$

For the primal problem, every extreme point of the feasible set satisfies $\tilde{x} = \left(-2\zeta, \eta^2 - \|\zeta\|_2^2\right)$ where ζ and η are the circumsphere center and radius, respectively, for a simplex in $DT(P)$ (not necessarily containing q). For the dual problem, every extreme point of the feasible set contains a vector of weights expressing q as a convex combination of $d+1$ vertices of a simplex (not necessarily Delaunay) containing q. The optimal solution for both corresponds to a Delaunay simplex containing q. The vertex set for a nondegenerate Delaunay simplex containing q is not immediately given by the solution to the primal or dual problem. Rather, the vertices can be inferred from a nondegenerate basic solution to the dual problem.

When P is in general position, DELAUNAYSPARSE can be interpreted as a different strategy for flipping through simplices than the Dantzig simplex algorithm. For real-world data, which could be degenerate, finding a nondegenerate basic solution is significantly harder than solving the linear program [Megiddo 1991]. By favoring a geometric interpretation of the problem, DELAUNAYSPARSE is robust for degenerate input sets. DELAUNAYSPARSE also avoids significant computational expense and memory burden, which would be introduced through auxiliary variables when reducing the primal problem to canonical form.

3. COMPUTATIONAL ASPECTS

In this section, the computational operations referenced in Algorithm 1 will be fully detailed. These operations are the growth of the seed simplex, the visibility walk, and flipping across a Delaunay facet. Due to floating point error, DELAUNAYSPARSE does not necessarily compute simplices from the true Delaunay triangulation for nearly degenerate P. Instead, the computed simplices are elements of $DT(\hat{P})$, where $\hat{P} \approx P$. The modifications needed to account for floating point error are detailed in the remarks throughout this section, and bounds on the perturbation between P and $\hat{P}$ are given in Section 4.3.

3.1 Growing the Seed Simplex

The seed simplex is constructed through a greedy algorithm, as detailed in Section 3.1 of Chang et al. [2018b]. The initial vertex s_1 is chosen to be the closest point in the data set P to the interpolation point q, and ties are resolved by choosing the point in P with the lowest index. The second vertex $s_2 \in P \setminus \{s_1\}$ is chosen such that

$$\|s_2 - s_1\| = \min_{\substack{p \in P, \\ p \neq s_1}} \|p - s_1\|.$$

Each subsequent vertex is chosen to minimize the radius of the minimum radius circumsphere passing through the resulting vertex set.

For $2 \leq j \leq d$ and $p \in P \setminus \{s_1, \ldots, s_j\}$, define the $j \times d$ matrix

$$A^{(j,p)} = \begin{bmatrix} (s_2 - s_1)^T \\ \vdots \\ (s_j - s_1)^T \\ (p - s_1)^T \end{bmatrix}$$

and the j-vector

$$b^{(j,p)} = \frac{1}{2} \begin{bmatrix} \|s_2 - s_1\|^2 \\ \vdots \\ \|s_j - s_1\|^2 \\ \|p - s_1\|^2 \end{bmatrix}.$$

If rank $A^{(j,p)} = j$, then the minimum norm solution to the under determined system

$$A^{(j,p)} x = b^{(j,p)} \tag{2}$$

is $x^* = c - s_1$, where c denotes the center of the minimum radius circumsphere about $s_1, \ldots, s_j, p$. So, each subsequent vertex s_{j+1} is given by the $p^* \in P \setminus \{s_1,$

$\ldots, s_j\}$ such that solving (2) with $A^{(j,p^*)}$ and $b^{(j,p^*)}$ produces the minimum 2-norm solution x^*.

If rank $A^{(j,p)} < j$, then $s_1, \ldots, s_j, p$ are not the vertices of a j-simplex, and p cannot be a vertex of any d-simplex with vertices $s_1, \ldots, s_j$. Hence, p can be skipped and need not be considered again when constructing the seed simplex. Due to memory constraints, there is no mechanism for marking a p that can be skipped, and hence any such p could be revisited in the future, though it will always be skipped.

If P is in general position, then there will always exist a unique point p^* that minimizes $\|x^*\|$. However, if P lies in a $(j-1)$-dimensional linear manifold where $j \le d$, then any set of $j+1$ or more points in P will be affinely dependent. Therefore rank $A^{(j,p)} < j$ for *all* $p \in P \setminus \{s_1, \ldots, s_j\}$, and no solution p^* can exist. Indeed, $DT(P)$ does not exist as discussed in Remark 2.3. Furthermore, $DT(P)$ is not unique if there exist $d+2$ or more points in P that lie on the same circumball, since there may be more than one $p^* \in P$ that minimize $\|x^*\|$. For the purpose of interpolation, all of these solutions are equally suitable, and the decision between any number of such candidate solutions can be made arbitrarily. In particular, these cases are resolved by choosing the candidate p^* with the lowest index in P. Pseudocode for this process follows.

Algorithm 2

```
$P$ contains $n$ $d$-dimensional data points;
$q \in CH(P)$ is the interpolation point;
$\{s_1, \ldots, s_{d+1}\}$ denote the vertices of the seed simplex.
begin
$s_1 \leftarrow \operatorname{arg\,min}_{p \in P} \|q - p\|$;
$s_2 \leftarrow \operatorname{arg\,min}_{p \in P, p \ne s_1} \|s_1 - p\|$;
for $j = 2, \ldots, d$ do
    initialize $r_{min} \leftarrow \infty$; $p^* \leftarrow null$;
    for all $p \in P \setminus \{s_1, \ldots, s_j\}$ do
        Compute $A^{(j,p)}$ and $b^{(j,p)}$;
        if rank $A^{(j,p)} < j$ then
            Skip this point;
        else if rank $A^{(j,p)} = j$ then
            Compute the minimum norm solution $x^*$ to (2);
            if $\|x^*\| < r_{min}$ then
                $r_{min} \leftarrow \|x^*\|$;
                $p^* \leftarrow p$;
            endif
        endif
    enddo
    $s_{j+1} \leftarrow p^*$;
enddo
```

```
if $p^* \neq null$ then
    return $\{s_1, \ldots, s_{d+1}\}$;
else
    return error (points in a lower-dimensional linear manifold);
endif
```

The dominant costs of the above algorithm are determining the rank of $A^{(j,p)}$ and finding the minimum norm solution x^* to (2). Both of these computations can be done using a LQ factorization of $A^{(j,p)}$. Such a factorization has a computational complexity of at most $\mathcal{O}\big(d^3\big)$ (in the case where $j = d$).

Remark 3.1.1. To determine the rank of $A^{(j,p)}$ using a LQ factorization, consider the final term on the diagonal of L, L_{jj}. From the construction of $A^{(j,p)}$, L_{jj} is the exact distance from p to the $(j-1)$-dimensional linear manifold defined by $\{s_1, \ldots, s_j\}$. To allow for floating point error, p should be at least a distance of ε outside of the manifold. Therefore $A^{(j,p)}$ is considered singular if $|L_{jj}| < \varepsilon$, where ε is a scale/machine dependent constant.

Remark 3.1.2. In each iteration of the inner loop in Algorithm 2 (over all $p \in P \setminus \{s_1, \ldots, s_j\}$), a jth row is appended onto $A^{(j-1,s_j)}$ to construct $A^{(j,p)}$. So, given the LQ factorization of $A^{(j-1,s_j)}$, it is possible to directly compute the minimum norm solution to (2) in $\mathcal{O}\big(d^2\big)$ time by using a rank-1 update. These updates are always applied to the original LQ factorization of $A^{(j-1,s_j)}$, which is full-rank by construction. Therefore there is no risk of compounding numerical error, and the complexity of the inner loop is $\mathcal{O}(nd^2)$.

Remark 3.1.3. In practice, the check described in Remark 3.1.1 is sufficient to avoid flat simplices. Furthermore, because the Euclidean distance from the $(j-1)$-dimensional linear manifold defined by $\{s_1, \ldots, s_j\}$ is used as the criterion for singularity, the check in Remark 3.1.1 is amenable to backward stability in the geometric sense. Specifically, no input point whose Euclidean distance from the manifold is greater than ε is treated as affinely dependent. This is distinctly different from approximating the distance to rank deficiency for the matrix $A^{(j,p)}$. Shroff and Bischof [1992] provide an algorithm for approximating the latter after a rank-1 update at the cost of an additional linear solve. In order to save computational expense (the check described in Remark 3.1.1 comes at no additional expense) and guarantee backward stability in the geometric sense, the simpler check in Remark 3.1.1 is favored here. In pathological cases that are rare in practice, this could result in a matrix $A^{(j,p^*)}$ that is nearly rank deficient in the matrix 2-norm.

Using the rank-1 update described in Remark 3.1.2, the total complexity of Algorithm 2 (for growing a seed d-simplex) is reduced to $\mathcal{O}\big(nd^3\big)$.

3.2 The Visibility Walk

After constructing the seed simplex, DELAUNAYSPARSE advances on the simplex containing an interpolation point q by following a visibility walk. A facet F

of a simplex S is said to be *visible* to q if there exists a point $\rho \in \text{int } S$ such that the line segment drawn from ρ to q intersects F. A visibility walk is a sequence of "flips" that always occur across a facet from which q is visible. Note that each flip in a visibility walk is generally not unique, as it is possible for q to be visible from multiple facets, and a flip across *any* visible facet constitutes a valid step in a visibility walk. Edelsbrunner [1989] showed that in a Delaunay triangulation, every visibility walk is acyclic and therefore must converge for any $q \in CH(P)$. The mechanics of performing each flip in the visibility walk will be detailed in Section 3.3. In this section, the processes of identifying each visible facet and terminating the visibility walk will be explored.

For a simplex S with vertices $s_1, \ldots, s_{d+1}$, define the $d \times d$ matrix

$$A^{(S)} = [\,(s_2 - s_1) \ \ \cdots \ \ (s_{d+1} - s_1)\,].$$

Let x_i denote the ith entry in the d-vector x, given by the solution to the linear system

$$A^{(S)}x = q - s_1. \qquad (3).$$

Then the vector of affine weights $w = [w_1, \ldots, w_{d+1}]^T$ for generating q as a combination of $s_1, \ldots, s_{d+1}$ is given by

$$w = \begin{bmatrix} \left(1 - \sum_{i=1}^{d} x_i\right) \\ x_1 \\ \vdots \\ x_d \end{bmatrix}.$$

If $w_i \geq 0$ for $i = 1, \ldots, d+1$, then $q \in S$ and w contains the interpolation weights in (1). If any $w_i < 0$, then dropping the corresponding vertex s_i leaves the vertices of a facet of S from which q is visible. If S is a valid Delaunay simplex, $A^{(S)}$ is nonsingular and (3) can be solved via LU factorization. In practice, DELAUNAYSPARSE solves (3) using a LQ factorization, which can be reused for performing the "flip" operation, as described in Section 3.3.

Remark 3.2.1. To account for floating point errors, the condition that $w_i \geq 0$ for $i = 1, \ldots, d+1$ should be replaced with $w_i \geq -\varepsilon$, where ε is a scale/machine dependent constant, similarly as in Remark 3.1.1.

The cost of a single LQ factorization for solving (3) is dominated by the cost of performing a flip, as described in the next section, which requires up to n rank-1 updates. However, the total length of the visibility walk (in number of flips) will be important in determining the computational complexity of the DELAUNAYSPARSE algorithm. This length k can only be analytically bounded by the total size of $DT(P)$. However, Bowyer [1981] claimed without proof that when starting a visibility walk from the center of a Delaunay triangulation, k is $\mathcal{O}\big(n^{1/d}\big)$. Mücke et al. [1999] proved that in up to three-dimensions, a stronger claim can be made for some variations of the standard visibility walk. Chang et al. [2018b]

showed empirically that for pseudo-randomly generated data points, when starting from a simplex grown off the nearest data point to q, k tends to grow polynomially with dimension and has no dependence on n for large values of n and d.

3.3 Flipping Across a Facet

Let $F = CH(\{s_1, \ldots, s_d\})$ be a facet of a previously constructed Delaunay simplex from which the interpolation point q is visible. Let $H(F)$ denote the hyperplane containing F, and let $H_q(F)$ denote the open halfspace (with respect to $H(F)$) that contains q. The goal of this section is to "flip toward" q, by constructing a new Delaunay d-simplex with vertices $s_1, \ldots, s_{d+1}$, where $s_{d+1} \in P \cap H_q(F)$.

Recall from Remark 2.1 that a d-simplex S is Delaunay if and only if $B_S \cap P = \emptyset$. Since at least one Delaunay simplex (of which F is a facet) has already been constructed, $DT(P)$ exists. Therefore, if F is **not** a facet of $CH(P)$, there must be at least one point $p^* \in P \cap H_q(F)$ such that the simplex S^* with vertices $\{s_1, \ldots, s_d, p^*\}$ is Delaunay, satisfying $B_{S^*} \cap P = \emptyset$. If no such p^* exists, then it can be inferred that $q \notin CH(P)$. So, to perform a "flip" to a new Delaunay simplex closer to q, it suffices to check inside the circumball of the simplex with vertices $s_1, \ldots, s_d, p$, for each $p \in P \cap H_q(F)$. By exploiting the property described in Remark 2.2, this can be done in a single pass over P.

For a facet F with vertices $s_1, \ldots, s_d$, let the $(d-1) \times d$ matrix

$$A^{(F)} = \begin{bmatrix} (s_2 - s_1)^T \\ \vdots \\ (s_d - s_1)^T \end{bmatrix}.$$

To obtain a normal to $H(F)$, it suffices to take any nontrivial vector in the nullspace of $A^{(F)}$. Since F is a Delaunay facet, rank $A^{(F)} = d - 1$. Therefore, using a LQ factorization with row pivoting, $\mathcal{P}A^{(F)} = LQ$, the unit normal to $H(F)$ is given by the last row of Q, $v^T = Q_{d\cdot}$.

Given the normal vector v for $H(F)$, consider the function

$$\sigma_F(p) = \operatorname{sgn}\big((p - s_1)^T v\big). \tag{4}$$

For each $p \in P$, $p \in H_q(F)$ if and only if $\sigma_F(p) = \sigma_F(q)$.

Remark 3.3.1. To account for floating point errors, the additional condition that $\big|(p - s_1)^T v\big| > \varepsilon$ should be imposed, where ε is a scale/machine dependent constant, similarly as in Remarks 3.1.1 and 3.2.1. Note that similarly as in Remark 3.1.1, this is a geometric distance threshold and does not guarantee that the resulting matrix $A^{(d,p)}$ is far from singular for pathological point sets.

Consider now those $p \in P \backslash \{s_1, \ldots, s_d\}$ such that $\sigma_F(p) = \sigma_F(q)$, i.e., $p \in H_q(F)$. Similarly as in Section 3.1, the center of the circumball about F and p is given by

$c = x^* + s_1$, and the radius of the circumball is given by $\|x^*\|$, where x^* is a solution to the system

$$A^{(d,p)}x = b^{(d,p)}. \tag{5}$$

Since F is a valid Delaunay facet, the first $d-1$ columns of $A^{(d,p)}$ must be linearly independent. Furthermore, if the condition described in Remark 3.1.1 has been satisfied, then it is safe to assume that $A^{(d,p)}$ is full-rank, and (5) has a unique solution. Then p^* is any point in $P \cap H_q(F)$ that satisfies $B_{\|x^*\|}(c) \cap P = \emptyset$, where $B_{\|x^*\|}(c)$ denotes the open ball centered at c with radius $\|x^*\|$. Pseudocode for this entire process follows.

Remark 3.3.2. If P is in general position, then p^* is unique. If there exist $d+2$ or more cospherical points in P, then p^* may not be unique. However, any $p^* \in P \cap H_q(F)$ that satisfies $B_{\|x^*\|}(c) \cap P = \emptyset$ can be chosen in union with $\{s_1, \ldots, s_d\}$ to form the vertex set for a valid simplex in *some* Delaunay triangulation. Such situations can be resolved by choosing the p^* with the greatest index in P.

```
Algorithm 3
$P$ contains $n$ $d$-dimensional data points;
$q \in CH(P)$ is the interpolation point;
$\{s_1, \ldots, s_{d+1}\}$ denote the vertices of the new simplex;
$F$ is the current Delaunay facet;
begin $\{s_1, \ldots, s_d\}$ are given by the vertices of $F$;
Compute the $LQ$ factorization $\mathcal{P}A^{(F)} = LQ$ and set $v^T \leftarrow Q_{d\cdot}$;
Compute $\sigma_F(q)$ using (4);
initialize $r_{min} \leftarrow \infty$; $c_{min} \leftarrow \vec{0}$; $p^* \leftarrow null$;
for all $p \in P \setminus \{s_1, \ldots, s_d\}$ do
    if $\sigma_F(p) \neq \sigma_F(q)$ or $|(p - s_1)^T v| < \varepsilon$ then
        Skip this point;
    else if $\|p - c_{min}\| < r_{min}$ then
        Update $A^{(d,p)}$ and $b^{(d,p)}$ and compute $x^*$, the solution to (5);
        $r_{min} \leftarrow \|x^*\|$;
        $c_{min} \leftarrow x^* + s_1$;
        $p^* \leftarrow p$;
    endif
enddo
if $p^* = null$ then
    return error, $q \notin CH(P)$;
else
    return $s_{d+1} \leftarrow p^*$;
endif
```

The dominant cost for Algorithm 3 is repeatedly solving (5). Since $A^{(d,p)}$ is always full-rank, each instance of (5) could be solved using a LU factorization, with computational complexity $\mathcal{O}(d^3)$. However, similarly as in Remark 3.1.2, a rank-1 update is applied to the LQ factorization of $A^{(F)}$ instead, reducing the complexity per iteration of the loop to $\mathcal{O}(d^2)$. So the worst-case complexity of Algorithm 3 is $\mathcal{O}(nd^2)$. Note that Algorithm 3 is called once in each iteration of the simplex walk. Therefore, from Section 3.1 and 3.2, the overall computational complexity for locating a single interpolation point is $\mathcal{O}(nd^3 + knd^2)$, where k is the length of the visibility walk. Since k is typically much greater than d, this can be simplified to $\mathcal{O}(knd^2)$.

4. SERIAL IMPLEMENTATION

The serial subroutine `DELAUNAYSPARSES` is implemented in ISO Fortran 2003. For efficient numerically stable linear algebra, `DELAUNAYSPARSES` uses LAPACK [Anderson et al. 1999].

4.1 Handling Multiple Interpolation Points

The serial subroutine `DELAUNAYSPARSES` performs interpolation at m points $Q = \{q_1, \ldots, q_m\}$ using Algorithms 1–3, as described in Sections 2 and 3. By default, `DELAUNAYSPARSES` will perform these interpolations sequentially with no modification to Algorithms 1–3. However, an optional argument can be set to "daisy chain" the visibility walks, i.e., for the ith interpolation point q_i where $i > 1$, the last constructed Delaunay simplex (typically the simplex containing q_{i-1}) is used as the first simplex for walking to q_i, replacing Algorithm 2. In general, this behavior can greatly increase the length of each visibility walk and is not recommended. However, if the interpolation points Q are tightly clustered in a relatively small region of $DT(P)$ or if the size of $DT(P)$ is relatively small, this behavior can slightly improve performance by avoiding the expense of Algorithm 2.

Additionally, note that after computing the LU factorization of $A^{(S)}$ for solving (3), it requires relatively little additional computation to check whether S contains any future interpolation points. Thus if S is a simplex with vertices $s_1, \ldots, s_{d+1}$ that has been constructed during the visibility walk to an interpolation point q_i, `DELAUNAYSPARSES` will check whether $q_j \in S$, where $i < j \leq n$ and q_j has not already been found in some simplex of $DT(P)$, by solving

$$A^{(S)}x = q_j - s_1, \tag{6}$$

and using the same criterion described in Section 3.2. If any $q_j \in S$, then $s_1, \ldots, s_{d+1}$ and the corresponding interpolation weights for (1) are saved, and q_j is marked as found and will not be considered again. Because of its cost effectiveness, this behavior is always active during an execution of `DELAUNAYSPARSES`. However, it is most effective for tightly clustered interpolation points.

4.2 Extrapolation

Often, it is reasonable to make predictions for extrapolation points that are *slightly* outside $CH(P)$. In these cases, the most reasonable solution is to project each extrapolation point onto $CH(P)$ and interpolate the projection, provided the residual is small. Let z be an extrapolation point, and let W be a $d \times n$ matrix whose columns are points in P. Then the projection $\hat{z}$ of z onto $CH(P)$ is given by $\hat{z} = Wx^*$, where x^* is the solution to the linearly constrained least squares problem

$$\min_{x \in \mathbf{R}^n} \|Wx - z\| \quad \text{subject to} \quad x \geq 0 \quad \text{and} \quad \sum_{i=1}^{n} x_i = 1. \tag{7}$$

Hanson et al. [1982] provide an efficient solution to (7) based on a slack variable formulation. The most recent version of their subroutine `DWNNLS` is available in the SLATEC software package, and included with the DELAUNAYSPARSE files.

Remark 4.2.1. Note that the visibility walk described in Section 3.2 is only guaranteed to converge for $q \in CH(P)$. In particular, if a projection $\hat{z}$ is left within floating point error of $CH(P)$, and the matrix $A^{(S)}$ for a nearby Delaunay simplex has smallest singular value $\mathcal{O}(\varepsilon)$, then it is possible for a visibility walk to fail by repeatedly calling for a flip that would lead outside of the convex hull. This is an extremely rare situation. However, in these situations, `DELAUNAYSPARSE` will first try to flip in other potential directions (i.e., by dropping different negatively weighted vertices). Then, if no "good" direction can be found, the correct interpolation weights must ultimately be computed by a second `DWNNLS` projection onto the current simplex.

Given the above solution, the residual is given by $r = \|z - \hat{z}\|$. When r is small with respect to the scale of the data, it is reasonable to perform extrapolation at z by interpolating at $\hat{z}$. However, when r is large it is impossible to make any reasonable prediction for $f(z)$. By default, when `DELAUNAYSPARSES` encounters an extrapolation point z, it computes the projection $\hat{z}$ and residual r using `DWNNLS`. If r is smaller than some percentage of the diameter of P, `DELAUNAYSPARSES` resumes interpolation using $q = \hat{z}$. If r is greater than that percentage of the diameter, the extrapolation point z is skipped and an appropriate error is returned.

By default, the threshold for extrapolation is 10% of the diameter of P, but this percentage can be adjusted using an optional input argument. Furthermore, setting this optional value to 0% of the diameter of P will short-circuit the extrapolation process, preventing $\hat{z}$ and r from ever being computed and preventing any `DWNNLS` work arrays from being allocated. Note that the time and space demands of `DWNNLS` can be significantly greater than those of `DELAUNAYSPARSES`. So, for large problems or in cases where computational resources are limited, it is often appropriate to turn off extrapolation by setting the extrapolation threshold to 0% of the diameter of P.

Remark 4.2.2. For similar reasons as in Remark 4.2.1, it is possible that for poorly spaced data points P, `DELAUNAYSPARSES` may incorrectly call for a projection of an interpolation point that is within floating point error of the boundary of $CH(P)$. After unnecessarily performing the projection and finding that $r = 0$, such situations are easily detected retrospectively. However, if the extrapolation threshold is set to 0% of the data diameter, the projection will short circuit and an interpolation point that is within ε of the convex hull $CH(P)$ could be incorrectly skipped. The conditions that could lead to such an error are pathological, but might still occur.

4.3 Data Scaling and Sensitivity Analysis

Recall from Remarks 3.1.1, 3.2.1, 3.3.1, and 4.2.1 that a small scale/machine dependent constant $\varepsilon > 0$ is used to account for floating point error. Affine operations do not affect the Delaunay triangulation or interpolation results, so to account for scaling, `DELAUNAYSPARSES` rescales and shifts the data points P and the interpolation points Q on input. First, the points in P are shifted so that their barycenter is at the origin, then they are rescaled so that they are contained in the unit ball. This ensures that ε can be chosen without accounting for data scale. The interpolation points Q must then be shifted and scaled by the same amounts to maintain relative positions.

After scaling, the default value ε can be chosen based only on machine precision. By default, $\varepsilon = \sqrt{\mu}$ where μ denotes the unit round-off. This is the minimum appropriate value of ε for most applications, and an optional argument can be used to increase ε where appropriate.

The purpose of ε (as described in Remarks 3.1.1, 3.2.1, and 3.3.1) is to guarantee robustness. Remarks 3.1.1 and 3.3.1 together guarantee that no simplex can be constructed whose vertex set is nearly coplanar. Remark 3.2.1 guarantees that any simplex that is within a small perturbation of containing the interpolation point will be accepted. Together, these remarks guarantee that a nondegenerate simplex is found such that the interpolation point is nearly contained in that simplex, even for degenerate and nearly degenerate data sets. Considering these remarks along with the standard floating point error, the computed Delaunay simplices are actually elements of a perturbed triangulation, $DT(\hat{P})$.

An appropriate value of ε should be larger than the backward error due to floating point error (in practice $\sqrt{\mu}$ is enough). Since Remarks 3.1.1 and 3.3.1 use ε as a geometric distance threshold, $\|p_i - \hat{p}_i\|_2 < \varepsilon$ for all $\hat{p}_i \in \hat{P}$ corresponding to $p_i \in P$. These conditions can be interpreted as backward stability guarantees for the rescaled problem.

4.4 Memory Usage

`DELAUNAYSPARSES` uses assumed-shape arrays where appropriate. To ensure expected behavior, the dimensions of each dummy array are checked against user-specified values of d, n, and m on input. Due to the size of $DT(P)$ in high dimensions, the space complexity of any Delaunay triangulation algorithm is equally as important as its time complexity. The computational operations described in Section 3 do not require any work arrays larger than $\mathcal{O}(d^2)$, making DELAUNAYSPARSE a space efficient algorithm.

However, to take full advantage of LAPACK code optimizations, one larger work array is required. Therefore, `DELAUNAYSPARSES` uses one allocatable work array, whose size is determined at runtime based on LAPACK queries. Other allocatable work arrays of size $\mathcal{O}(nd)$ are required by `DWNNLS` for extrapolation, but are only allocated if an extrapolation is performed.

4.5 The Cost of Robustness and Correctness

`DELAUNAYSPARSES` is designed to be robust for a wide variety of use cases and usage errors. In particular, `DELAUNAYSPARSES` uses the diameter of P to judge extrapolation residuals, as discussed in Section 4.2. Also, the minimum pairwise distance between points in P is used to catch bad inputs, since after rescaling, any two points that are closer than ε will be indistinguishable from the perspective of `DELAUNAYSPARSES` and could cause hard to find bugs. The computation of the diameter and minimum pairwise distance is performed while the points are being rescaled, as discussed in Section 4.3. The computational complexity of these distance computations is $\mathcal{O}(n^2d)$. Recall that the computational complexity of the DELAUNAYSPARSE algorithm is $\mathcal{O}(knd^2)$, where k is independent of n for uniformly spaced P. So the cost of interpolating at m points is $\mathcal{O}(knmd^2)$. Therefore in situations where n^2d is significantly larger than $knmd^2$, the complexity of `DELAUNAYSPARSES` can be dominated by nonessential distance computations, used only for robustness and extrapolation checks.

Since these computations are not necessary for the DELAUNAYSPARSE algorithm, it is possible to "turn off" exact extrapolation and error checking by setting an optional input argument. When d is relatively small and $m < n$, this can greatly improve the time complexity of the algorithm. However, doing so uses a diameter approximation of twice the distance from the barycenter of P to the farthest point, which could be off by up to a factor of two. Additionally, it will be possible for duplicate points to go undetected, causing difficult to find bugs and irregularities in results. For this reason, it is recommended that users only switch off these computations once they have already carefully cleaned their data of duplicate points and when the extrapolation cutoff is flexible.

5. PARALLEL IMPLEMENTATION

The parallel subroutine `DELAUNAYSPARSEP` is based on the serial subroutine `DELAUNAYSPARSES` and shares the implementation details discussed in Sections 4.1–5. `DELAUNAYSPARSEP` uses OpenMP 4.5 [OpenMP ARB, 2015] to implement a shared memory paradigm. It is also possible to achieve distributed parallelism by breaking up the interpolation points Q into β batches $Q = Q_1 \cup \ldots \cup Q_\beta$, then distributing these batches Q_i across available nodes (along with copies of the data points P). Each batch can then be evaluated independently on its corresponding node.

Remark 5.1. Recall from Section 4.1 that code optimizations for handling multiple interpolation points are most effective when the points are clustered in $DT(P)$.

Therefore, optimal distributed memory performance is achieved when each Q_i represents a cluster of interpolation points from Q. Note that clustering is an open problem, and the above described distributed memory parallelism can be implemented trivially using separate calls to `DELAUNAYSPARSES` or `DELAUNAYSPARSEP`. Therefore, a distributed implementation with clustering of Q is not provided, and left entirely to the user.

For the OpenMP shared memory implementation, two levels of parallelism are targeted. The first level of parallelism is the loop over all m interpolation points Q. The second level of parallelism applies to the various loops over all n data points P and the loop over all unresolved interpolation points, as computed by (6) and described in Section 4.1. The details for exploiting the first and second levels of parallelism are given in Sections 5.1 and 5.2, respectively. There is also a loop over the n data points for computing the scale factor (as discussed in Section 4.3) and a pair of nested loops over the n data points for computing the diameter and minimum pairwise distance of P (as discussed in Section 4.5). These loops can be parallelized independently of either level of parallelism using a static scheduler.

If m is small with respect to the number of available processors, level one parallelism will not saturate the available computational resources. In the extreme case where $m = 1$, level one parallelism is not available. However, when available, level one parallelism is significantly more efficient than level two parallelism. Therefore, the default behavior for `DELAUNAYSPARSEP` is to exploit level one parallelism whenever it is available (if $m > 1$), and to exploit level two parallelism otherwise (if $m = 1$). If this is not the desired behavior, the type of parallelism can be set manually via an optional argument, and for advanced users, both levels can be activated at the same time resulting in nested parallelism.

Remark 5.2. In order for OpenMP to apply nested parallelism, the environment variable `OMP_NESTED` must be set to `TRUE`. Furthermore, note that if a team of t_1 threads is deployed at the first level and a team of t_2 threads is deployed at the second level, then a total of $t_1 \cdot t_2$ threads could be active at any time. t_1 and t_2 can be set by assigning the environment variable `OMP_NUM_THREADS=`t_1,t_2.

5.1 Level 1 Parallelism

The first level of parallelism is the loop over m interpolation points in Q: **for all** $q \in Q$ **do**, from Algorithm 1. Since the variation in the length k of each visibility walk could be large, `DELAUNAYSPARSEP` uses a dynamic scheduler with a chunk size of one to parallelize this Q loop. The only dependencies between iterations of the Q loop occur when implementing the code optimizations described in Section 4.1.

First, when "checking ahead" for future interpolation points $q_j \in Q$, there is a possible race condition since another thread could already be performing a visibility walk toward q_j. Since these dependencies are minimal, an OpenMP `CRITICAL` lock is used with minimal modification to the serial code to ensure safe sequentially consistent memory accesses. Once any thread of `DELAUNAYSPARSEP` has begun constructing the first simplex in the walk toward q_j, no other threads will continue to test q_j when "checking ahead." Additionally, if daisy chaining is activated, each thread in the team must maintain a private copy of the seed simplex. Therefore, only previously constructed simplices of the active thread are considered for seeding future visibility walks.

Other than the minor issues discussed above, level one parallelism is dependency free and dynamically load balanced. Therefore, under ideal conditions, `DELAUNAYSPARSEP` is capable of weak scaling with respect to the problem dimension m, with negligible overhead.

5.2 Level 2 Parallelism

The second level of parallelism applies primarily to the various loops over n data points in P: **for all** $p \in P \setminus \{s_1, \ldots, s_j(s_d)\}$, as appear in Algorithms 2 and 3. Note that these P loops are not totally free of dependencies. Therefore, to parallelize the P loops, private copies of certain variables (such as r_{min} and p^*) must be maintained. Then, after completing these loops in parallel and producing private solutions, a reduction can be done to determine the global solutions. Note that this will result in some redundant computations. There is relatively little room for performance variation between iterations of the P loops, so `DELAUNAYSPARSEP` parallelizes them with a static scheduler and a fixed chunk size of $\lceil n/t_2 \rceil$, where t_2 denotes the number of threads in each level two team.

The one exception to the above methodology is the loop over all future interpolation points, as described in Section 4.1. This loop is dependency free and can be parallelized using a static scheduler with a fixed chunk size of $\lceil (m-i)/t_2 \rceil$, where i is the index of the current interpolation point. However, if level one parallelism is active, parallelizing this loop results in significant conflict. Therefore, in the case of nested (level one and two) parallelism, this loop executes serially within each level one thread.

Remark 5.2.1 There is no true dependency between iterations of the above loop over remaining interpolation points. However, the OpenMP `CRITICAL` directive used in Section 5.1 locks code segments, as opposed to variable addresses and cannot distinguish between loop iterations, inducing a "false" conflict. As mentioned above, when level one parallelism is available, it is recommended that all available threads be devoted to level one parallelism. Therefore, in the recommended use case, this loop would not offer significant parallelism, and serializing it is no significant loss.

Due to interloop dependencies, exploiting level two parallelism can significantly increase the total number of computations performed by DELAUNAYSPARSE. Furthermore, there are significant regions of serial code separating each level two parallel block. So, the parallel efficiency of `DELAUNAYSPARSEP` with level two parallelism can be poor.

6. ORGANIZATION AND USAGE INFORMATION

The physical organization of the DELAUNAYSPARSE package is described in its included README file. Most notably, DELAUNAYSPARSE is distributed with a copy of the REAL_PRECISION module (from HOMPACK90, ACM TOMS Algorithm 777), for approximately 64-bit arithmetic on all known machines. For completeness, all required LAPACK and BLAS subroutines are included, along with `DWNNLS` and all its dependencies from the SLATEC library. The included copies of the SLATEC subroutines have been updated in accordance with the Fortran 2003 standard. Additionally, legacy implementations for two of `DWNNLS`'s BLAS dependencies (`DROTM` and `DROTMG`) have been included under different names. Finally, sample main programs for the serial and parallel versions illustrating the use of the optional arguments have been included. Sample data sets for these main programs are real data sets (with points not in general position) from the VarSys project on high performance computing system performance variability [Cameron et al. 2019].

The master module DELSPARSE_MOD includes the REAL_PRECISION module and interface blocks for both `DELAUNAYSPARSES` and `DELAUNAYSPARSEP`, as well as an interface block for the updated subroutine `DWNNLS`, which may be of separate interest. Note that by default, `DELAUNAYSPARSES` and `DELAUNAYSPARSEP` do not actually compute the Delaunay interpolant, but return the containing simplex and weights for computing (1). This behavior was chosen to accommodate a wide variety of use cases, including those where the function values $f(x)$ cannot be expressed as real-valued vectors (e.g., when $f(x)$ is a function $g(\omega; x)$ parameterized by x). When the values $f(x)$ *can* be expressed as real-valued vectors, an optional input argument can be supplied with the response values, and then an optional output argument must appear to collect interpolation results. Additionally, recall that P and Q are shifted and rescaled on input. So, if the original P and Q are needed, copies should be made prior to execution.

7. PERFORMANCE

This section summarizes previous results about the approximation accuracy of Delaunay interpolation and presents new results on the runtime performance of the DELAUNAYSPARSE software package.

7.1 Approximation Accuracy

The approximation accuracy of the Delaunay interpolant and other simplicial interpolation schemes is analyzed in Belkin et al. [2018], Liu and Yin [2019], Lux et al. [2019], and Regis [2014]. In particular, Lux et al. [2019] gives an error bound when using $T(P)$ to interpolate a once differentiable scalar function g with γ-Lipschitz continuous gradient in the 2-norm. Consider an interpolation point $q_0 \in \mathbf{R}^d$, whose containing simplex is $S_0 \in T(P)$. Suppose S_0 contains p_0 in its vertex set, has a maximum edge length of ξ, and has a barycentric transformation matrix whose smallest singular value is λ_d. Then

$$|g(q_0) - \hat{g}_T(q_0)| \leq \frac{\gamma \|q_0 - p_0\|_2^2}{2} + \frac{\sqrt{d}\gamma\xi^2}{2\lambda_d}\|q_0 - p_0\|_2.$$

Since the Delaunay property tends to minimize circumball radii [Rajan 1994], it follows that $DT(P)$ is optimal for interpolation in comparison with other triangulations when λ_d is bounded away from zero.

Lux et al. [2019] provides a detailed comparison between Delaunay interpolation and other approximation techniques for a wide variety of high-dimensional data science problems. The approximation accuracy of the Delaunay interpolant has also been empirically evaluated for interpolating computer system performance data by Chang et al. [2018a] and Lux et al. [2018].

7.2 Runtime Performance

This section will focus on the runtime performance of `DELAUNAYSPARSES` and `DELAUNAYSPARSEP`. For reference, first consider Table I, which presents performance data (as reported by Boissonnat et al. [2009]) in up to six dimensions for computing the complete Delaunay triangulation of uniform randomly distributed data points in the unit cube using Quickhull [Barber et al. 1996] and the graph based algorithm proposed by Boissonnat et al. [2009]. Boissonnat et al. gathered this data using a 2.6 GHz Intel processor with 6MB of level 2 cache and 4 GB of DDR2 RAM. The purpose of including Table I is not for direct comparison, as the problem of computing the complete Delaunay triangulation is significantly harder than that of locating a single interpolation point. Indeed, recall that standard Delaunay triangulation algorithms are not capable of scaling past six or seven dimensions for sufficiently large problems. Rather, this data is intended to clarify the issues addressed by DELAUNAYSPARSE, and inform users on when DELAUNAYSPARSE is an appropriate choice over algorithms that compute the complete Delaunay triangulation.

Table I. Time and space requirements (as reported in Boissonnat et al. [2009]) for computing the complete Delaunay triangulation using Quickhull and the Delaunay Graph algorithm. Entries containing the word "swap" indicate that the process exceeded RAM limitations.

Problem Sizes (d & n):		Algorithms:	
		Quickhull	Delaunay Graph
5 dimensions, 2,000 points	time	3.2 sec.	58 sec.
	space	52 MB	10.1 MB
5 dimensions, 32,000 points	time	76 sec.	1463.46 sec.
	space	973 MB	106 MB
6 dimensions, 32,000 points	time	swap	28,296 sec.
	space	swap	267 MB

To test the performance of `DELAUNAYSPARSES`, runtimes have been gathered on AMD CPUs @2.3 GHz. Table II presents runtimes for interpolating at a single interpolation point (the center of the unit hypercube) using `DELAUNAYSPARSES`, for various problem sizes (n) and dimensions (d) with uniform randomly distributed data in the unit hypercube. To account for performance variance, each runtime represents an average over 20 independent runs of `DELAUNAYSPARSES`, each with a different data set of the same size and dimension. Note that in the higher dimensions, the data points (P) are extremely sparse, even for large values of n. For such problems, it is typical to employ some intelligent experimental design. Therefore, in the higher dimensions, the uniform randomly spaced data used for testing becomes increasingly unrepresentative of real-world data. However, this data is sufficient for discussing how the runtime of `DELAUNAYSPARSES` scales with n and d and is comparable to the data used to generate Table I. Since extrapolation presents additional computational complexities, any data set that does not contain the interpolation point ($q = [0.5, 0.5, \ldots, 0.5]^T$) in its convex hull is discarded and regenerated (an unlikely occurrence for the problem sizes shown). Note that the distance computations discussed in Section 4.5 cause an overall computational complexity of $\mathcal{O}(n^2)$ in the lower dimensions. However, in higher dimensions, the cost of the DELAUNAYSPARSE algorithm dominates, and the runtimes approach linear growth with respect to the number of data points n, as predicted in Section 3.

Table II. Serial runtimes (in seconds) for computing the Delaunay interpolant at a single interpolation point. Values shown represent the average over 20 independent trials with n pseudo-randomly generated data points in the d-dimensional unit hypercube.

	Problem dimension (d)				
Problem size (n)	2	8	32	64	128
250	0.005	0.013	0.150	3.404	27.078
500	0.021	0.042	0.325	6.479	59.511
1000	0.083	0.152	0.791	14.020	124.320
2000	0.344	0.583	2.230	28.984	242.066
4000	1.314	2.284	7.165	62.494	502.620
8000	5.580	9.027	26.210	151.177	905.711
16,000	22.086	35.725	109.448	386.596	2190.362
32,000	82.915	145.115	421.934	1097.060	5024.675

In general, users should expect these costs to grow linearly with m, the number of interpolation points. However, in the case where all the interpolation points are tightly clustered, the cost for interpolating at m points can be bounded by a constant times the cost for interpolating at a single point. See Tables 2 and 3 in Chang et al. [2018b]. The improved scaling for clustered interpolation points is directly caused by the code optimizations introduced in Section 4.1.

To compare the performance of `DELAUNAYSPARSEP` at all levels of parallelism against the performance of `DELAUNAYSPARSES`, runtimes have been gathered over a cluster of eight NUMA nodes with four AMD cores per node @2.3 GHz (each core identical to when timing `DELUNAYSPARSES`). Figures 1–3 plot the average elapsed wallclock time (in seconds) for 20 independent runs of `DELAUNAYSPARSEP` against the number of cores used by OpenMP. The data for these experiments was generated using a randomized Latin hypercube design, as described in Amos et al. [2014].

Then the interpolation points were generated from random convex combinations of $d + 1$ randomly selected points from the design.

Figure 1 presents runtimes and parallel speedup factors for interpolating at 1024 points in a 10-dimensional design with 1000 data points, reflecting workloads where m is large. Figure 2 presents runtimes and parallel speedup factors for interpolating at 64 points in a 10-dimensional design with 20,000 data points, reflecting workloads where n is large. Figure 3 presents runtimes and parallel speedup factors for interpolating at 64 points in a 50-dimensional design with 500 data points, reflecting workloads where d is large. For nested parallel runs with four, eight, 32 total active cores, the number of level one (two) threads is two, four, eight (two, four, four), respectively.

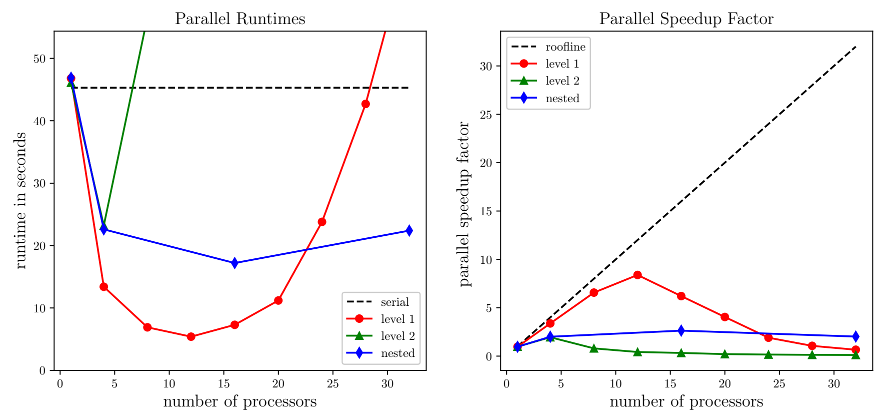

Fig. 1. Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 1000$ and $m = 1024$.

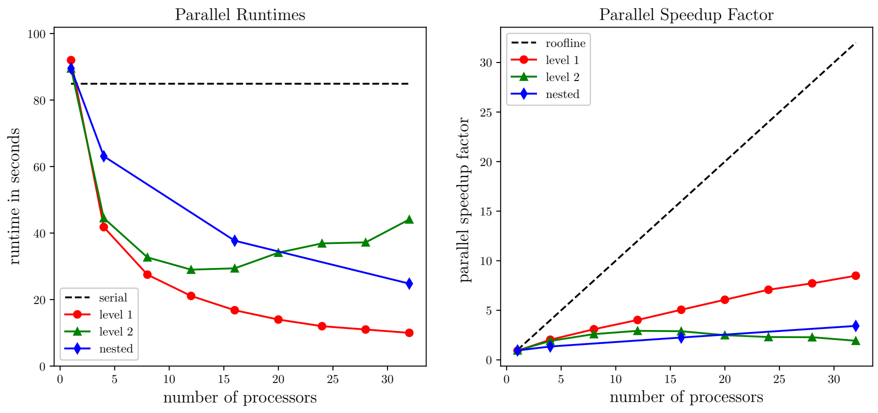

Fig. 2. Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 20,000$ and $m = 64$.

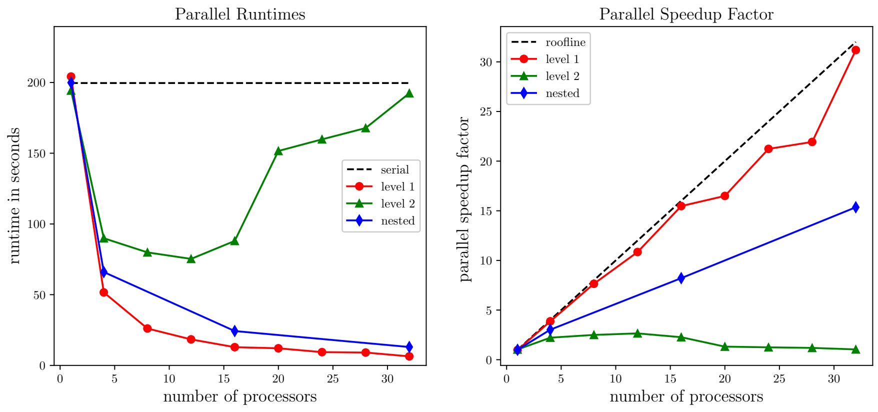

Fig. 3. Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 50-dimensional problem, with $n = 500$ and $m = 64$.

In all three figures, level one parallelism achieves the best parallel speedup factors, as expected. In particular, for the most expensive problem (Figure 3), level one parallelism hugs tightly against the strong scaling roofline. It is somewhat surprising to observe that level one parallelism did not scale to more than 12 processors for the problem size used in Figure 1 ($d = 10$, $n = 1000$, $m = 1024$), given that the opportunities for level one parallelism are maximized for large values of m. In fact, when the cost of performing each individual flip is relatively small (when d and n are small), the number of interpolation points (m) is relatively large, and the thread count per contention group is relatively high, level one threads can be blocked to the point of serialization by `CRITICAL` locks during the loop to "check ahead," as described in Section 5.1. This presents a trade-off since checking ahead does not result in significant conflict for relatively large values of n and d and could offer significant performance benefits when Q is clustered (as shown in Chang et al. [2018b]). In the cases where the cost of each flip is small, it is recommended that users partition the interpolation points into β batches, $Q = Q_1 \cup \ldots \cup Q_\beta$. Then each Q_i can be handled by a separate call to `DELAUNAYSPARSEP` with level one parallelism and an appropriate thread count. For example, in the case of Figure 1, for optimal performance over 32 active cores, `DELAUNAYSPARSEP` could be called four times ($\beta = 4$), with 8 threads and 256 interpolation points per call. In general, the optimal choices for β, the thread count per batch, and the physical partition are highly problem dependent and beyond the scope of this work.

Level two parallelism provides some performance improvements for large values of n (although less than level one), but does not appear to scale well to large numbers of processors. This is due to the redundant computations that are introduced with each additional level two thread. As a usage example, a user might prefer level two parallelism in a hybrid distributed/shared memory setting. First, the interpolation points (Q) could be broken up into extremely small batches and distributed across a large number of nodes, as suggested in Section 5. Then level two parallelism could be applied within each individual node, which may offer a limited number of shared memory processors.

In most cases, nested parallelism seems to offer a parallel speedup factor between that of level one and level two parallelism. Nested parallelism has a relatively limited usage case, providing a middle ground when there are multiple interpolation points (so that level one parallelism is available), but there are not enough interpolation points to fully saturate the system, making pure level one parallelism inefficient.

BIBLIOGRAPHY

Amos, B. D., Easterling, D. R., Watson, L. T., Thacker, W. I., Castle, B. S., Trosset, M. W. 2014. Algorithm XXX: QNSTOP: Quasi-Newton algorithm for stochastic optimization. *ACM Trans. Math Softw.*, to appear.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide (Third Edition).* SIAM, Philidelphia, PA.

Aurenhammer, F., Klein, R., and Lee, D. T. 2013. *Voronoi diagrams and Delaunay triangulations.* World Scientific Publishing Co., Hackensack, NJ.

Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. 1996. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. 22(4)*, 469–483.

Belkin, M., Hsu, D., and Mitra, P. P. 2018. Overfitting or perfect fitting? Risk bounds for classification and regression rules that interpolate. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, Curran Associates Inc., Montréal, Canada, 2306–2317.

Boissonnat, J.-D., Devillers, O., and Hornus, S. 2009. Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension. In *Proc. Twenty-fifth Annual Symp. on Computational Geometry*, ACM, Aarhus, Denmark, 208–216.

Bowyer, A. 1981. Computing Dirichlet tessellations. *The Computer Journal 24(2)*, 162–166.

Cameron, K. W., Anwar, A., Cheng, Y., Xu, L., Li, B., Ananth, U., Bernard, J., Jearls, C., Lux, T., Hong, Y., Watson, L. T., and Butt, A. R. 2019. MOANA: Modeling and analyzing I/O variability in parallel system experimental design. *IEEE Trans. Parallel and Distributed Systems 30(8)*, 1843–1856.

Chang, T. H., Watson, L. T., Lux, T. C. H., Bernard, J., Li, B., Xu, L., Back, G., Butt, A. R., Cameron, K. W., and Hong, Y. 2018a. Predicting system performance by interpolation using a high-dimensional Delaunay triangulation. In *Proc. 2018 Spring Simulation Conference, the 26th High Performance Computing Symp.*, Soc. for Modeling and Simulation Internat., Baltimore, MD, Article No. 2.

Chang, T. H., Watson, L. T., Lux, T. C. H., Li, B., Xu, L., Butt, A. R., Cameron, K. W., and Hong, Y. 2018b. A polynomial time algorithm for multivariate interpolation in arbitrary dimension via the Delaunay triangulation. In *Proc. ACM 2018 Southeast Conference (ACMSE '18)*, ACM, Richmond, KY, Article No. 12.

Cheney, E. W. and Light, W. A. 2009. *A Course in Approximation Theory.* American Mathematical Soc., Providence, RI.

Cheng, S. W., Dey, T. K., and Shewchuk, J. 2012. *Delaunay Mesh Generation.* Computer and Information Science Series, CRC Press, Boca Raton, FL.

Cignoni, P., Montani, C., and Scopigno, R. 1998. DeWall: A fast divide & conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design 30(5)*, 333–341.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. 2008. *Computational Geometry: Algorithms and Applications (third edition).* Springer-Verlag TELOS, Santa Clara, CA.

DEVILLERS, O., PION, S., AND TEILLAUD, M. 2001. Walking in a triangulation. In *Proc. Seventeenth Annual Symp. on Computational Geometry*, ACM, Medford, MA, 106–114.

EDELSBRUNNER, H. 1989. An acyclicity theorem for cell complexes in d dimensions. In *Proc. Fifth Annual Symp. on Computational Geometry*, ACM, Saarbruchen, West Germany, 145–151.

HANSON, R. J. AND HASKELL, K. H. 1982. Algorithm 587: Two algorithms for the linearly constrained least squares problem. *ACM Trans. Math. Softw. 8(3)*, 323–333.

KLEE, V. 1980. On the complexity of d-dimensional Voronoi diagrams. *Archiv der Mathematik 34(1)*, 75–80.

LIU, Y. AND YIN, G. 2019. Nonparametric functional approximation with Delaunay triangulation learner. In *Proc. 2019 IEEE International Conference on Big Knowledge*, IEEE, Beijing, China, pp. 167–174.

LUX, T. C. H., WATSON, L. T., CHANG, T. H., BERNARD, J., LI, B., XU, L., BACK, G., BUTT, A. R., CAMERON, K. W., AND HONG, Y. 2018. Predictive modeling of I/O characteristics in high performance computing systems. In *Proc. 2018 Spring Simulation Conference, the 26th High Performance Computing Symp.*, Soc. for Modeling and Simulation Internat., Baltimore, MD, Article No. 8.

LUX, T. C. H., WATSON, L. T., CHANG, T. H., HONG, Y., CAMERON, K. W. 2019. Interpolation of sparse high-dimensional data. *Numerical Algorithms*, submitted.

MÜCKE, E. P., SAIAS, I., AND ZHU, B. 1999. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry 12(1)*, 63–83.

MEGIDDO, N. 1991. On finding primal- and dual-optimal bases. *ORSA Journal on Computing 3(1)*, 63–65.

OMOHUNDRO, S. M. 1990. Geometric learning algorithms. *Physica D: Nonlinear Phenomena 42(1)*, 307–321.

OPENMP ARCHITECTURE REVIEW BOARD (ARB) 2015. OpenMP Application Programming Interface. version 4.5, November, 2015.

RAJAN, V. T. 1994. Optimality of the Delaunay triangulation in R^d. *Discrete & Computational Geometry 12(2)*, 189–202.

REGIS, R. G. 2015. The calculus of simplex gradients. *Optimization Letters 9(5)*, 845–865.

MILLER, G., PHILLIPS, T., AND SHEEHY, D. 2008. Linear-size meshes. In *Proc. of the 20th Canadian Conference on Computational Geometry*, Montréal, Québec, pp. 175–178.

SHROFF, G. M. AND BISCHOF, C. H. 1992. Adaptive condition estimation for rank-one updates of QR factorizations. *SIAM Journal on Matrix Analysis and Applications 13(4)*, 1264–1278.

SU, P. AND DRYSDALE, R. L. S. 1997. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry 7(5)*, 361–385.

WATSON, D. F. 1981. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal 24(2)*, 167–172.