Fast and Robust Approximation of Smallest Enclosing Balls in Arbitrary Dimensions

Thomas Larsson & Linus Källberg
Mälardalen University, Sweden

Abstract
In this paper, an algorithm is introduced that computes an arbitrarily fine approximation of the smallest enclosing ball of a point set in any dimension. This operation is important in, for example, classification, clustering, and data mining. The algorithm is very simple to implement, gives reliable results, and gracefully handles large problem instances in low and high dimensions, as confirmed by both theoretical arguments and empirical evaluation. For example, using a CPU with eight cores, it takes less than two seconds to compute a 1.001-approximation of the smallest enclosing ball of one million points uniformly distributed in a hypercube in dimension 200. Furthermore, the presented approach extends to a more general class of input objects, such as ball sets.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems, Analysis of algorithms and problem complexity [F.2.2]: Nonnumerical Algorithms and Problems—Geometrical problems and computations

1. Introduction
This paper is about fast construction of approximate smallest enclosing balls (SEBs), which are used widely to accelerate geometric operations and computational tasks in fields such as robotics, computer graphics, data mining, and machine learning. Consequently, besides being a challenging and rewarding problem to study in itself, the efficient construction of SEBs gives benefits in many different applications.

The smallest enclosing ball problem was first formulated for points in the plane by Sylvester in the year 1857. An interesting historical account of early work that followed is given by Blumenthal and Wahlin [BW41]. For a long time, the developed algorithms were superlinear in the number of input points, \( n \), and focused mainly on the planar case (some examples are [SH75, EH72, Sky91]). A breakthrough occurred when Megiddo presented a prune-and-search method that is able to find the smallest enclosing ball in worst-case linear time in any fixed dimension \( d \) [Meg83]. An actual implementation of this method, however, would probably be very slow due to a large hidden constant in the time complexity.

In contrast, practical algorithms that run in expected linear time have been proposed [Wel91, Gär99]. These methods repeatedly refine a candidate support set with up to \( d + 1 \) points until the exact solution is obtained. Such approaches can be very fast in low dimensions, say when \( d \leq 30 \), but in higher dimensions the performance deteriorates drastically due to their exponential dependency on \( d \). Another elegant combinatorial approach based on the idea of deflating a too large enclosing ball until the optimal solution is obtained can handle much larger dimensions [FGK03]. This solution appears to be usable in practice for quite large problem instances, say \( n, d \leq 10^4 \). A disadvantage, however, is that polynomial run-times cannot be guaranteed.

Also, in the more general case of computing the SEB of a set of balls, some interesting strategies are discussed in the literature. For instance, there are exact solutions based on linear programming [FG03] and approximation methods utilizing second-order cone programming (SOCP) formulations [KMY03, ZTS05].

The quite recent concept of coresets enables the design of fast \((1 + \epsilon)\)-approximation algorithms in arbitrary dimensions [BC03]. Initially, the user is allowed to request an approximation factor \( 1 + \epsilon \), \( \epsilon > 0 \). Then the algorithm locates an \( \epsilon \)-coreset \( C \), which is a subset of the input points \( P \) with a very useful property: the SEB of \( C \) scaled by a factor of \( 1 + \epsilon \) is guaranteed to also enclose the entire point set \( P \). The simplest algorithm in this category has running time \( O(nd/\epsilon^2) \) [BC03]. By repeatedly computing the small-
est enclosing ball of the points inserted in the coreset so far, algorithms with complexity $O(\frac{md}{\varepsilon^2} + \frac{1}{\varepsilon} f(d, \lceil \frac{1}{\varepsilon} \rceil))$ can be designed, where $f(d, m)$ denotes the time for solving the SEB sub-problem for a coreset of size $m$ in dimension $d$. By employing an SOCP-solver, Kumar et al. achieve time complexity $O(\frac{md}{\varepsilon^2} + \frac{1}{\varepsilon} \log \frac{1}{\varepsilon})$ [KMY03]. Several other algorithms relying on coresets with slightly worse time bounds have also been proposed [NN04, BC08].

The algorithms with the currently best theoretical time bound finish in $O(nd/\varepsilon)$ operations [Pan06, Yil08]. The cheap iterative update scheme of the current enclosing ball solution employed by these methods, however, leads to more passes through the input points and slower execution times in practice, except in high dimensions, where solving the SEB sub-problems is likely to dominate the runtime in algorithms utilizing more advanced sub-problem solvers. For planar point sets, an efficient solution that runs in $O(n\log(1/\varepsilon))$ is known [NN05].

Here, a $(1 + \varepsilon)$-approximation algorithm based on coresets is proposed. It possesses the following attractive advantages: (1) The algorithm (and its source code) is short and simple to understand. (2) It gives very reliable results. There are no degenerate cases that require special-case solutions. (3) It is worst-case efficient. The asymptotic time complexity is $O(\frac{md}{\varepsilon^2} + \frac{1}{\varepsilon})$. In fixed dimension with $\varepsilon$ constant, the algorithm runs in worst-case $O(n)$ time, which immediately suggests that the algorithm is very fast in low dimensions for, say, any $\varepsilon \geq 10^{-3}$. (4) It allows large problem instances to be solved efficiently in very high dimensions. (5) The algorithm is amenable to parallel processing on modern CPUs using SIMD computation and multi-core execution, as shown in Section 3. (6) The algorithm generalizes straightforwardly to handle, e.g., ball sets. For clarity of presentation, however, only point sets are considered in Section 2 and 3. Extensions to other inputs sets are discussed briefly in Section 4.

2. Algorithm description

As mentioned in Section 1, coresets provide a very elegant and useful tool when designing enclosing ball algorithms with provable approximation quality. The algorithm proposed here is based on the following design goals: (1) The solution converges in at most $O(1/\varepsilon)$ passes over the input points $P = \{p_0, p_1, \ldots, p_{n-1}\} \subset \mathbb{R}^d$ by finding a coreset $C$ of size $O(1/\varepsilon)$. (2) Rather than using an exact solver for small subsets of the input by solving systems of linear equations, a numerical search algorithm is utilized, which approximates the exact SEB $B^* = \langle c^*, r^* \rangle$ from below in a refined manner, i.e., $r_i < r_{i+1}$ and $r_i \leq r^*$. The maintained approximate solution is refined according to an effective ball expand operation until the requested approximation quality is reached. Given a current ball $B_i = \langle c_i, r_i \rangle$ with $r_i < r^*$, and the farthest outlier $q_i \in P$, a new increased radius is found from the smallest possible enclosing

![Figure 1: The expand ball operation. Given the current ball $B_i = \langle c_i, r_i \rangle$ and the farthest outlier $q_i$ from $c_i$, the next refined ball $B_{i+1} = \langle c_{i+1}, r_{i+1} \rangle$ is derived by computing $r_{i+1}$ as the circumradius of an isosceles triangle with height $h_i$ and base $2r_i$.](image1)

Figure 1: The expand ball operation. Given the current ball $B_i = \langle c_i, r_i \rangle$ and the farthest outlier $q_i$ from $c_i$, the next refined ball $B_{i+1} = \langle c_{i+1}, r_{i+1} \rangle$ is derived by computing $r_{i+1}$ as the circumradius of an isosceles triangle with height $h_i$ and base $2r_i$.}

**SOLVEAPXBALL** $(B, S, \delta)$

**input:** $B = \langle c, r \rangle$, $S = \{s_0, s_1, \ldots, s_{n-1}\}$, $\delta > 0$

**output:** A new ball $B = \langle c, r \rangle$ where $r \leq r^* \leq (1 + \delta) r$

1. loop $2/\delta$ times (at most)
2. $q, h \leftarrow$ FINDFARTHESTPOINT $(c, S)$
3. if $h \leq r(1 + \delta)$ then exit loop
4. $r \leftarrow (\frac{c}{\pi} + h)/2$
5. $c \leftarrow q + \frac{h}{2}(c - q)$
6. return $(c, r)$

**FASTAPXBALL** $(P, \varepsilon)$

**input:** $P = \{p_0, p_1, \ldots, p_{n-1}\}$, $\varepsilon > 0$

**output:** A $(1 + \varepsilon)$-approximation $B$ of $B^*$

1. $q', h' \leftarrow$ FINDFARTHESTPOINT $(p_0, P)$
2. $q, h \leftarrow$ FINDFARTHESTPOINT $(q', P)$
3. $(c, r) \leftarrow ((q' + q)/2, h/2)$
4. $C \leftarrow \{q', q\}$
5. loop $2/\varepsilon$ times (at most)
6. $q, h \leftarrow$ FINDFARTHESTPOINT $(c, P)$
7. if $h \leq r(1 + \varepsilon)$ then exit loop
8. $r \leftarrow (\frac{c}{\pi} + h)/2$
9. $c \leftarrow q + \frac{h}{2}(c - q)$
10. $C \leftarrow C \cup \{q\}$
11. $(c, r) \leftarrow$ SOLVEAPXBALL $(\langle c, r \rangle, C, \varepsilon/2)$
12. return $(c, h)$

**Figure 2: The FASTAPXBALL algorithm. An incrementally constructed coreset $C \subseteq P$ is maintained for which the inner solver SOLVEAPXBALL computes approximate SEBs.**
ball of a hemisphere $H_i$ of $B_i$ and $q_i$. This update operation can be derived from simple geometrical arguments, which are illustrated in Figure 1. Pick any diameter $ab$ in the base of the hemisphere $H_i = (c_i, r_i, q_i - c_i)$, and form the triangle $abq_i$. Since this is, by construction, an isosceles triangle with base $2r_i$ and height $h_i = \|c_i - q_i\|$, the radius of $B_{i+1}$ is directly given by

$$r_{i+1} = \frac{r_i^2/h_i + h_i}{2}. \quad (1)$$

The updated center point is then simply computed as

$$c_{i+1} = q_i + \frac{r_{i+1}(c_i - q_i)}{h_i}.$$ \quad (2)

This means that $c_{i+1}$ is positioned along the line segment $q_ic_i$ at distance $r_{i+1}$ from $q_i$. Hence, the movement of the center point is the smallest possible to get the outlier $q_i$ in the surface of the updated ball $B_{i+1}$.

Since $h_i > r_i$ and the diameter $ab$ in $B_i$ is in fact a chord in the updated ball $B_{i+1}$, it is obvious that the ball grows in each update operation. This can be shown also algebraically by rewriting Equation 1 as

$$r_{i+1} = r_i + \frac{(h_i - r_i)^2}{2h_i}. \quad (3)$$

Clearly, since $r_i \geq 0$ and $h_i > r_i$, the produced sequence of radii is monotonically increasing.

Of course, the initial ball $B_0$ must be defined before the first expand operation is carried out. A sufficient initialization would be to let $r_0 = 0$ and $c_0$ be any point in the input. However, an initial greedy search for a long line segment $q'q$ is preferred here. First, find the point $q'$ at farthest distance from input point $r_0$, then find the point $q$ farthest from $q'$. The first ball $B_0$ is then simply given by

$$r_0 = \frac{\|q' - q\|}{2}, \quad (4)$$

$$c_0 = \frac{(q' + q)}{2}. \quad (5)$$

Note, this is the same initialization as used by Kumar et al. [KMY03]. Clearly, the initial radius $r_0 \geq r^*/2$ in any dimension, since as shown by [GIV01], every hemisphere of $B^*$ has at least on point on its surface. Hence, the minimum distance from an arbitrary input point to its farthest neighbour is at least $r^*$.

The complete algorithm, called FASTAPXBALL, is presented using pseudocode in Figure 2. On Lines 1–3, the initialization just described is performed, and then the two points spanning the initial ball are inserted in the coreset $C$ (Line 4). This initial ball is then iteratively refined by the main loop starting on Line 5, which is guaranteed to end with the requested approximation after at most $2/\varepsilon$ passes. In each pass, the point $q$ farthest away from the current center $c$ is found by a linear scan of the input points, together with its distance $h$ from $c$ (Line 6). Clearly, this immediately gives an enclosing ball $(c, h)$ of the input $P$, with $h \geq r^*$. Furthermore, since $r \leq r^*$, this enclosing ball is at least an $h/r$-approximation. When the condition on Line 7 is true, the loop can therefore be terminated and the solution returned (Line 12). Otherwise, the current ball is expanded according to the Equations 1 and 2 (Lines 8–9), and the farthest outlier is added to the coreset $C$ (Line 10).

After this, the procedure SOLVEAPXBALL is called that computes an approximate SEB of the current coreset $C$ (Line 11). This is done by refining the current ball solution, which is given as input. Also, note that a stricter approximation factor, $\varepsilon/2$, is used. When SOLVEAPXBALL returns, the main loop continues with the next pass. The purpose of the procedure SOLVEAPXBALL, which will be referred to as the inner solver hereafter, is to reduce the number of passes over the entire input $P$ by iterating over a subset of points, the coreset $C$. Note that inside the inner solver, the current ball solution is updated exactly in the same way as is done in the main loop, i.e., by finding the farthest point from the center $c$, checking if the required approximation is reached, and if not refine the current ball by applying Equations 1 and 2.

2.1. Analysis

The presented algorithm FASTAPXBALL returns a $(1 + \varepsilon)$-approximation of $B^*$ in a finite number of steps. The algorithms are as follows. First, it is shown that $r_i \leq r^*$ holds for all intermediate balls $B_i$ computed during the run of the algorithm. To this end, the following property of an intermediate ball is defined.

Definition 1 An intermediate ball $B_i$ is viable if it has at least one diameter inside of $B^*$, i.e.,

$$\|c_i - c^*\|^2 \leq r^{*2} - r_i^2. \quad (6)$$

Clearly, $r_i \leq r^*$ holds for a viable ball $B_i$. 

![Figure 3: Illustration of an imaginary situation where $B_{i+1}$ is not viable, which necessarily implies an obtuse angle at $c_{i+1}$ in the triangle with corners $c_i$, $c_{i+1}$, and $c^*$.](image-url)
Lemma 1 If \( B_i \) is a viable ball, \( q_i \in B^* \) is a point outside of \( B_i \), and \( B_{i+1} \) is a ball computed according to Equations 1 and 2, then \( B_{i+1} \) is viable.

Proof Assume that \( B_{i+1} \) is not viable, i.e., \( \|c_{i+1} - c^*\|^2 > r_{i+1}^2 \). Recall that \( B_{i+1} \) tightly encloses a diameter of \( B_i \):

\[
r_i^2 = r_{i+1}^2 - \|c_{i+1} - c_i\|^2.
\]

Also recall that \( c_i, c_{i+1}, \) and \( q_i \) are collinear with \( c_{i+1} \) lying in between the other two points. Since \( q_i \) lies on the surface of \( B_{i+1} \) and inside of \( B^* \), \( c_i \) must then lie in the hemisphere of \( B_{i+1} \) facing away from \( c^* \), see illustration in Figure 3. Clearly, the interior angle at \( c_{i+1} \) in the triangle defined by the center points \( c_i, c_{i+1}, \) and \( c^* \) must be obtuse. Thus, by the law of cosines,

\[
\|c_i - c^*\|^2 = \|c_{i+1} - c^*\|^2 + \|c_{i+1} - c_i\|^2 + \gamma,
\]

where \( \gamma \geq 0 \). Since \( B_i \) is viable, the condition in Equation 6 must be met with the value of \( r_i^2 \) given in Equation 7 and the value of \( \|c_i - c^*\|^2 \) according to Equation 8. Thus,

\[
\|c_{i+1} - c^*\|^2 + \|c_{i+1} - c_i\|^2 + \gamma \leq \|c_i - c^*\|^2 - r_i^2 + \|c_{i+1} - c_i\|^2.
\]

Cancelling out the common term \( \|c_{i+1} - c_i\|^2 \) leaves

\[
\|c_{i+1} - c^*\|^2 + \gamma \leq \|c_i - c^*\|^2 - r_i^2,
\]

which contradicts the initial assumption that \( B_{i+1} \) is not viable. \( \square \)

Lemma 2 No intermediate ball produced during the run of algorithm FASTAPXBALL has a radius larger than \( r^* \).

Proof Since \( B_0 \) is spanned by two antipodal points, both enclosed by \( B^* \), it is clearly viable. This, together with Lemma 1, ensures that all subsequent balls are also viable. Since viability of a ball \( B_i \) ensures \( r_i \leq r^* \), this proves the lemma. \( \square \)

Lemma 3 Algorithm FASTAPXBALL generates a \((1 + \varepsilon)\)-approximation of \( B^* \) in at most \( 2/\varepsilon \) iterations.

Proof During iteration \( i \), let \( h_i \) be the distance to the point farthest from \( c_i \). If \( h_i \leq R^* := (1 + \varepsilon) r^* \), then we are done. Otherwise, the next radius \( r_{i+1} \) satisfies

\[
r_{i+1} \geq \frac{r_i^2 h_i + h_i^2}{2} > \frac{r_i^2 R^* + R^*}{2}.
\]

Note, without the inner solver, the first inequality above becomes equality. Normalizing \( r_{i+1} \) with respect to \( R^* \) gives

\[
r_{i+1}/R^* > \frac{(r_i/R^*)^2 + 1}{2}.
\]

This lower bound on \( r_{i+1}/R^* \) is exactly the same bound as derived by Bădoiu and Clarkson [BC03] for the minimum growth of the radius in the case where an exact inner solver is used. We may therefore proceed in a similar manner as in their proof. Since \( r_0 \geq 0 \), the recurrence relation in Equation 12 is bounded below by

\[
r_i/R^* \geq 1 - \frac{1}{1 + i/2}.
\]

Due to Lemma 2, we must be done when reaching the state \( r_i/R^* \geq 1/(1 + \varepsilon) \), since this implies \( r_i \geq r^* \). By equating the right-hand-side of Equation 13 with this value of \( r_i/R^* \) and solving for \( i \), we get an upper bound on the number of iterations:

\[
1 - \frac{1}{1 + i/2} = \frac{1}{1 + \varepsilon} \Rightarrow i = 2/\varepsilon
\]

\( \square \)

By using Lemma 3, the time complexity of the algorithm can be derived. The number of iterations in the outer loop is bounded by \( 2/\varepsilon \), since one update of a viable ball is made in each pass (Lines 8–9). Each iteration makes a single linear scan of the input points at cost \( O(nd) \) (Line 6). The inner solver is called once for each main iteration (Line 11), and it runs itself at most \( 2/\delta = 4/\varepsilon \) passes over the current coreset whose size is at most \( 2/\varepsilon \). Therefore, the total work \( T \) of the algorithm amounts to

\[
T(n,d,\varepsilon) = O\left(\frac{2}{\varepsilon} (nd + 4 \frac{1}{\varepsilon} d)\right) = O\left(\frac{nd}{\varepsilon} + \frac{d}{\varepsilon^2}\right).
\]

Note that it is impossible for the inner solver to update the current ball in such a way that it leads to more than \( 2/\varepsilon \) passes in the outer main loop as a side effect. The reason is that the inner solver not only increases the radius by requiring a finer approximation, i.e., \( \delta < 1/2 \), but at the same time, it also maintains a viable ball with at least a diameter within \( B^* \), according to Lemmas 1 and 2.

3. Benchmarks

The proposed algorithm has been rigorously tested in practice under varying circumstances. Here, three experiments are described in detail, which illustrate the performance behaviour over a broad range of inputs. No robustness issues have been observed. Note, however, that it is required that the ball update operations, described in Equations 1 and 2, are evaluated using at least twice as many digits of precision as used for the approximation factor \((1 + \varepsilon)\). This means that for \( \varepsilon \geq 10^{-6} \), double precision (eight bytes wide) floating point operations are sufficient, and for \( \varepsilon \geq 10^{-4} \), single precision (four bytes wide) operations are enough. Also, note that it is enough to store the coreset \( C \) in a simple array. When inserting a point, there is no need to look for duplicates in the coreset, since if a point is selected as the farthest point a second time, then the required approximation factor must have been reached, and the algorithm terminates. Consequently, the number of passes \( I \) through the input points \( P \) is always \( I = |C| + 1 \).
In this first experiment, the algorithm is applied on large problem instances \( \varepsilon = 10^{-3} \), \( n \cdot d = 10^9 \), and two different input distributions. The sequential performance, \( t_S \) is compared to parallel versions using vectorized instructions \( t_V \), multiple cores without vectorized instructions \( t_M \), and both vectorization and multiple cores \( t_{V,M} \). All results are averages from runs over 10 different random point sets.

### Uniform distribution in the unit hypercube

| \( n \) | \( d \) | \( |C| \) | \( t_S \) | \( t_V \) | \( t_M \) | \( t_{V,M} \) |
|---|---|---|---|---|---|---|
| \( 10^7 \) | \( 10^2 \) | 61.2 | 59 | 24 | 11 | 7 |
| \( 10^6 \) | \( 10^3 \) | 185.4 | 194 | 60 | 32 | 19 |
| \( 10^5 \) | \( 10^4 \) | 401.2 | 419 | 136 | 69 | 40 |
| \( 10^4 \) | \( 10^5 \) | 611.5 | 659 | 237 | 108 | 65 |
| \( 10^3 \) | \( 10^6 \) | 686.2 | 969 | 354 | 158 | 104 |

### Uniform distribution in the unit ball

| \( n \) | \( d \) | \( |C| \) | \( t_S \) | \( t_V \) | \( t_M \) | \( t_{V,M} \) |
|---|---|---|---|---|---|---|
| \( 10^7 \) | \( 10^2 \) | 80.2 | 74 | 32 | 14 | 9 |
| \( 10^6 \) | \( 10^3 \) | 248.2 | 256 | 81 | 43 | 26 |
| \( 10^5 \) | \( 10^4 \) | 655.3 | 684 | 222 | 112 | 65 |
| \( 10^4 \) | \( 10^5 \) | 898.7 | 976 | 352 | 162 | 95 |
| \( 10^3 \) | \( 10^6 \) | 945.6 | 1460 | 531 | 241 | 156 |

Table 1: Coreset sizes \( |C| \) and execution times in seconds for large problem instances \( nd = 10^9 \), \( \varepsilon = 10^{-3} \) and two different input distributions. The sequential performance, \( t_S \) is compared to parallel versions using vectorized instructions \( t_V \), multiple cores without vectorized instructions \( t_M \), and both vectorization and multiple cores \( t_{V,M} \). All results are averages from runs over 10 different random point sets.

The algorithm and a suitable test framework were implemented in C++ and compiled as a 64-bit program using Visual Studio 2012 with the release mode setting under Windows 7. Then all practical tests were run using a standard PC with an Intel Xeon CPU E5-2665 running at 2.4 GHz with 16 GB of main memory. This CPU has 8 physical cores and each core supports the first generation of Intel’s vectorized instruction set Advanced Vector Extensions (AVX). Since standard PC computers are developed towards becoming highly parallel machines, several parallel versions of the algorithm were compared to the sequential implementation. The used parallelization strategies are described further together with the experiments in the following sections.

### 3.1. First experiment: large problems

In this first experiment, the algorithm is applied on large problems instances with \( \varepsilon = 10^{-3} \). Both \( n \) and \( d \) are varied, but the total size of the input data is held constant with \( nd = 10^9 \). The data is stored in a dynamically allocated 2D array. Each data value requires 4 bytes, which gives a storage cost of approximately 3.7 GB. For each problem instance, the algorithm is run on ten different random point sets and the results are averaged. Also, the entire testing procedure is repeated for two different data distributions.

The coreset sizes and the sequential run-times are listed in columns \( |C| \) and \( t_S \) in Table 1. Evidently, inputs uniformly distributed inside a unit ball are more challenging for the algorithm than points in a cube. When \( n \) is decreased and \( d \) increased, the problem becomes more computationally intensive with longer run-times and larger coresets.

Parallel versions of the algorithm are also tested. Clearly, the bottleneck of the algorithm is the repeated computation of the farthest point in \( P \) from another point \( c \). This search involves \( n \) computations of the squared distance between two points in each pass. Hence, the main focus is on parallelizing this procedure. It is straightforward to utilize data-parallel instructions for these distance computations. Groups of eight coordinate values, each one 4 bytes wide, are repeatedly packed into AVX registers and then AVX instructions are applied. For each squared distance computation, eight partial sums are computed simultaneously, which are then added together.

To exploit another layer of parallelism, multi-threaded execution on multiple CPU cores is used. Again the focus is on the procedure searching for the farthest point. The idea is to divide the job over a group of \( w \) worker threads, each one processing 1/\( w \) of the search loop. Implementing this is straightforward, since the order in which the points are considered is not important. Of course, when the threads have finished, their computed maximum distances must be merged. To avoid unnecessary overhead from thread creation and initialization, a pool of threads is created and put to sleep, when the algorithm starts. Then at each call to the farthest point procedure, the threads are simply notified.

Note that AVX and multi-core parallelization can be used independently from each other, or be combined (since each core support AVX). The run-times obtained from using these parallelization techniques are given in columns \( t_V \), \( t_M \), and \( t_{V,M} \) in Table 1. As can be seen, the AVX-only version gives speedups in the range 2.3–3.2. On the other hand, when using multi-threaded execution, without AVX instructions, the speedups are between 5.3 and 6.1. And finally, combining the multi-threaded solution with data-parallel instructions gives 8.2–10.5 times faster execution times. In all multi-threaded runs, \( w = 8 \) is used. The probable reason why the AVX version does not give a larger increase in performance, despite the fact that the SIMD registers hold eight scalar values, is that the computation of squared distances between points involves very few arithmetic operations leading to a mainly memory-bounded execution.

In this experiment, no direct comparisons have been made to other methods. Running an exact algorithm on the problem sizes in Table 1 seems not feasible, and Kumar et al. only report approximation results up to dimension 1500 [KMY03]. Zhou et al. report running times over an hour for problems sizes of the order \( nd = 10^7 \) [ZTS05].

### 3.2. Second experiment: varying \( d \) and \( \varepsilon \)

In this experiment the dimension is increased from 500 to 5000 while the number of points is always \( n = 5000 \). This
means that more moderate problem sizes are studied where $n \geq d$. Also, three different values of $\epsilon$ are included to show the effect on the coreset sizes and how the algorithm scales under increased approximation quality. The input points are stored in a dynamically allocated 2D array using 8 bytes wide floating point numbers. Again, the average results are obtained from 10 different random inputs for each problem size and two different input distributions.

The resulting run-times are plotted in Figure 4. The sequential single-threaded run-times (in the top row) can be compared to the parallel run-times, where the utilization of both AVX instructions and multi-threaded execution results in a $5.8 \times$ speedup on average (middle row). The parallelization techniques used are exactly the same as in the first experiment, except that the AVX register can only hold four packed data values, since the numbers are 8 bytes wide. This means the potential speedup for an AVX version is halved. Although not included in the plots shown here, it is worth mentioning that using only AVX parallelization gives roughly 1.6–2.4 as fast execution times compared to the sequential solution. Using only multi-threading gives better results, with a speedup of 4.7 on average. This means that combining both techniques gives a somewhat disappointing
The coreset sizes are plotted in the bottom row. Clearly, they are much smaller for inputs uniformly distributed in a cube compared to in a ball, which immediately shows up in significantly faster execution times for the cube data. Also, note that the coreset sizes appear to grow sublinearly with the dimension in these plots. In all cases, the coreset sizes are significantly smaller than the upper bound of $2^d$.

**Comparison to other methods:** The exact solver presented by Fischer et al. [FGK03] is publicly available with source code on the web. First of all, this made it possible to verify the correctness of the enclosing balls computed by FastApproxBall. In all cases, it was confirmed that the algorithm computes $(1 + \varepsilon)$-approximations correctly. Furthermore, the exact algorithm was found to be substantially slower, with runtimes in the range 3–150 and 43–1050 seconds for the cube and ball data, respectively. Kumar et al. demonstrate good performance when $n \gg d$, but they only report results for $d \leq 1500$ [KMY03].

### 3.3. Third experiment: low dimensions

In three dimensions, it is preferable to implement the algorithm slightly differently. For example, it is advantageous to store the coreset as a 1D array holding deep copies of the corresponding points in the input to improve data coherence in the inner solver. In high dimensions, however, this would incur a quite substantial additional storage cost as opposed to storing only references to points. When it comes to vectorizing the code, it is important to utilize the full width of the AVX registers in the computations. Therefore, when scanning the inputs for the farthest point, data swizzling is performed on-the-fly to load eight vertices at a time into three full AVX registers, one for each coordinate $x$, $y$, and $z$, respectively. Then the squared distance computations can be performed for 8 points in parallel. On the other hand, if the data is given in three separate 1D arrays initially, one for each type of coordinate, no data swizzling is needed. To examine both cases, two AVX versions are included in this experiment, together with the sequential version.

To evaluate the algorithm in 3D, random point sets uniformly distributed in a cube and a ball, as well as the vertices of some polygon meshes, are used as input. Rendered images of these meshes confined in computed bounding spheres are shown in Figure 5. Note that some of these objects have become standard models for testing various kinds of algorithms in computer graphics and related fields. The results are presented in Table 2 for $\varepsilon = 10^{-3}$ and $\varepsilon = 10^{-5}$. In all cases, the coreset size, $|C|$, is substantially smaller than the upper bound $2^d$. In fact, the worst case encountered here is a coreset of size 14. Thus, the number of main passes over the entire input is indeed few in low dimensions. The sequential run times, $t_S$, are compared to vectorized run times $t_{V1}$ (with data-swizzling) and $t_{V2}$ (no data-swizzling). The AVX versions give speedups of 1.1–3.6 and 1.1–5.6, respectively, compared to the sequential solution. Also, note that the overhead cost of on-the-fly swizzling is quite small.

The results for the Cube and Blade data sets show no dif-

### Table 2: Coreset sizes $|C|$ and average run-times in seconds from ten runs on each data set. The sequential times, $t_S$, are compared with the times from two AVX versions, $t_{V1}$ and $t_{V2}$.

| Data sets in 3D | $P$     | $n$   | $\varepsilon$ | $|C|$ | $t_S$ | $t_{V1}$ | $t_{V2}$ |
|----------------|---------|-------|---------------|------|-------|--------|--------|
| Cube           | 4M      | $10^{-3}$ | 5           | 0.058 | 0.022 | 0.019  |
| Cube           | 4M      | $10^{-5}$ | 5           | 0.069 | 0.032 | 0.029  |
| Cube           | 16M     | $10^{-3}$ | 3           | 0.157 | 0.057 | 0.049  |
| Cube           | 16M     | $10^{-5}$ | 3           | 0.156 | 0.057 | 0.049  |
| Ball           | 4M      | $10^{-3}$ | 3           | 0.038 | 0.014 | 0.013  |
| Ball           | 4M      | $10^{-5}$ | 9           | 0.129 | 0.070 | 0.065  |
| Ball           | 16M     | $10^{-3}$ | 3           | 0.155 | 0.058 | 0.050  |
| Ball           | 16M     | $10^{-5}$ | 7           | 0.331 | 0.135 | 0.119  |
| Hand           | 0.33M   | $10^{-3}$ | 4           | 0.204 | 0.0042 | 0.0013 | 0.0009 |
| Hand           | 0.33M   | $10^{-5}$ | 6           | 0.0485 | 0.0441 | 0.0430 |
| Blade          | 0.88M   | $10^{-3}$ | 2           | 0.0062 | 0.0017 | 0.0011 |
| Blade          | 0.88M   | $10^{-5}$ | 2           | 0.0062 | 0.0017 | 0.0012 |
| Goblet         | 1M      | $10^{-3}$ | 9           | 0.231 | 0.0066 | 0.0046 |
| Goblet         | 1M      | $10^{-5}$ | 14          | 0.1078 | 0.0824 | 0.0796 |
| Lucy           | 14M     | $10^{-3}$ | 4           | 0.1707 | 0.0635 | 0.0552 |
| Lucy           | 14M     | $10^{-5}$ | 6           | 0.2515 | 0.1021 | 0.0906 |
ferences in coreset sizes for different values of $\varepsilon$. In contrast, the Ball data and the Hand, Goblet and Lucy meshes are more challenging. In these cases, the size of the coreset increases with finer $\varepsilon$. In fact, for $\varepsilon = 10^{-3}$, the Hand and Goblet meshes give quite poor performance. A closer look at these cases reveals that the number of iterations in the inner solver is relatively high, which means the second term in the time complexity given in Equation 15 starts to dominate. Clearly, when $n$ is not significantly larger than $2/\varepsilon$, the risk for such performance degradation increases. Therefore, under such circumstances, it would be much more efficient to replace the inner solver with an exact SEB solver, which can be realized very efficiently in low dimensions.

4. Generalization to other types of input

The presented algorithm can be generalized to compute $(1 + \varepsilon)$-approximate SEBs of other geometrical objects, at least as long as these objects are themselves compact point sets and a practical operation is available to locate the farthest point from the center $c_i$ of the current ball. Then it suffices to replace the calls of the subroutine FINDFARTESTPOINT on Lines 1, 2, and 6 in FASTAPXBALL to a procedure with different input primitives, but the same type of output—the farthest outlier $q_i$ at distance $h_i$ from the current center $c_i$. Clearly, Lemmas 1–3 remain valid, which means the size of the coreset is still $O(1/\varepsilon)$.

To exemplify, consider a set of $n$ balls with center points $a_j$ and radii $b_j$ as input. Then the maximum distance from $c_i$ to a point in the ball set is given by

$$h_i = \max(||a_j - c_i|| + b_j).$$

(16)

Denote the ball that satisfy this Equation by $(a_q, b_q)$. The actual point farthest from $c_i$ is then given by

$$q_i = a_q + \frac{h_i}{b_i - b_q}(a_q - c_i).$$

(17)

Clearly, since the cost of processing each ball is only $O(d)$, the time complexity of the algorithm stated for point sets in Equation 15 is valid also for ball sets. Indeed, the algorithm is fast in practice also for balls. A few brief example results in 3D are given in Figure 6.

An example of a more advanced primitive is the ellipsoid. Since computing the farthest point on an ellipsoid from a given point requires some kind of iterative root finding technique, the overall time complexity of the algorithm increases for ellipsoids.

5. Conclusions

FASTAPXBALL is a very attractive $(1 + \varepsilon)$-approximation algorithm. Clearly, the algorithm is fast, simple to implement, and works in general dimension. In particular, the algorithm has excellent performance for massive point sets when $\varepsilon \geq 10^{-3}$. Also, the developed parallel versions of the algorithm demonstrate the necessity to embrace heterogeneous hardware features available in today’s computers when run-time efficiency matters. Of course, since finding the farthest point is a bottleneck also in many previously proposed ball construction methods, these methods can benefit from similar parallelization.

Since the algorithm is useful in both low and high dimensions, it may be considered in many different applications dealing with multi-dimensional spatial queries. Bounding volume hierarchies are often employed to accelerate neighbour queries, collision detection, and rendering, and the enclosing ball is one of the most used bounding shapes in such data structures (some interesting examples are [KS97, Hub96, KL10]). High-dimensional cases arise in SVM training and classification, where the utilization of fast approximate enclosing ball algorithms has enabled a breakthrough in performance and problem sizes that can be handled. Prominent examples of this are the core vector machines [TKC05, TKK07].

Finally, note that if Line 11 in the pseudocode in Figure 2 is removed, thereby eliminating the inner solver, a new algorithm emerges, call it SIMPLEAPXBALL, with an improved overall asymptotic time complexity of $O(\frac{d}{\varepsilon})$, which is on par with the best known algorithms [Pan06, Yil08]. But since the inner solver often reduces the number of passes over the input significantly, this variation tends to be considerably slower. In fact, the difference can be several orders of magnitude in low dimensions.

6. Further studies

There are several interesting avenues for further work. For instance, a careful study of the behaviour of FASTAPXBALL vs. SIMPLEAPXBALL in practice is needed. Clearly, when $n \gg d$, FASTAPXBALL is substantially faster. However, as $d$ approaches and grows beyond $n$, the need for the inner solver diminishes and SIMPLEAPXBALL may become the winner. A hybrid between the two algorithms seems inter-

Figure 6: Ball sets with bounding spheres ($\varepsilon = 10^{-3}$). The 1000 balls shown left results in $|C| = 8$, and the Bunny made of 69K balls gives $|C| = 5$ in sequential time 1.3 ms.
References


