Fast Smallest-Enclosing-Ball Computation in High Dimensions[⋆]

Kaspar Fischer[1] [⋆⋆], Bernd Gärtner[1], and Martin Kutz[2] [⋆⋆⋆]

[1] ETH Zürich, Switzerland
[2] FU Berlin, Germany

Abstract. We develop a simple combinatorial algorithm for computing the smallest enclosing ball of a set of points in high dimensional Euclidean space. The resulting code is in most cases faster (sometimes significantly) than recent dedicated methods that only deliver approximate results, and it beats off-the-shelf solutions, based e.g. on quadratic programming solvers.
The algorithm resembles the simplex algorithm for linear programming; it comes with a Bland-type rule to avoid cycling in presence of degeneracies and it typically requires very few iterations. We provide a fast and robust floating-point implementation whose efficiency is based on a new dynamic data structure for maintaining intermediate solutions.
The code can efficiently handle point sets in dimensions up to 2,000, and it solves instances of dimension 10,000 within hours. In low dimensions, the algorithm can keep up with the fastest computational geometry codes that are available.

1 Introduction

The problem of finding the smallest enclosing ball (SEB, a.k.a. minimum bounding sphere) of a set of points is a well-studied problem with a large number of applications; if the points live in low dimension d ($d \leq 30$, say), methods from computational geometry yield solutions that are quite satisfactory in theory and in practice [1–5]. The case $d = 3$ has important applications in graphics, most notably for visibility culling and bounding sphere hierarchies.

There are a number of very recent applications in connection with *support vector machines* that require the problem to be solved in higher dimensions; these include e.g. high-dimensional clustering [6, 7] and nearest neighbor search [8], see also the references in Kumar et al. [9].

[⋆] Partly supported by the IST Programme of the EU and the Swiss Federal Office for Education and Science as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG – Effective Computational Geometry for Curves and Surfaces).

[⋆⋆] Supported by the Berlin/Zürich joint graduate program "Combinatorics, Geometry, and Computation" (CGC).

[⋆⋆⋆] Member of the European graduate school "Combinatorics, Geometry, and Computation" supported by the Deutsche Forschungsgemeinschaft, grant GRK588/2.

The existing computational geometry approaches cannot (and were not designed to) deal with most of these applications because they become inefficient already for moderately high values of d. While codes based on Welzl's method [3] cannot reasonably handle point sets beyond dimension $d = 30$ [4], the quadratic programming (QP) approach of Gärtner and Schönherr [5] is in practice polynomial in d. However, it critically requires arbitrary-precision linear algebra to avoid robustness issues, which limits the tractable dimensions to $d \leq 300$ [5].

Higher dimensions can be dealt with using state-of-the-art floating point solvers for QP, like e.g. the one of CPLEX [10]. It has also been shown that the SEB problem is an instance of *second order cone programming* (SOCP), for which off-the-shelf solutions are available as well, see [11] and the references there.

It has already been observed by Zhou et al. that general-purpose solvers can be outperformed by taking the special structure of the SEB problem into account. Their result [11] is an interior point code which can even handle values up to $d = 10{,}000$. The code is designed for the case where the number of points is not larger than the dimension; test runs only cover this case and stop as soon as the ball has been determined up to a fixed accuracy. Zhou et al.'s method also works for computing the approximate smallest enclosing ball of a set of balls.

The recent polynomial-time $(1+\epsilon)$-approximation algorithm of Kumar et al. goes in a similar direction: it uses additional structure (in this case *core sets*) on top of the SOCP formulation in order to arrive at an efficient implementation for higher d (test results are only given up to $d = 1{,}400$) [9].

In this paper, we argue that in order to obtain a fast solution even for very high dimensions, it is not necessary to settle for suboptimal balls: we compute the *exact* smallest enclosing ball using a *combinatorial* algorithm. Unlike the approximate methods, our algorithm constitutes an exact method in the RAM model; and our floating-point implementation shows very stable behaviour. This implementation beats off-the-shelf interior-point-based methods as well as Kumar et. al.'s approximate method; only for $d \geq 1{,}000$, and if the number of points does not considerably exceed d, our code is outperformed by the method of Zhou et al. (which, however, only computes approximate solutions).

Our algorithm—which is a pivoting scheme resembling the simplex method for linear programming—actually computes the set of at most $d + 1$ *support points* whose circumsphere determines the smallest enclosing ball. The number of iterations is small in practice, but there is no polynomial bound on the worst-case performance.

The idea behind the method is simple and a variant has in fact already been proposed by Hopp et al. as a heuristic, along with an implementation for $d = 3$, but without any attempts to prove correctness and termination [12]: start with a balloon strictly containing all the points and then deflate it until it cannot shrink anymore without loosing a point. Our contribution is two-fold. On the theoretical side, we develop a pivot rule which guarantees termination of the method even under degeneracies. In contrast, a naive implementation might cycle (Hopp et al. ignore this issue). The rule is Bland's rule for the simplex method [13], adapted to our scenario, in which case the finiteness has an appealing geometric proof. On the practical side, we represent intermediate solutions (which are affinely independent point sets, along with their circumcenters) in a way that allows fast and robust updates under insertion or deletion of a single point. Our representation is an adaptation of the QR-factorization technique [14]. Already for $d = 3$, this makes our code much faster than that of Hopp et al. and we can efficiently handle point sets in dimensions up to $d = 2{,}000$. Within hours, we are even able to compute the SEB for point sets in dimensions up to 10,000, which is the highest dimension for which Zhou et al. give test results [11].

2 Sketch of the Algorithm

This section provides the basic notions and a central fact about the SEB problem. We briefly sketch the main idea of our algorithm, postponing the details to Section 3.

Basics. We denote by $B(c,r) = \{x \in \mathbb{R}^d \mid \|x-c\| \leq r\}$ the d-dimensional ball of center $c \in \mathbb{R}^d$ and radius $r \in \mathbb{R}_+$. For a point set T and a point c in $\mathbb{R}^d$, we write $B(c,T)$ for the ball $B(c, \max_{p\in T} \|p-c\|)$, i.e., the smallest ball with given center c that encloses the points T. The *smallest enclosing ball* $\textsc{seb}(S)$ of a finite point set $S \subset \mathbb{R}^d$ is defined as the ball of minimal radius which contains the points in S, i.e., the ball $B(c,S)$ of smallest radius over all $c \in \mathbb{R}^d$. The existence and uniqueness of $\textsc{seb}(S)$ are well-known [3], and so is the following fact which goes back to Seidel [5].

Lemma 1 (Seidel). *Let T be a set of points on the boundary of some ball B with center c. Then $B = \textsc{seb}(T)$ if and only if $c \in \operatorname{conv}(T)$.*

We provide a simple observation about convex combinations of points on a sphere, which will play a role in the termination proof of our algorithm.

Lemma 2. *Let T be a set of points on the boundary of some ball with positive radius and with center $c \in \operatorname{conv}(T)$. Fix any set of coefficients such that*

$$c = \sum_{p\in T} \lambda_p p, \quad \sum_{p\in T} \lambda_p = 1, \quad \forall p \in T\colon \lambda_p \geq 0.$$

Then $\lambda_p \leq 1/2$ for all $p \in T$.

The pivot step. The *circumsphere* $\textsc{cs}(T)$ of a nonempty affinely independent set T is the unique sphere with center in the affine hull $\operatorname{aff}(T)$ that goes through the points in T; its center is called the *circumcenter of* T, denoted by $\textsc{cc}(T)$. A nonempty affinely independent subset T of the set S of given points will be called a *support set*.

Our algorithm steps through a sequence of pairs (T,c), maintaining the invariant that T is a support set and c is the center of a ball B containing S and having T on its boundary. Lemma 1 tells us that we have found the smallest enclosing ball when $c = \textsc{cc}(T)$ and $c \in \operatorname{conv}(T)$. Until this criterion is fulfilled, the algorithm performs an iteration (a pivot step) consisting of a *walking phase* which is preceeded by a *dropping phase* in case $c \in \operatorname{aff}(T)$.

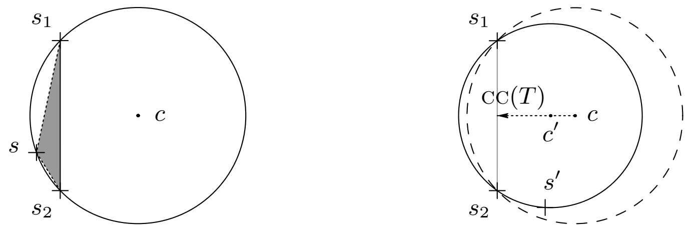

Fig. 1. Dropping s from $T = \{s, s_1, s_2\}$ (left) and walking towards the center $\mathrm{CC}(T)$ of the circumsphere of $T = \{s_1, s_2\}$ until s' stops us (right).

Dropping. If $c \in \mathrm{aff}(T)$, the invariant guarantees that $c = \mathrm{CC}(T)$. Because $c \notin \mathrm{conv}(T)$, there is at least one point $s \in T$ whose coefficient in the affine combination of T forming c is negative. We drop such an s and enter the walking phase with the pair $(T \setminus \{s\}, c)$, see left of Fig. 1.

Walking. If $c \notin \mathrm{aff}(T)$, we move our center on a straight line towards $\mathrm{CC}(T)$. Lemma 3 below establishes that the moving center is always the center of a (progressively smaller) ball with T on its boundary. To maintain the algorithm's invariant, we must stop walking as soon as a new point $s' \in S$ hits the boundary of the shrinking ball. In that case we enter the next iteration with the pair $(T \cup \{s'\}, c')$, where c' is the stopped center; see Fig. 1. If no point stops the walk, the center reaches $\mathrm{aff}(T)$ and we enter the next iteration with $(T, \mathrm{CC}(T))$.

3 The Algorithm in Detail

Let us start with some basic facts about the walking direction from the current center c towards the circumcenter of the boundary points T.

Lemma 3. *Let T be a nonempty affinely independent point set on the boundary of some ball $B(c, r)$, i.e., $T \subset \partial B(c, r) = \partial B(c, T)$. Then*

(i) *the line segment $[c, \mathrm{CC}(T)]$ is orthogonal to* $\mathrm{aff}(T)$*,*
(ii) *$T \subset \partial B(c', T)$ for each $c' \in [c, \mathrm{CC}(T)]$,*
(iii) $\mathrm{radius}(B(\cdot, T))$ *is a strictly monotone decreasing function on* $[c, \mathrm{CC}(T)]$*, with minimum attained at* $\mathrm{CC}(T)$*.*

Note that part (i) of this lemma implies that the circumcenter of T coincides with the orthogonal projection of c onto $\mathrm{aff}(T)$, a fact that will become important for our actual implementation.

When moving along $[c, \mathrm{CC}(T)]$, we have to check for new points to hit the shrinking boundary. The subsequent lemma tells us that all points "behind" $\mathrm{aff}(T)$ are uncritical in this respect, i.e., they cannot hit the boundary and thus cannot stop the movement of the center. Hence, we may ignore these points during the walking phase.

```
procedure seb($S$);
begin
    $c$ := any point of $S$;
    $T := \{p\}$, for a point $p$ of $S$ at maximal distance from $c$;
    while $c \notin \mathrm{conv}(T)$ do
        [ Invariant: $B(c,T) \supset S$, $\partial B(c,T) \supset T$, and $T$ affinely independent ]
        if $c \in \mathrm{aff}(T)$ then drop a point $q$ from $T$ with $\lambda_q < 0$ in (2);
        [ Invariant: $c \notin \mathrm{aff}(T)$ ]
        among the points in $S \setminus T$ that do not satisfy (1) find one, $p$ say,
        that restricts movement of $c$ towards $\mathrm{CC}(T)$ most, if one exists;
        move $c$ as far as possible towards $\mathrm{CC}(T)$;
        if walk has been stopped then $T := T \cup \{p\}$;
    end while;
    return $B(c,T)$;
end seb;
```

Fig. 2. The algorithm to compute $\mathrm{SEB}(S)$.

Lemma 4. *Let T and c as in Lemma 3 and let $p \in B(c,T)$ lie behind* $\mathrm{aff}(T)$*, precisely,*

$$\langle p - c, \mathrm{CC}(T) - c\rangle \geq \langle \mathrm{CC}(T) - c, \mathrm{CC}(T) - c\rangle. \tag{1}$$

Then p is contained in $B(c',T)$ for any $c' \in [c, \mathrm{CC}(T)]$.

It remains to identify which point of the boundary set T should be dropped in case that $c \in \mathrm{aff}(T)$ but $c \notin \mathrm{conv}(T)$. Here are the suitable candidates.

Lemma 5. *Let T and c as in Lemma 3 and assume that $c \in \mathrm{aff}(T)$. Let*

$$c = \sum_{q \in T} \lambda_q q, \quad \sum_{q \in T} \lambda_q = 1 \tag{2}$$

be the affine representation of c with respect to T. If $c \notin \mathrm{conv}(T)$ then $\lambda_p < 0$ for at least one $p \in T$ and any such p satisfies inequality (1) *with T replaced by the reduced set $T \setminus \{p\}$ there.*

Combining Lemmata 4 and 5, we see that if we drop a point with negative coefficient in (2), this point will not stop us in the subsequent walking step.

The Algorithm in detail. Fig. 2 gives a formal description of our algorithm. The correctness follows easily from the previous considerations and we will address the issue of termination soon. Before, let us consider an example in the plane. Figure 3, (a)–(c), depicts all three iterations of our algorithm on a four-point

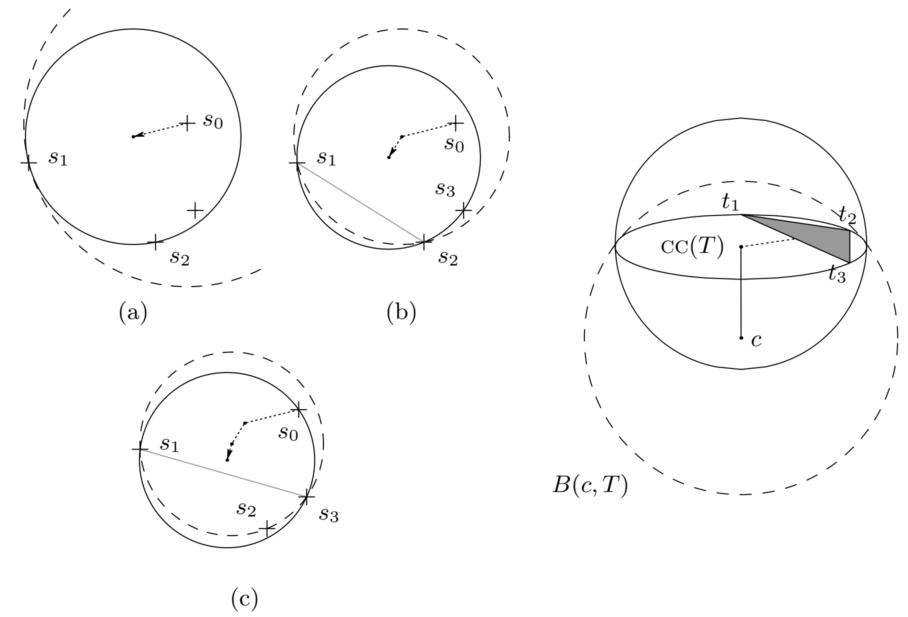

Fig. 3. A full run of the algorithm in 2D (left) and two consecutive steps in 3D (right).

set. Each picture shows the current ball $B(c, T)$ just before (dashed) and right after (solid) the walking phase.

After the initialization $c = s_0$, $T = \{s_1\}$, we move towards the singleton T until s_2 hits the boundary (step (a)). The subsequent motion towards the circumcenter of two points is stopped by the point s_3, yielding a 3-element support (step (b)). Before the next walking we drop the point s_2 from T. The last movement (c) is eventually stopped by s_0 and then the center lies in the convex hull of $T = \{s_0, s_1, s_3\}$.

Observe that the 2-dimensional case obscures the fact that in higher dimensions, the target $\mathrm{CC}(T)$ of a walk need not lie in the convex hull of the support set T. In the right picture of Fig. 3, the current center c first moves to $\mathrm{CC}(T) \notin \mathrm{conv}(T)$, where $T = \{t_1, t_2, t_3\}$. Then, t_2 is dropped and the walk continues towards $\mathrm{aff}(T \setminus \{t_2\})$.

Termination. It is not clear whether the algorithm as stated in Fig. 2 always terminates. Although the radius of the ball clearly decreases whenever the center moves, it might happen that a stopper already lies on the current ball and thus no real movement is possible. In principle, this might happen repeatedly from some point on, i.e., we might run in an infinite cycle, perpetually collecting and dropping points without ever moving the center at all. However, for points in sufficiently general position such infinite loops cannot occur.

Proposition 1. *If for all affinely independent subsets $T \subseteq S$, no point of $S \setminus T$ lies on the circumsphere of T then algorithm* $\mathrm{seb}(S)$ *terminates.*

Proof. Right after a dropping phase, the dropped point cannot be reinserted (Lemmata 4 and 5) and by assumption no other point lies on the current boundary. Thus, the sequence of radii measured right before the dropping steps is strictly decreasing; and since at least one out of d consecutive iterations demands a drop, it would have to take infinitely many values if the algorithm did not terminate. But this is impossible because before a drop, the center c coincides with the circumcenter $\text{CC}(T)$ of one out of finitely many subsets T of S. $\square$

The degenerate case. In order to achieve termination for arbitrary instances, we equip the procedure $\text{seb}(S)$ with the following simple rule, resembling Bland's pivoting rule for the simplex algorithm [13] (for simplicity, we will actually call it *Bland's rule* in the sequel):

> *Fix an arbitrary order on the set S. When dropping a point with negative coefficient in* (2)*, choose the one of smallest rank in the order. Also, pick the smallest-rank point for inclusion in T when the algorithm is simultaneously stopped by more than one point during the walking phase.*

As it turns out, this rule prevents the algorithm from "cycling", i.e., it guarantees that the center of the current ball cannot stay at its position for an infinite number of iterations.

Theorem 1. *Using Bland's rule,* $\text{seb}(S)$ *terminates.*

Proof. Assume for a contradiction that the algorithm cycles, i.e., there is a sequence of iterations where the first support set equals the last and the center does not move. We assume w.l.o.g. that the center coincides with the origin. Let $C \subseteq S$ denote the set of all points that enter and leave the support during the cycle and let among these be m the one of maximal rank.

The key idea is to consider a slightly modified instance X of the SEB problem. Choose a support set $D \not\ni m$ right after dropping m and let $X := D \cup \{-m\}$, mirroring the point m at 0. There is a unique affine representation of the center 0 by the points in $D \cup \{m\}$, where by Bland's rule, the coefficients of points in D are all nonnegative while m's is negative. This gives us a *convex* representation of 0 by the points in X and we may write

$$0 = \langle \sum_{p \in X} \lambda_p p, \text{CC}(I) \rangle = \sum_{p \in D} \lambda_p \langle p, \text{CC}(I) \rangle - \lambda_{-m} \langle m, \text{CC}(I) \rangle. \tag{3}$$

We have introduced the scalar products because of their close relation to criterion (1) of the algorithm. We bound these by considering a support set $I \not\ni m$ just before insertion of the point m. We have $\langle m, \text{CC}(I) \rangle < \langle \text{CC}(I), \text{CC}(I) \rangle$ and by Bland's rule and the maximality of m, there cannot be any other points of C in front of $\text{aff}(I)$; further, all points of D that do not lie in C must, by definition, also lie in I. Hence, we get $\langle p, \text{CC}(I) \rangle \geq \langle \text{CC}(I), \text{CC}(I) \rangle$ for all $p \in I$. Plugging these inequalities into (3) we obtain

$$0 > \big(\sum_{p \in D} \lambda_p - \lambda_{-m} \big) \langle \text{CC}(I), \text{CC}(I) \rangle = (1 - 2\lambda_{-m}) \langle \text{CC}(I), \text{CC}(I) \rangle,$$

which implies $\lambda_{-m} > 1/2$, a contradiction to Lemma 2. $\square$

4 The Implementation

We have programmed our algorithm in C++ using floating point arithmetic. At the heart of this implementation is a dynamic QR-decomposition, which allows updates for point insertion into and deletion from T and supports the two core operations of our algorithm:

- compute the affine coefficients of some $p \in \mathrm{aff}(T)$,
- compute the circumcenter $\mathrm{CC}(T)$ of T.

Our implementation, however, does not tackle the latter task in the present formulation. From part (i) of Lemma 3 we know that the circumcenter of T coincides with the orthogonal projection of c onto $\mathrm{aff}(T)$ and this is how we actually compute $\mathrm{CC}(T)$ in practice. Using this reformulation, we shall see that the two tasks at hand are essentially the same problem. We briefly sketch the main ideas behind our implementation, which are combinations of standard concepts from linear algebra.

Computing orthogonal projections and affine coefficients. In order to apply linear algebra, we switch from the affine space to a linear space by fixing some "origin" q_0 of $T = \{q_0, q_1, \ldots, q_r\}$ and defining the relative vectors $a_i = q_i - q_0$, $1 \leq i \leq r$. For the matrix $A = [a_1, \ldots, a_r]$, which has full column rank r, we maintain a QR-decomposition $QR = A$, that is, an orthogonal $d \times d$ matrix Q and a rectangular $d \times r$ matrix $R = \left[\begin{smallmatrix}\hat{R}\\0\end{smallmatrix}\right]$, where $\hat{R}$ is square upper triangular.

Recall how such a decomposition can be used to "solve" an overdetermined system of linear equations $Ax = b$ (the right hand side $b \in \mathbb{R}^d$ being given) [14, Sec. 5.3]: Using orthogonality of Q, first compute $y := Q^{-1}b = Q^T b$; then discard the lower $d - r$ entries of $y = \left[\begin{smallmatrix}\hat{y}\\ *\end{smallmatrix}\right]$ and evaluate $\hat{R}x^* = \hat{y}$ through back substitution. The resulting x^* is known to minimize the residual $||Ax - b||$, which means that Ax^* is the unique point in $\mathrm{im}(A)$ closest to b. In other words, Ax^* is the orthogonal projection of b onto $\mathrm{im}(A)$.

This already solves both our tasks. The affine coefficients of some point $p \in \mathrm{aff}(T)$ are exactly the entries of the approximation x^* for the shifted equation $Ax = p - q_0$ (the missing coefficient of q_0 follows directly from the others); further, for arbitrary p, Ax^* is just the orthogonal projection of p onto $\mathrm{aff}(T)$. With a QR-decomposition of A at hand, these calculations can be done in quadratic time (in the dimension) as seen above.

The computation of orthogonal projections even allows some improvement. By reformulating it as a Gram-Schmidt-like procedure and exploiting a duality in the QR-decomposition, we can obtain running times of order $\min\{\mathrm{rank}(A), d - \mathrm{rank}(A)\} \cdot d$. This should, however, be seen only as a minor technical modification of the general procedure described above.

Maintaining the QR-decomposition. Of course, computing orthogonal projections and affine coefficients in quadratic time would not be of much use if we had to set up a complete QR-decomposition of A in cubic time each time a point is inserted into or removed from the support set T—the basic modifications applied to T in each iteration.

Golub and van Loan [14, Sec. 12.5] describe how to update a QR-decomposition in quadratic time, using *Givens rotations*. We briefly present those techniques and show how they apply to our setting of points in affine space.

Adding a point p to T corresponds to appending the column vector $u = p - q_0$ to A yielding a matrix A' of rank $r + 1$. To incorporate this change into Q and R, append the column vector $w = Q^T u$ to R so that the resulting matrix R' satisfies the desired equation $QR' = A'$. But then R' is no longer in upper triangular form. This defect can be repaired by application of $d - r - 1$ Givens rotations $G_{r+1}, \ldots, G_{d-1}$ from the left (yielding the upper triangular matrix $R'' = G_{r+1} \cdots G_{d-1} R'$). Applying the transposes of these orthogonal matrices to Q from the right (giving the orthogonal matrix $Q' = QG_{d-1}^T \cdots G_{r+1}^T$) then provides consistency again ($Q'R'' = A'$). Since multiplication with a Givens rotator effects only two columns of Q, the overall update takes $\mathcal{O}(d^2)$ steps.

Removing a point from T works similarly to insertion. Simply erase the corresponding column from R and shift the higher columns one place left. This introduces one subdiagonal entry in each shifted column which again can be zeroed with a linear number of Givens rotations, resulting in a total number of $\mathcal{O}(dk)$ steps, where $k < d$ is the number of columns shifted.

The remaining task of removing the origin q_0 from T (which does not work in the above fashion since q_0 does not correspond to a particular column of A) can also be dealt with efficiently (using an appropriate rank-1-update). Thus, all updates can be realized in quadratic time. (Observe that we used the matrices A and A' only for explanatory purposes. Our program does not need to store them explicitly but performs all computation on Q and R directly.)

Stability, termination, and verification. As for numerical stability, QR-decomposition itself behaves nicely since all updates on Q and R are via orthogonal Givens rotators [14, Sec. 5.1.10]. However, for our coefficient computations and orthogonal projections to work, we have to avoid degeneracy of the matrix R. Though in theory this is guaranteed through the affine independence of T, we introduce a stability threshold in our floating point implementation to protect against unfortunate interaction of rounding errors and geometric degeneracy in the input set. This is done by allowing only such stopping points to enter the support set that are not closer to the current affine hull of T than some ϵ. (Remember that points behind $\operatorname{aff}(T)$ are ignored anyway because of Lemma 4.)

Our code is equipped with a result checker, which upon termination verifies whether all points of S really lie in the computed ball and whether the support points all lie on the boundary. We further do some consistency checking on the final QR-decomposition by determining the affine coefficients of each individual point in T. In all our test runs, the overall error computed thus was never larger than 10^{-12}, about 10^4 times the machine precision.

Finally, we note that while Bland's rule guarantees termination in theory, it is slow in practice. As with LP-solvers, we resort to a different heuristic in practice, which yields very satisfactory running-time behavior and has the fur-

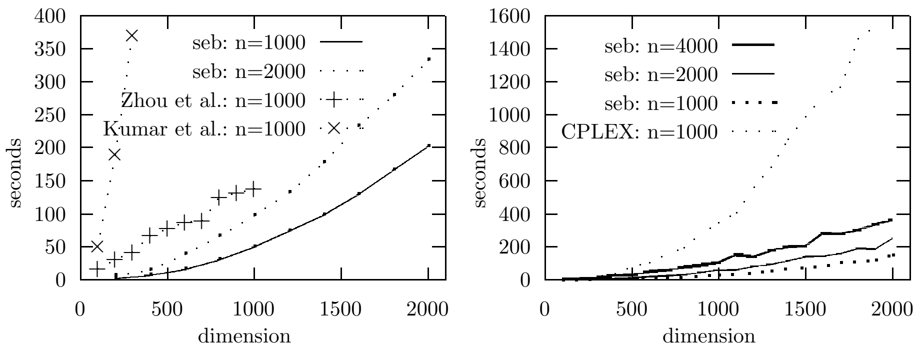

Fig. 4. (a) Our Algorithm seb, Zhou et al., and Kumar et al. on uniform distribution, (b) Algorithm seb on normal distribution.

ther advantage of greater robustness with respect to roundoff errors: The rule for deletion from T in the dropping phase is to pick the point t of minimal λ_t in (2). For insertion into T in the walking phase we consider, roughly speaking, all points of S which would result in almost the same walking distance as the actual stopping point p (allowing only some additional ϵ) and choose amongst these the one farthest from $\mathrm{aff}(T)$.

Note that our stability threshold and the above selection threshold are a concept entirely different from the approximation thresholds of [9] and [11]. Our ϵ's do not enter the running time and choosing them close to machine precision already resulted in very stable behavior in practice.

5 Testing Results

We have run the floating point implementation of our algorithm on random point sets drawn from different distributions:

- uniform distribution in the unit cube,
- normal distribution with standard deviation 1,
- uniform distribution on the surface of the unit sphere, with each point perturbed in direction of the center by some number drawn uniformly at random from a small interval $[-\delta, +\delta)$.

The tests were performed on a 480Mhz Sun Ultra 4 workstation.

Figure 4(a) shows the running time of our algorithm on instances with 1,000 and 2,000 points drawn uniformly at random from the unit cube in up to 2,000 dimensions. For reference, we included the running times given in [9] and [11] on similar point sets. However, these data should be interpreted with care: they are taken from the respective publications and we have not recomputed them under our conditions (the code of [11] is not available). However, the hardware was comparable to ours. Also observe that Zhou et al. implemented their algorithm

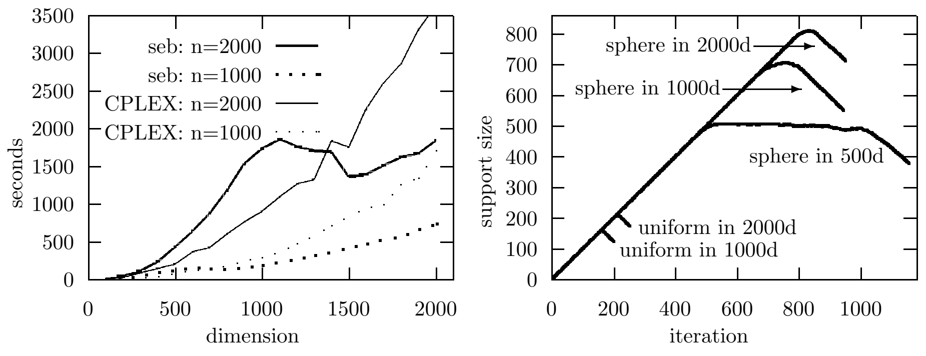

Fig. 5. (a) Algorithm seb and CPLEX on almost spherical distribution ($\delta = 10^{-4}$), (b) Support-size development depending on the number of iterations, for 1,000 points in different dimensions, distributed uniformly and almost spherically ($\delta = 10^{-3}$)

in Matlab, which is not really comparable with our C++ implementation. Still, the figures give an idea of the relative performances of the three methods for several hundred dimensions.

On the normal distribution our algorithm performs similarly as for the uniform distribution. Figure 4(b) contains plots for sets of 1,000, 2,000, and 4,000 points. We compared these results to the performance of the general-purpose QP-solver of CPLEX (version 6.6.0, which is the latest version available to us). We outperform CPLEX by wide margins. (The running times of CPLEX on 2,000 and 4,000 points, which are not included in the figure, scale almost uniformly by a factor of 2 resp. 4 on the tested data.) Again, these results are to be seen in the proper perspective; it is of course not surprising that a dedicated algorithm is superior to a general-purpose code; moreover, the QP arising from the SEB problem are not the "typical" QP that CPLEX is tuned for. Still, the comparison is necessary in order to argue that off-the-shelf methods cannot successfully compete with our approach.

The most difficult inputs for our code are sets of (almost) cospherical points. In such situations, it typically happens that many points enter the support set in intermediate steps only to be dropped again later. This is quite different to the case of the normal distribution, where a large fraction of the points will never enter the support and the algorithm needs much fewer iterations. Figure 5 compares our algorithm and again CPLEX on almost-spherical instances. For smaller dimensions, CPLEX is faster, but starting from roughly d = 1,500, we again win.

The papers of Zhou et al. as well as Kumar et al. do not contain tests with almost-cospherical points. In case of the QP-solver of CPLEX, our observation is that the point distribution has no major influence on the runtime; in particular, cospherical points do not give rise to particularly difficult inputs. It might be the case that the same is true for the methods of Zhou et al. and Kumar et al., but it would still be interesting to verify this on concrete examples.

To provide more insight into the actual behavior of our algorithm, Figure 5(b) shows the support-size development for complete runs on different inputs. In all our tests we observed that the computation starts with a long point-collection phase during which no dropping occurs. This initial phase is followed by an intermediate period of dropping and inserting during which the support size changes only very little. Finally, there is a short dropping phase almost without new insertions. The intermediate phase is usually quite short, except for almost spherical distributions with n considerably larger than d. The explanation for this phenomenon being that only for such distributions there are many candidate points for the final support set which are repeatedly dropped and inserted several times. With growing dimension, more and more points of an almost-spherical distribution belong into the final support set, leaving only few points ever to be dropped. We thank Emo Welzl for pointing out this fact, thus explaining the nonmonotone running-time behavior of our algorithm in Fig. 5(a).

References

1. Megiddo, N.: Linear-time algorithms for linear programming in R^3 and related problems. SIAM J. Comput. **12** (1983) 759–776
2. Dyer, M.E.: A class of convex programs with applications to computational geometry. In: Proc. 8th Annu. ACM Sympos. Comput. Geom. (1992) 9–15
3. Welzl, E.: Smallest enclosing disks (balls and ellipsoids). In Maurer, H., ed.: New Results and New Trends in Computer Science. Volume 555 of Lecture Notes Comput. Sci. Springer-Verlag (1991) 359–370
4. Gärtner, B.: Fast and robust smallest enclosing balls. In: Proc. 7th Annual European Symposium on Algorithms (ESA). Volume 1643 of Lecture Notes Comput. Sci., Springer-Verlag (1999) 325–338
5. Gärtner, B., Schönherr, S.: An efficient, exact, and generic quadratic programming solver for geometric optimization. In: Proc. 16th Annu. ACM Sympos. Comput. Geom. (2000) 110–118
6. Ben-Hur, A., Horn, D., Siegelmann, H.T., Vapnik, V.: Support vector clustering. Journal of Machine Learning Research **2** (2001) 125–137
7. Bulatov, Y., Jambawalikar, S., Kumar, P., Sethia, S.: Hand recognition using geometric classifiers (2002) Abstract of presentation for the DIMACS Workshop on Computational Geometry (Rutgers University).
8. Goel, A., Indyk, P., Varadarajan, K.R.: Reductions among high dimensional proximity problems. In: Symposium on Discrete Algorithms. (2001) 769–778
9. Kumar, P., Mitchell, J.S.B., Yıldırım, E.A.: Computing core-sets and approximate smallest enclosing hyperspheres in high dimensions (2003) To appear in the Proceedings of ALENEX'03.
10. ILOG, Inc.: ILOG CPLEX 6.5 user's manual (1999)
11. Zhou, G., Toh, K.C., Sun, J.: Efficient algorithms for the smallest enclosing ball problem. Manuscript (2002)
12. Hopp, T.H., Reeve, C.P.: An algorithm for computing the minimum covering sphere in any dimension. Technical Report NISTIR 5831, National Institute of Standards and Technology (1996)
13. Chvátal, V.: Linear programming. W. H. Freeman, New York, NY (1983)
14. Golub, G.H., van Loan, C.F.: Matrix Computations. third edn. Johns Hopkins University Press (1996)