Fast and Robust Smallest Enclosing Balls *

Bernd Gärtner

Institut für Theoretische Informatik, ETH Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland (`gaertner@inf.ethz.ch`)

Abstract. I describe a `C++` program for computing the smallest enclosing ball of a point set in d-dimensional space, using floating-point arithmetic only. The program is very fast for $d \leq 20$, robust and simple (about 300 lines of code, excluding prototype definitions). Its new features are a pivoting approach resembling the simplex method for linear programming, and a robust update scheme for intermediate solutions. The program with complete documentation following the *literate programming* paradigm [3] is available on the Web.[1]

1 Introduction

The smallest enclosing ball (or Euclidean 1-center) problem is a classical problem of computational geometry. It has a long history which dates back to 1857 when Sylvester formulated it for points in the plane [8]. The first optimal linear-time algorithm for fixed dimension was given by Megiddo in 1982 [4]. In 1991, Emo Welzl developed a simple randomized method to solve the problem in expected linear time [9]. In contrast to Megiddo's method, his algorithm is easy to implement and very fast in practice, for dimensions 2 and 3. In higher dimensions, a heuristic *move-to-front* variant considerably improves the performance.

The roots of the program I will describe go back to 1991 when I first implemented Emo Welzl's new method. Using the move-to-front variant I was able to solve problems on 5000 points up to dimension $d = 10$ (see [9] for all results). Back then, the program was written in `MODULA-2` (the language I had learned in my undergraduate CS courses), and it was running on a 80386 PC with 20 MHz.

After the algorithm and the implementation results had been published, we constantly received requests for source code, from an amazingly wide range of application areas. To name some, there was environmental science (design and optimization of solvents), pattern recognition (finding reference points), biology (proteine analysis), political science (analysis of party spectra), mechanical engineering (stress optimization) and computer graphics (ray tracing, culling).

Soon it became clear that the `MODULA-2` source was not of great help in serving these requests; people wanted `C` or `C++` code. Independently, at least two

* This work was supported by grants from the Swiss Federal Office for Education and Science (Projects ESPRIT IV LTR No. 21957 CGAL and No. 28155 GALIA).

[1] `http://www.inf.ethz.ch/personal/gaertner/miniball.html`

persons ported the undocumented program to C. (One of them remarked that "although we don't understand the complete procedure, we are confident that the algorithm is perfect".) Vishwa Ranjan kindly made his carefully adapted C program accessible to me; subsequently, I was able to distribute a C version. (An independent implementation by David Eberly for the cases $d = 2, 3$ based on [9] is available online. [2])

Another shortcoming of my program—persisting in the C code—was the lack of numerical stability, an issue not yet fashionable in computational geometry at that time. The main primitive of the code was to solve a linear system, and this was done with plain Gaussian elimination, with no special provisions to deal with ill-conditioned systems. In my defense I must say that the code was originally only made to get test results for Emo's paper, and in the tests with the usual random point sets I did for that, I discovered no problems, of course.

Others did. David White, who was developing code for the more general problem of finding the smallest ball enclosing balls, noticed unstable behavior of my program, especially in higher dimensions. In his code, he replaced the naive Gaussian elimination approach by a solution method based on *singular value decomposition* (SVD). This made his program pretty robust, and a C++ version (excluding the SVD routines which are taken from *Numerical Recipes in C* [6]) is available from David White's Web page. [3]

Meanwhile, I got involved in the CGAL project, a joint effort of seven European sites to build a C++ library of computational geometry algorithms.[4] To prepare my code for the library, I finally wrote a C++ version of it from scratch. As the main improvement, the code was no longer solving a complete linear system in every step, but was updating previous imformation instead. A "CGALized" version of this code is now contained in the library, and using it together with any exact (multiple precision) number type results in error-free computations.

Still, the numerical problems arising in floating-point computations were not solved. Stefan Gottschalk, one of the first users of my new C++ code, encountered singularities in the update routines, in particular if input points are (almost) cospherical, very close together or even equal. The effect is that center and squared radius of the ball maintained by the algorithm can become very large or even undefined due to exponent overflow, even though the smallest enclosing ball problem itself is well-behaved in the sense that small perturbations of the input points have only a small influence on the result.

As it turned out, previous implementations suffered from an inappropriate representation of intermediate balls that ignores the good-naturedness of the problem. The new representation scheme respects the underlying geometry—it actually resulted from a deeper understanding of the geometric situation— and solves most of the problems.

The second ingredient is a new high-level algorithm replacing the move-to-front method. Its goal is to decrease the overall number of intermediate solutions

[2] `http://www.cs.unc.edu/~eberly/`

[3] `http://vision.ucsd.edu/~dwhite`

[4] `http://www.cs.uu.nl/CGAL`

computed during the algorithm. This is achieved by reducing the problem to a small number of calls to the move-to-front method, with only a small point set in each call. These calls can then be interpreted as 'pivot steps' of the method. The advantages are a substantial improvement in runtime for dimensions $d \geq 10$, and much more robust behavior.

The result is a program which I think reaches a high level of efficiency and stability per lines of code. In simplicity, it is almost comparable to the popular approximation algorithms from the *Graphics Gems* collection [7, 10]; because the latter usually compute suboptimal balls, the authors stress their simplicity as the main feature. The code presented here shares this feature, while computing the optimal ball.

2 The Algorithms

Given an n-point set $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^d$, let $\text{MB}(P)$ denote the ball of smallest radius that contains P. $\text{MB}(P)$ exists and is unique. For $P, B \subseteq \mathbb{R}^d$, $P \cap B = \emptyset$, let $\text{MB}(P, B)$ be the smallest ball that contains P and has all points of B on its boundary. We have $\text{MB}(P) = \text{MB}(P, \emptyset)$, and if $\text{MB}(P, B)$ exists, it is unique. Finally, define $\overline{\text{MB}}(B) := \text{MB}(\emptyset, B)$ to be the smallest ball with all points of B on the boundary (if it exists).

A support set of (P, B) is an inclusion-minimal subset of P with $\text{MB}(P, B) = \text{MB}(S, B)$. If the points in B are affinely independent, there always exists a support set of size at most $d + 1 - |B|$, and we have $\text{MB}(S, B) = \overline{\text{MB}}(S \cup B)$.

If $p \notin \text{MB}(P, B)$, then p lies on the boundary of $\text{MB}(P \cup \{p\}, B)$, provided the latter exists—that means, $\text{MB}(P \cup \{p\}, B) = \text{MB}(P, B \cup \{p\})$. All this is well-known, see e.g. [9] and the references there.

The basis of our method is Emo Welzl's *move-to-front* heuristic to compute $\text{MB}(P, B)$ if it exists[9]. The method keeps the points in an ordered list L which gets reorganized as the algorithm runs. Let L_i denote the length-i prefix of the list, p^i the element at position i in L. Initially, $L = L_n$ stores the points of P in random order.

Algorithm 1.

```
mtf_mb(L_n, B):
   (* returns MB(L_n, B) *)
   mb := MB‾(B)
   IF |B| = d + 1 THEN
      RETURN mb
   END
   FOR i = 1 TO n DO
      IF p^i ∉ mb THEN
         mb := mtf_mb(L_{i-1}, B ∪ {p^i})
         update L by moving p^i to the front
      END
   END
   RETURN mb
```

This algorithm computes $\text{MB}(P, B)$ incrementally, by adding one point after another from the list. One can prove that during the call to $\texttt{mtf\_mb}(L_n, \emptyset)$, all sets B that come up in recursive calls are affinely independent. Together with the above mentioned facts, this ensures the correctness of the method. By induction, one can also show that upon termination, a support set of (P, B) appears as a prefix L_s of the list L, and below we will assume that the algorithm returns the size s along with mb.

The practical efficiency comes from the fact that 'important' points (which for the purpose of the method are points outside the current ball) are moved to the front and will therefore be processed early in subsequent recursive calls. The effect is that the ball maintained by the algorithm gets large fast.

The second algorithm uses the move-to-front variant only as a subroutine for small point sets. Large-scale problems are handled by a *pivoting* variant which in every iteration adds the point which is most promising in the sense that it has largest distance from the current ball. Under this scheme, the ball gets large even faster, and the method usually terminates after very few iterations. (As the test results in Section 5 show, the move-to-front variant will still be faster for d not too large, but there are good reasons to prefer the pivoting variant in any case.)

Let $e(p, \text{mb})$ denote the *excess* of p w.r.t. mb, defined as $\|p-c\|^2 - r^2$, c and r^2 the center and squared radius of mb.

Algorithm 2.

$\texttt{pivot\_mb}(L_n)$:
  (* returns $\text{MB}(L_n)$ *)
  $t := 1$
  $(\text{mb}, s) := \texttt{mtf\_mb}(L_t, \emptyset)$
  REPEAT
    (* Invariant: $\text{mb} = \text{MB}(L_t) = \overline{\text{MB}}(L_s), s \leq t$ *)
    choose $k > t$ with $e := e(p^k, \text{mb})$ maximal
    IF $e > 0$ THEN
      $(\text{mb}, s') := \texttt{mtf\_mb}(L_s, \{p^k\})$
      update L by moving p^k to the front
      $t := s + 1$
      $s := s' + 1$
    END
  UNTIL $e \leq 0$
  RETURN mb

Because mb gets larger in every iteration, the procedure eventually terminates. The computation of (mb, s') can be viewed as a 'pivot step' of the method, involving at most $d + 2$ points. The choice of k is done according to a heuristic 'pivot rule', with the intention of keeping the overall number of pivot steps small. With this interpretation, the procedure $\texttt{pivot\_mb}$ is similar in spirit to the simplex method for linear programming [1], and it has in fact been designed with regard to the simplex method's efficiency in practice.

3 The Primitive Operation

During a call to algorithm `pivot_mb`, all nontrivial computations take place in the primitive operation of computing $\overline{\mathrm{MB}}(B)$ for a given set B in the subcalls to `mtf_mb`. The algorithm guarantees that B is always a set of affinely independent points, from which $|B| \leq d+1$ follows. In that case, $\overline{\mathrm{MB}}(B)$ is determined by the unique circumsphere of the points in B with center restricted to the affine hull of B. This means, the center c and squared radius r^2 satisfy the following system of equations, where $B = \{q_0, \ldots, q_{m-1}\}, m \leq d+1$.

$$
\begin{aligned}
(q_i - c)^T(q_i - c) &= r^2, \quad i = 0, \ldots m-1, \\
\sum_{i=0}^{m-1} \lambda_i q_i &= c, \\
\sum_{i=0}^{m-1} \lambda_i &= 1.
\end{aligned}
$$

Defining $Q_i := q_i - q_0$, for $i = 0, \ldots, m-1$ and $C := c - q_0$, the system can be rewritten as

$$
\begin{aligned}
C^T C &= r^2, \\
(Q_i - C)^T(Q_i - C) &= r^2, \quad i = 1, \ldots, m-1, \\
\sum_{i=1}^{m-1} \lambda_i Q_i &= C.
\end{aligned}
\tag{1}
$$

Substituting C with $\sum_{i=1}^{m-1} \lambda_i Q_i$ in the equations (1), we deduce a linear system in the variables $\lambda_1, \ldots, \lambda_{m-1}$ which we can write as

$$
A_B \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_{m-1} \end{pmatrix} = \begin{pmatrix} Q_1^T Q_1 \\ \vdots \\ Q_{m-1}^T Q_{m-1} \end{pmatrix}, \tag{2}
$$

where

$$
A_B := \begin{pmatrix} 2Q_1^T Q_1 & \cdots & 2Q_1^T Q_{m-1} \\ \vdots & & \vdots \\ 2Q_1^T Q_{m-1} & \cdots & 2Q_{m-1}^T Q_{m-1} \end{pmatrix}. \tag{3}
$$

Computing the values of $\lambda_1, \ldots, \lambda_{m-1}$ amounts to solving the linear system (2). C and r^2 are then easily obtained via

$$
C = \sum_{i=1}^{m-1} \lambda_i Q_i, \quad r^2 = C^T C. \tag{4}
$$

We refer to C as the *relative center* w.r.t. the (ordered) set B.

4 The Implementation

Algorithms 1 and 2 are implemented in a straightforward manner, following the pseudocode given above. In case of algorithm `mtf_mb`, the set B does not appear as a formal parameter but is updated before and after the recursive call by 'pushing' resp. 'popping' the point p^i. This stack-like behavior of B also makes it possible to implement the primitive operation in a simple, robust and efficient way. More precisely, the algorithm maintains a device for solving system (2) which can conveniently be updated if B changes. The update is easy when element p^i is removed from B—we just need to remember the status prior to the addition of p^i. In the course of this addition, however, some real work is necessary.

A possible device for solving system (2) is the explicit inverse A_B^{-1} of the matrix A_B defined in (3), along with the vector

$$v_B := \begin{pmatrix} Q_1^T Q_1 \\ \vdots \\ Q_{m-1}^T Q_{m-1} \end{pmatrix}.$$

Having this inverse available, it takes just a matrix-vector multiplication to obtain the values $\lambda_1, \ldots, \lambda_{m-1}$ that define C via (4).

Assume B is enlarged by pushing another point q_m. Define $B' = B \cup \{q_m\}$. Let's analyze how $A_{B'}^{-1}$ can be obtained from A_B^{-1}. We have

$$A_{B'} = \left(\begin{array}{ccc|c} & & & 2Q_1^T Q_m \\ & A_B & & \vdots \\ & & & 2Q_{m-1}^T Q_m \\ \hline 2Q_1^T Q_m & \cdots & 2Q_{m-1}^T Q_m & 2Q_m^T Q_m \end{array}\right),$$

and it is not hard to check that this equation can be written as

$$A_{B'} = L \left(\begin{array}{ccc|c} & & & 0 \\ & A_B & & \vdots \\ & & & 0 \\ \hline 0 & \cdots & 0 & z \end{array}\right) L^T,$$

where

$$L = \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline \mu_1 & \cdots & \mu_{m-1} & 1 \end{array}\right), \quad \mu := \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_{m-1} \end{pmatrix} = A_B^{-1} \begin{pmatrix} 2Q_1^T Q_m \\ \vdots \\ 2Q_{m-1}^T Q_m \end{pmatrix} \tag{5}$$

and

$$z = 2Q_m^T Q_m - (2Q_1^T Q_m, \cdots, 2Q_{m-1}^T Q_m) A_B^{-1} \begin{pmatrix} 2Q_1^T Q_m \\ \vdots \\ 2Q_{m-1}^T Q_m \end{pmatrix}. \tag{6}$$

This implies

$$A_{B'}^{-1} = (L^T)^{-1} \left(\begin{array}{ccc|c} & & & 0 \\ & A_B^{-1} & & \vdots \\ & & & 0 \\ \hline 0 & \cdots & 0 & 1/z \end{array} \right) L^{-1}, \tag{7}$$

where

$$L^{-1} = \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline -\mu_1 & \cdots & -\mu_{m-1} & 1 \end{array} \right).$$

Expanding (7) then gives the desired update formula

$$A_{B'}^{-1} = \left(\begin{array}{c|c} A_B^{-1} + \mu\mu^T/z & -\mu/z \\ \hline -\mu^T/z & 1/z \end{array} \right), \tag{8}$$

with μ and z as defined in (5) and (6).

Equation (8) shows that A_B may become ill-conditioned (and the entries of A_B^{-1} very large and unreliable), if z evaluates to a very small number. The subsequent lemma develops a geometric interpretation of z from which we can see that this happens exactly if the new point q_m is very close to the affine hull of the previous ones. This can be the case e.g. if input points are very close together or even equal. To deal with such problems, we need a device that stays bounded in every update operation.

As it turns out, a suitable device is the $(d \times d)$-matrix

$$M_B := 2 Q_B A_B^{-1} Q_B^T,$$

where

$$Q_B := (Q_1 \cdots Q_{m-1})$$

stores the points Q_i as columns. Lemma 1 below proves that the entries of M_B stay bounded, no matter what. We will also see how the new center is obtained from M_B, which is not clear anymore now.

Lemma 1.

(i) With μ as in (5), we have

$$\sum_{i=1}^{m-1} \mu_i Q_i = \bar{Q}_m,$$

where $\bar{Q}_m$ is the projection of Q_m onto the subspace spanned by the Q_i.

(ii) $M_B Q_m = \bar{Q}_m$.

(iii) $z = 2(Q_m - \bar{Q}_m)^T (Q_m - \bar{Q}_m)$, i.e. z is twice the distance from Q_m to its projection.

(iv) If C and r^2 are relative center and squared radius w.r.t. B, then the new relative center C' and squared radius r'^2 (w.r.t. B') satisfy

$$C' = C + \frac{e}{z}(Q_m - \bar{Q}_m), \tag{9}$$

$$r'^2 = r^2 + \frac{e^2}{2z}, \tag{10}$$

where

$$e = (Q_m - C)^T(Q_m - C) - r^2.$$

(v) M_B is updated according to

$$M_{B'} = M_B + \frac{2}{z}(Q_m - \bar{Q}_m)(Q_m - \bar{Q}_m)^T. \tag{11}$$

The proof involves only elementary calculations and is omitted here. Property (ii) gives M_B an interpretation as a linear function: M_B is the projection onto the linear subspace spanned by $Q_1, \ldots, Q_{m-1}$. Furthermore, property (v) implies that M_B stays bounded. This of course does not mean that no 'bad' errors can occur anymore. In (11), small errors in $Q_m - \bar{Q}_m$ can get hugely amplified if z is close to zero. Still, M_B degrades gracefully in this case, and the typical relative error in the final ball is by an order of magnitude smaller if the device M_B is used instead of A_B^{-1}.

The lemma naturally suggests an algorithm to obtain C', r'^2 and $M_{B'}$ from C, r^2 and M_B, using the values $\bar{Q}_m, e$ and z.

As already mentioned, even M_B may get inaccurate, in consequence of a very small value of z before. The strategy to deal with this is very simple: we ignore push operations leading to such dangerously small values! In the ambient algorithm `mtf_mb` this means that the point to be pushed is treated as if it were inside the current ball (in `pivot_mb` the push operation is never dangerous, because we push onto an empty set B). Under this scheme, it could happen that points end up outside the final ball computed by `mtf_mb`, but they will not be very far outside, if we choose the threshold for z appropriately.

The criterion is that we ignore a push operation if and only if the *relative* size of z is small, meaning that

$$\frac{z}{r^2_{curr}} < \varepsilon \tag{12}$$

for some constant ε, where r^2_{curr} is the current squared radius. Now consider a subcall to `mtf_mb`$(L_s, \{p^k\})$ inside the algorithm `pivot_mb`, and assume that a point $p \in L_s$ ends up outside the ball mb_0 with support set S_0 and radius r_0 computed by this subcall.

One can check that after the last time the query '$p^i \notin \mathrm{mb}$?' has been executed with p^i being equal to p in `mtf_mb`, no successful push operations have occured anymore. It follows that $\mathrm{mb} = \mathrm{mb}_0$ in this last query, the query had a positive answer (because p lies outside), and the subsequent push operation failed. This means, we had $z/r_0^2 < \varepsilon$ at that time.

Let r_{max} denote the radius of $\text{MB}(L_s, \{p^k\})$. Because of (10), we also had $e^2/2z \leq r_{max}^2$ at the time of the failing push operation, where e is the excess of p w.r.t. a ball $\overline{\text{MB}}(B \cup \{p^k\})$ with $B \subseteq S_0$. We then get

$$\left(\frac{e}{r_{max}^2}\right)^2 \leq \frac{2z}{r_{max}^2} \leq \frac{2z}{r_0^2} \leq 2\varepsilon.$$

Assuming that r_{max} is not much larger than r_0 (we expect push operations to fail rather at the end of the computation, when the ball is already large), we can argue that

$$\frac{e}{r_0^2} = O(\sqrt{\varepsilon}).$$

Moreover, because mb_0 contains the intersection of $\overline{\text{MB}}(B \cup \{p^k\})$ with the affine hull of $B \cup \{p^k\}$, to which set p is quite close due to z being small, we also get

$$\frac{e_0}{r_0^2} = O(\sqrt{\varepsilon}), \tag{13}$$

where e_0 is the excess of p w.r.t. the final ball mb_0, as desired. This argument is not a strict proof for the correctness of our rejection criterion, but it explains why the latter works well in practice. In the code, ε is chosen as 10^{-32}. Because of (13), the relative error of a point w.r.t. the final ball is then expected to stay below 10^{-16} in magnitude. The latter value is the relative accuracy in the typical situations where the threshold criterion is not applied by the algorithm at all. Thus, ε is chosen in such a way that even when the criterion comes in, the resulting error does not go up.

Checking

While it is easy to verify that the computed ball is admissible in the sense that it contains all input points and has all support points on the boundary (approximately), its optimality does not yet follow from this; if there are less than $d + 1$ support points, many balls are admissible with respect to this definition. The following lemma gives an optimality condition.

Lemma 2. *Let S be a set of affinely independent points. $\overline{\text{MB}}(S)$ is the smallest enclosing ball of S if and only if its center lies in the convex hull of S.*

The statement seems to be folklore and can be proved e.g. by using the Karush-Kuhn-Tucker optimality conditions for constrained optimization [5], or by elementary methods.

From Section 2 we know that the algorithm should compute a support set S that behaves according to the lemma; still, we would like to have a way to check this in order to safeguard against numerical errors that may lead to admissible balls which are too large. Under the device A_B^{-1}, this is very simple—the coefficients λ_i we extract from system (2) in this case give us the desired information: exactly if they are all nonnegative, S defines the optimal ball.

The weakness in this argumentation is that due to (possibly substantial) errors in A_B^{-1}, the λ_i might appear positive, although they are not. One has to be aware that "checking" in this case only adds more plausibility to a seemingly correct result. Real checking would ultimately require the use of exact arithmetic, which is just not the point of this code.

Still, if the plausibility test fails (and some λ_i turn out to be far below zero), we *do* know that something went wrong, which is important information in evaluating the code.

Unfortunately, in using the improved device M_B during the computations, we do not have immediate access to the λ_i. To obtain them, we express C as well as the points $Q_1, \ldots, Q_{m-1}$ with respect to a different basis of the linear span of $Q_1, \ldots, Q_{m-1}$. In this representation, the linear combination of the Q_i that defines C will be easy to deduce.

The basis we use will be the set of (pairwise orthogonal) vectors $Q_i - \bar{Q}_i, i = 1, \ldots, m-1$. From the update formula for the center (9) we immediately deduce that

$$C = \sum_{i=1}^{m-1} f_i(Q_i - \bar{Q}_i),$$

where f_i is the value e/z that was computed according to Lemma 1(iv) when pushing q_i. This means, the coordinates of C in the new basis are $(f_1, \ldots, f_{m-1})$.

To get the representations of the Q_i, we start off by rewriting M_B as

$$M_B = \sum_{k=1}^{m-1} \frac{2}{z_k}(Q_k - \bar{Q}_k)(Q_k - \bar{Q}_k)^T,$$

which follows from Lemma 1(v). Here, z_k denotes the value z we got when pushing point q_k.

Now consider the point Q_i. We need to know the coefficient α_{ik} of $Q_k - \bar{Q}_k$ in the representation

$$Q_i = \sum_{k=1}^{m-1} \alpha_{ik}(Q_k - \bar{Q}_k).$$

With

$$M_{B^i} := \sum_{k=1}^{i} \frac{2}{z_k}(Q_k - \bar{Q}_k)(Q_k - \bar{Q}_k)^T \tag{14}$$

we get

$$M_{B^i} Q_i = Q_i$$

(after adding q_i to B, Q_i projects to itself). Via (14), this entails $\alpha_{i,i+1} = \cdots = \alpha_{i,m-1} = 0$ and

$$\alpha_{ik} = \frac{2}{z_k}(Q_k - \bar{Q}_k)^T Q_i, \quad k \leq i.$$

In particular we get $\alpha_{ii} = 1$. The coefficients λ_i in the equation

$$C = \sum_{i=1}^{m-1} \lambda_i Q_i$$

are now easy to compute in the new representations of C and the Q_i we have just developed. For this, we need to solve the linear system

$$\begin{pmatrix} a_{11} & \cdots & a_{m-1,1} \\ \vdots & & \vdots \\ a_{1,m-1} & \cdots & a_{m-1,m-1} \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_{m-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_{m-1} \end{pmatrix}$$

This system is triangular—everything below the diagonal is zero, and the entries on the diagonal are 1. So we can get the λ_i by a simple back substitution, according to

$$\lambda_i = f_i - \sum_{k=i+1}^{m-1} \alpha_{ki}\lambda_k.$$

Finally, we set

$$\lambda_0 = 1 - \sum_{k=1}^{m-1} \lambda_k,$$

and check whether all these values are nonnegative.

How much effort is necessary to determine the values α_{ik}? Here comes the punch line: if we actually represent the M_{B^i} according to (14) and during the push of q_i evaluate the product

$$M_{B^{i-1}}Q_i = \sum_{k=1}^{i-1} \frac{2}{z_k}(Q_k - \bar{Q}_k)(Q_k - \bar{Q}_k)^T Q_i = \sum_{k=1}^{i-1} \alpha_{ik}(Q_k - \bar{Q}_k)$$

according to this expansion, we have already computed α_{ik} by the time we need it for the checking!

Moreover, if we make representation (14) implicit by only storing the z_k and the vectors $Q_k - \bar{Q}_k$, we can even perform the multiplication $M_{B^{i-1}}Q_i$ with $\Theta(di)$ arithmetic operations, compared to $\Theta(d^2)$ operations when we really keep M_B as a matrix or a sum of matrices.

The resulting implementation of the 'push' routine is extremely simple and compact (about 50 lines of code), and it allows the checker to be implemented in 10 more lines of code.

5 Experimental Results

I have tested the algorithm on various point sets: random point sets (to evaluate the speed), vertices of a regular simplex (to determine the dimension limits) and (almost) cospherical points (to check the degeneracy handling). In further rounds, all these examples have been equipped with 'extra degeneracies' obtained by duplicating input points, replacing them by 'clouds' of points very close together, or embedding them into a higher dimensional space. This covers all inputs that have ever been reported as problematic to me. A test suite (distributed with the code) automatically generates all these scenarios from the master point sets and prints out the results.

In most cases, the correct ball is obtained by the pivoting method, while the move-to-front method frequently fails (results range from mildly wrong to wildly wrong on cospherical points, and under input point duplication resp. replacement by clouds). This means, although the move-to-front approach is still slightly faster than pivoting in low dimensions (see the results in the next paragraph), it is highly advisable to use the pivoting approach; it seems to work very well together with the robust update scheme based on the matrix M_B, as described in Section 4. The main drawbacks of the move-to-front method are its dependence on the order of the input points, and its higher number of push operations (the more you push, the more can go wrong). Of course, the input order can be randomly rearranged prior to computation (as originally suggested in [9]), but that eats up the gain in runtime over the pivoting method. On the other hand, if one does not rearrange, it is very easy to come up with bad input orders (try a set of points ordered along a line).

Random point sets. I have tested the algorithm on random point sets up to dimension 30 to evaluate the speed of the method, in particular with respect to the relation between the pivoting and the move-to-front variant. Table 1 (left) shows the respective runtimes for $100,000$ points randomly chosen in the d-dimensional unit cube, in logarithmic scale (averaged over 100 runs). All runtimes (excluding the time for generating and storing the points) have been obtained on a SUN Ultra-Sparc II (248 MHz), compiling with the GNU `C++`-Compiler `g++` Version 2.8.1, and options `-O3 -funroll-loops`. The latter option advises the compiler to perform loop unrolling (and `g++` does this to quite some extent). This is possible because the dimension is fixed at compile time via a template argument. By this mechanism, one also gets rid of dynamic storage management.

As it turns out, the move-to-front method is faster than the pivoting approach up to dimension 8 but then loses dramatically. In dimension 20, pivoting is already more than ten times faster. Both methods are exponential in the dimension, but for applications in low dimensions (e.g. $d = 3$), even $1,000,000$ points can be handled in about two seconds.

Vertices of a simplex. The results for random point sets suggest that dimension 30 is still feasible using the pivoting method. This, however, is not the case for all inputs. In high dimensions, the runtime is basically determined by the calls to the move-to-front method with point set $S \cup \{p^i\}$, S the current support set. We know that $|S| \leq d+1$, but if the input is random, $|S|$ will frequently be smaller (in dimension 20, for example, the average number of support points turns out to be around 17). In this case, a 'pivot step' and therefore the complete algorithm is much faster than in the worst case. To test this worst case, I have chosen as input the vertices of a regular d-simplex in dimension d, spanned by the unit vectors. In this case, the number of support points is d. Table 1 (right) shows the result (move-to-front and pivoting variant behave similarly). Note that to

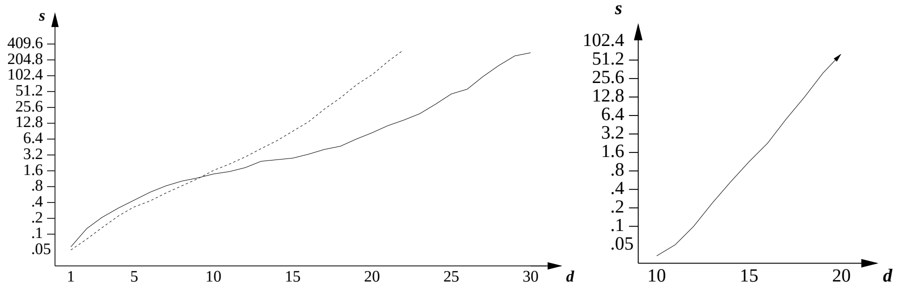

Table 1. Runtime in seconds for 100, 000 random points in dimension d: pivoting (solid line) and move-to-front (dotted line) (left). Runtime in seconds on regular d-simplex in dimension d (right).

solve the problem on 20 points in dimension 20, one needs about as long as for 100, 000 random points in dimension 26!

As a conclusion, the method reaches its limits much earlier than in dimension 30, when it comes to the worst case. In dimension 20, however, you can still expect reasonable performance in any case.

Cospherical points. Here, the master point sets are *exactly* cocircular points in dimension 2, almost cospherical points in higher dimensions (obtained by scaling random vectors to unit length), a tesselation of the unit sphere in 3-space by longitude/latitude values, and vertices of a regular d-cube. While the pivoting method routinely handles most test scenarios, the move-to-front method mainly has problems with duplicated input points and slightly perturbed inputs. It may take very long and computes mildly wrong results in most cases. The slow behavior is induced by many failing push-operations due to the value z being too small, see Section 4. This causes many points which have mistakenly been treated as inside the current ball to reappear outside later.

The most difficult problems for the pivoting method arise from the set of 6144 integer points on the circle around the origin with squared radius $r^2 = 3728702916375125$. The set itself is handled without any rounding errors at all appearing in the result (this is only possible because r^2 still fits into a floating-point value of the `C++` type `double`). However, embedding this point set into 4-space (by adding zeros in the third and fourth coordinate), combined with a random perturbation by a relative amount of about 10^{-30} in each coordinate makes the algorithm fail occasionally. In these case, the computed support set does not have the orgin in its convex hull, which is detected by the checking routine.

6 Conclusion

I have presented a simple, fast and robust code to compute smallest enclosing balls. The program is the last step so far in a chain of improvements and simplifications of the original program written back in 1991. The distinguishing feature is the nice interplay between a new high-level algorithm (the pivoting method) and improved low-level primitives (the M_B-based update scheme).

For dimensions $d \leq 10$, the method is extremely fast, beyond that it slows down a bit, and for $d > 20$ it is not suitable anymore in some cases. This is because every 'pivot step' (a call to the move-to-front method with few points) takes time exponential in d. Even slight improvements here would considerably boost the performance of the whole algorithm. At this point, it is important to note that high dimension is *not* prohibitive for the smallest enclosing ball problem itself, only for the method presented. Interior point methods, or 'real' simplex-type methods in the sense that the pivot step is a polynomial-time operation (see e.g.[2]) might be able to handle very high dimensions in practice, but most likely at the cost of losing the simplicity and stability of the solution I gave here.

Acknowledgment

I would like to thank all the people that have in one way or the other contributed to this code over the last eight years, by using previous versions, making suggestions and reporting bugs. Special thanks go to Emo Welzl for acquainting me with his algorithm eight years ago, and for always keeping an eye on the progress.

References

1. V. Chvátal. *Linear Programming.* W. H. Freeman, New York, NY, 1983.
2. B. Gärtner. Geometric optimization. Lecture Notes for the *Equinoctial School on Geometric Computing*, ETH Zürich, 1997, `http://www.inf.ethz.ch/personal/gaertner/publications.html`
3. Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
4. N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 329–338, 1982.
5. A. L. Peressini, F. E. Sullivan, and J. J. Uhl. *The Mathematics of Nonlinear Programming.* Undergraduate Texts in Mathematics. Springer-Verlag, 1988.
6. William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 2nd edition, 1993.
7. J. Ritter. An efficient bounding sphere. In Andrew S. Glassner, editor, *Graphics Gems.* Academic Press, Boston, MA, 1990.
8. J. J. Sylvester. A question on the geometry of situation. *Quart. J. Math.*, 1:79, 1857.
9. Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.
10. X. Wu. A linear-time simple bounding volume algorithm. In D. Kirk, editor, *Graphics Gems III.* Academic Press, Boston, MA, 1992.