Smallest Enclosing Balls
A Fast and Robust C++ Implementation

Bernd Gärtner

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

Revision: 1.4 , Date: 1999/07/19

This document is available at `http://www.inf.ethz.ch/personal/gaertner/miniball.html`.

Contents

1 Introduction

1.1 About this document

This is an implementation documentation, describing a `C++` code for computing the smallest enclosing ball of an n-point set in d-dimensional Euclidean space. The documentation follows the *literate programming* paradigm [3] and is written in `Anyweb`, Lutz Kettner's front end to the `Funnelweb` system, see `http://www.inf.ethz.ch/personal/kettner/others.html`. The code accompanies my ESA '99 paper *Fast and Robust Smallest Enclosing Balls* [2].

In the next two subsections you find information about the program and how to install it (the trivial process doesn't deserve the word 'installation'), the remaining sections contain the actual documentation and source code.

1.2 The program

The code presented here is small (about 300 lines of code, excluding prototype declarations), fast and robust. It is tuned for dimensions $d \leq 20$ – expect a severe performance dropoff for still higher values of d. Input Points are basically of type `double[d]`, wrapped in a primitive class to allow proper usage in STL lists etc.

No customization is possible, unless you change the code (which you are free to do, of course); for generic code, please use the version of the program contained in the *Computational Geometry Algorithms Library* CGAL, see `http://www.cs.uu.nl/CGAL`.

The program compiles on `g++`, Version 2.8.1 and higher (including recent EGCS compilers), Microsoft Visual C++ and MIPS (Silicon Graphics IRIX). Moreover, it should be easy to adapt to any reasonable platform.

Usage of this code is free of charge for non-commercial purposes. Please contact me if you have further questions, suggestions, bug-reports or comments of any kind.

1.3 Installation

At my web page `http://www.inf.ethz.ch/personal/gaertner/miniball.html`, you will find information on how to download the code. Compile the demo program `miniball_example.C` showing how to use the code, or the test suite `test_suite.C` that performs systematic testing of the code with various point sets. For best performance, choose (under `g++`) options `-O3` and `-funroll-loops`.

2 The Smallest Enclosing Ball Problem

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

For proofs of these facts and further material see [4].

3 Algorithms

Here we describe the two algorithms we implement in the code below, plus the computational primitives needed on the low level.

3.1 Welzl's move-to-front method

The basis of our method is Welzl's *move-to-front* heuristic to compute $\text{MB}(P, B)$ if it exists[4]. 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.

Algorithm 3.1

```
mtf_mb(L_n, B):
    (* returns MB(L_n, B) *)
    mb := MB-bar(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.

3.2 A Pivoting variant

The following improved method 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 'furthest outlier'. Under this scheme, the ball gets large even faster, and the method usually terminates after very few iterations. In small dimensions ($d \leq 10$), the move-to-front method might still be slightly faster, but the pivoting variant has the advantage of being independent of the order in which the points are presented. In contrast, the move-to-front method can suffer from a bad insertion order, both in efficiency and numerical stability. This can be avoided by randomly shuffling the points prior to computation, but then there is no gain in runtime over the pivoting method anymore.

Let $e(p, \text{mb})$ denote the *excess* of p w.r.t. mb, defined as

$$\|p - c\|^2 - r^2,$$

where c is the center and r^2 the squared radius of mb. p lies outside of the current ball if and only if its excess is positive.

Algorithm 3.2

```
pivot_mb(L_n):
    (* returns MB(L_n) *)
    t := 1
    (mb, s) := mtf_mb(L_t, ∅)
    REPEAT
        (* Invariant: mb = MB(L_t) = MB‾(L_s), s ≤ t *)
        choose k > t with e := e(p^k, mb) maximal
        IF e > 0 THEN
            (mb, s') := mtf_mb(L_s, {p^k})
            update L by moving p^k to the front
            t := s + 1
            s := s' + 1
        END
    UNTIL e = 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 i 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 `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.

3.3 Primitive operations

We still need to describe how the computation of $\overline{\text{MB}}(B)$ is realized, which is the only operation involving nontrivial computations.

Recall that during the algorithm, $\overline{\text{MB}}(B)$ is the smallest ball with the set B on its boundary, where B is a set of at most $d + 1$ affinely independent points. In that case, $\overline{\text{MB}}(B)$ is 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 can be computed from 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 of equations can be rewritten as

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

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

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

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

$$C = \sum_{i=1}^{m-1} \lambda_i Q_i, \tag{2}$$

$$r^2 = C^T C. \tag{3}$$

4 Class template `Miniball<d>`: Interface

An object of the class `Miniball<d>` represents a ball of the type MB(P). The construction proceeds by first *checking in* the points of P, then computing their smallest ball using the method `build` (the user may choose between the two algorithms `mtf_mb` and `pivot_mb` described above.

The point set P is maintained in an STL list, supporting the move-to-front operations in constant time. The points themselves are just wrapped arrays, see section 9 below (you may change them via a single `typedef` in the interface below).

During the computations, the set B (which we refer to as the *basis* in the following) is maintained in a stack-like data structure (of type `Basis<d>`) which stores additional information to allow fast solutions of the upcoming systems (1), see Sections 7 and 8 below.

The class `Miniball<d>` has public methods to check in points and to construct the ball for the stored point set. Then we have access functions to retrieve center, squared radius, (number of) points and (number

of) support points of the computed ball. Finally, checking routines can be used to test whether the result is accurate. Their meaning is explained in the implementation section 5 below.

The most important private methods implement the functions `mtf_mb`(L_n, B) and `pivot_mb`(L_n) described above. In case of `mtf_mb`, pre- and postconditions are that `B` represents the current basis B. The parameter i which identifies the list prefix L_i is implemented by an iterator which points to the list entry immediately following the prefix.

In addition, we have a method to perform the move-to-front operation, and a method to compute the maximum excess during an iteration of `pivot_mb`.

Here is the complete class interface.

Miniball interface [1] $\equiv$ {

```
template <int d>
class Miniball {
    public:
        // types
        typedef Wrapped_array<d>                              Point;
        typedef typename std::list<Point>::iterator           It;
        typedef typename std::list<Point>::const_iterator     Cit;

    private:
        // data members
        std::list<Point> L;         // STL list keeping the points
        Basis<d>    B;              // basis keeping the current ball
        It          support_end;    // past-the-end iterator of support set

        // private methods
        void        mtf_mb (It k);
        void        pivot_mb (It k);
        void        move_to_front (It j);
        double      max_excess (It t, It i, It& pivot) const;
        double      abs (double r) const {return (r>0)? r: (-r);}
        double      sqr (double r) const {return r*r;}

    public:
        // construction
        Miniball() {}
        void        check_in (const Point& p);
        void        build (bool pivoting = true);

        // access
        Point       center() const;
        double      squared_radius () const;
        int         nr_points () const;
        Cit         points_begin () const;
        Cit         points_end () const;
        int         nr_support_points () const;
        Cit         support_points_begin () const;
        Cit         support_points_end () const;

        // checking
        double      accuracy (double& slack) const;
```

```
        bool        is_valid (double tolerance = 1e-15) const;
    };

}
```
This macro is invoked in definition 28.

5 Class template `Miniball<d>`: Implementation

5.1 Construction

The `check_in` method uses the STL list's `push_back` method to append the point at the end.

Miniball construction methods [2] $+\equiv$ {
```
    template <int d>
    void Miniball<d>::check_in (const Point& p)
    {
        L.push_back(p);
    }

}
```
This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

The construction method `build` calls by default the function `pivot_mb` to construct the ball of the points previously checked in. The basis B initially corresponds to an empty ball (established by the `Basis` reset method). The user has the choice to compute the ball using the method `mtf_mb` by providing a corresponding flag. As already mentioned, this can be faster in small dimensions (and is therefore an intersting option e.g. in 3d) but there are two drawbacks: (i) it is very easy to construct input sequences where `mtf_mb` is extremely slow (take a set of points ordered along a line, for example). `pivot_mb` on the other hand is basically independent of the point order. (ii) `mtf_mb` can be numerically less stable, e.g. if points occur more than once in the input sequence.

Miniball construction methods [3] $+\equiv$ {
```
    template <int d>
    void Miniball<d>::build (bool pivoting)
    {
        B.reset();
        support_end = L.begin();
        if (pivoting)
            pivot_mb (L.end());
        else
            mtf_mb (L.end());
    }

}
```
This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

The method `mtf_mb` is a quite direct realization of the pseudocode above.

A minor difference is that `B` does not show up as a formal parameter but is manipulated before and after the recursive call by pushing resp. popping the point p_i. As a postcondition, `support_end` is a past-the-end iterator of the support set. Note that the push operation returns a boolean value; normally, this will be `true`, but to safeguard against instabilities in exceptional cases, the push might be ignored, resulting in return value `false`. Below we argue that in such a situation, it causes no harm to ignore the push.

Miniball construction methods [4] $+\equiv$ {

```
template <int d>
void Miniball<d>::mtf_mb (It i)
{
    support_end = L.begin();
    if ((B.size())==d+1) return;
    for (It k=L.begin(); k!=i;) {
        It j=k++;
        if (B.excess(*j) > 0) {
            if (B.push(*j)) {
                mtf_mb (j);
                B.pop();
                move_to_front(j);
            }
        }
    }
}
```

This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

The move-to-front operation has to take care that the support end iterator is advanced first in case it points to the item to be moved.

Miniball construction methods [5] $+\equiv$ {

```
template <int d>
void Miniball<d>::move_to_front (It j)
{
    if (support_end == j)
        support_end++;
    L.splice (L.begin(), L, j);
}


}
```

This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

The `pivot_mb` method is a quite direct implementation of the pseudocode above. The update of s (realized by the `support_end` iterator) happens inside the call to `mtf_mb`. During the main loop, the iterator `t` always denotes the first point not handled after the call to `mtf_mb`. (If `t` is initially the pivot point itself, `t` has to be advanced to guarantee that.) To avoid cycling of the method, we stop as soon as the squared radius of the current ball does not get larger anymore. Further, note that here we do not check whether the push-operation was successful. This is not necessary, because we push onto a basis of size zero, in which case nothing can go wrong (see below).

Miniball construction methods [6] $+\equiv$ {

```
template <int d>
void Miniball<d>::pivot_mb (It i)
{
    It t = ++L.begin();
    mtf_mb (t);
    double max_e, old_sqr_r;
    do {
        It pivot;
        max_e = max_excess (t, i, pivot);
        if (max_e > 0) {
```

```
                t = support_end;
                if (t==pivot) ++t;
                old_sqr_r = B.squared_radius();
                B.push (*pivot);
                mtf_mb (support_end);
                B.pop();
                move_to_front (pivot);
            }
        } while ((max_e > 0) && (B.squared_radius() > old_sqr_r));
    }

}
```
This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

Miniball construction methods [7] $+\equiv$ {
```
    template <int d>
    double Miniball<d>::max_excess (It t, It i, It& pivot) const
    {
        const double *c = B.center(), sqr_r = B.squared_radius();
        double e, max_e = 0;
        for (It k=t; k!=i; ++k) {
            const double *p = (*k).begin();
            e = -sqr_r;
            for (int j=0; j<d; ++j)
                e += sqr(p[j]-c[j]);
            if (e > max_e) {
                max_e = e;
                pivot = k;
            }
        }
        return max_e;
     }

}
```
This macro is defined in definitions 2, 3, 4, 5, 6, and 7.
This macro is invoked in definition 29.

5.2 Access

If the basis's support size is zero, the center will be the origin, and the squared radius equals -1.

Miniball access methods [8] $+\equiv$ {
```
    template <int d>
    typename Miniball<d>::Point Miniball<d>::center () const
    {
        return Point(B.center());
    }

    template <int d>
    double Miniball<d>::squared_radius () const
    {
        return B.squared_radius();
    }
```

```
}
```
This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 29.

The point and support point access is straightforward. In STL style, we return start and (past-the-)end iterators.

Miniball access methods [9] $+\equiv$ {
```
    template <int d>
    int Miniball<d>::nr_points () const
    {
        return L.size();
    }

    template <int d>
    typename Miniball<d>::Cit Miniball<d>::points_begin () const
    {
        return L.begin();
    }

    template <int d>
    typename Miniball<d>::Cit Miniball<d>::points_end () const
    {
        return L.end();
    }

}
```
This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 29.

Miniball access methods [10] $+\equiv$ {
```
    template <int d>
    int Miniball<d>::nr_support_points () const
    {
        return B.support_size();
    }

    template <int d>
    typename Miniball<d>::Cit Miniball<d>::support_points_begin () const
    {
        return L.begin();
    }

    template <int d>
    typename Miniball<d>::Cit Miniball<d>::support_points_end () const
    {
        return support_end;
    }

}
```
This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 29.

5.3 Checking

We have two checking routines. One of them returns numerical values that tell you something about the actual accuracy of the computed ball; if that is not needed, a predicate exists that simply returns whether the ball is ok within a prescribed tolerance.

The accuracy is determined in two steps. First, we check whether all points are inside the ball, and whether the support points are on its boundary. Because this will not absolutely be true, we compute the *absolute point error*, being the maximum absolute excess of any support point or point outside the ball. The return value then is the *relative point error*, given by the absolute point error divided by the squared radius of the ball. Expect a value of less than 10^{-15} in most cases.

In a second step we check 'how far' the computed ball is from being the smallest ball determined by the support points. This is the *slack* of the ball (see Section 8 for the definition of slack). Expect a value of zero in most cases.

Miniball check method [11] $+\equiv$ {

```
template <int d>
double Miniball<d>::accuracy (double& slack) const
{
    double e, max_e = 0;
    int n_supp=0;
    Cit i;
    for (i=L.begin(); i!=support_end; ++i,++n_supp)
        if ((e = abs (B.excess (*i))) > max_e)
            max_e = e;

    // you've found a non-numerical problem if the following ever fails
    assert (n_supp == nr_support_points());

    for (i=support_end; i!=L.end(); ++i)
       if ((e = B.excess (*i)) > max_e)
            max_e = e;

    slack = B.slack();
    return (max_e/squared_radius());
}
```

}

This macro is defined in definitions 11 and 12.
This macro is invoked in definition 29.

A coarser check is implemented by the following routine. It returns `true` if and only if the slack is zero and the accuracy value is below the prescribed tolerance.

Miniball check method [12] $+\equiv$ {

```
template <int d>
bool Miniball<d>::is_valid (double tolerance) const
{
    double slack;
    return ( (accuracy (slack) < tolerance) && (slack == 0) );
}
```

}

This macro is defined in definitions 11 and 12.
This macro is invoked in definition 29.

6 Maintaining the Basis – Theory

6.1 Update

To be efficient, the basis has not only to fulfill the task of solving system (1) by some device, but it also needs to update this device once B changes, and this should be cheaper than recomputing it from scratch. The update is easy when element p^i is removed from the basis (method `pop`) after the recursive call to `mtf_mb`$(L_{i-1}, B \cup \{p^i\})$ – we just need to remember the status prior to the addition of p^i. In the course of this addition (method `push`), however, some real work is necessary.

A possible device for solving system (1) is the explicit inverse A_B^{-1} of the matrix

$$A_B := \begin{pmatrix} 2Q_1^TQ_1 & \cdots & 2Q_1^TQ_{m-1} \\ \vdots & & \vdots \\ 2Q_1^TQ_{m-1} & \cdots & 2Q_{m-1}^TQ_{m-1} \end{pmatrix}$$

along with the vector

$$v_B := \begin{pmatrix} Q_1^TQ_1 \\ \vdots \\ Q_{m-1}^TQ_{m-1} \end{pmatrix}.$$

Having this matrix available, it takes just a matrix-vector multiplication to obtain the values $\lambda_1, \ldots, \lambda_{m-1}$.

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

$$A_{B'} = \left(\begin{array}{ccc|c} & A_B & & \begin{matrix} 2Q_1^TQ_m \\ \vdots \\ 2Q_{m-1}^TQ_m \end{matrix} \\ \hline 2Q_1^TQ_m & \cdots & 2Q_{m-1}^TQ_m & 2Q_m^TQ_m \end{array}\right), \tag{4}$$

and it is not hard to check that equation (4) can be written as follows.

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

where

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

and

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

This implies

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

where

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

Equivalently, we have

$$A_{B'}^{-1} = \left(\begin{array}{c|c} A_B^{-1} + \mu\mu^T/z & -\mu/z \\ \hline -\mu^T/z & 1/z \end{array}\right). \tag{5}$$

As this shows, A_B can easily become ill-conditioned (and the entries of A_B^{-1} unreliable), namely if z evaluates to a very small number. Lemma 6.1 develops a geometric interpretation of z from which we can see that this happens if the new point is very close to the affine hull of the previous ones.

To overcome this problem, we pursue the following alternative update strategy. We will maintain the $(d \times d)$-matrix

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

where

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

stores the points Q_i as columns. Lemma 6.1 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 6.1

(i) With μ as above, 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

$$C' = C + \frac{e}{z}(Q_m - \bar{Q}_m), \tag{6}$$

$$r'^2 = r^2 + \frac{e^2}{2z}, \tag{7}$$

where

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

(v) M_B is updated according to

$$M_{B'} = M_B + \frac{2}{z}(Q_m - \bar{Q}_m)(Q_m - \bar{Q}_m)^T. \tag{8}$$

Property (ii) gives M_B its interpretation as a linear function, namely the projection onto the linear subspace spanned by $Q_1, \ldots, Q_{m-1}$. Further, note that property (v) implies that M_B stays bounded.

For specific inputs, 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 algorithms `mtf_mb` and `pivot_mb` this means that the point to be pushed is treated as if it were inside the current ball. To argue that this is not a crucial mistake, we need to specify what z being 'small' means. The criterion is that we ignore a push operation if and only if the *relative* size of z is small, meaning that

$$\frac{z}{r^2} < \varepsilon$$

for some constant ε, where r^2 is the current squared radius. Here is a heuristic explanation why this works.

Consider a subcall to `mtf_mb`$(L_s, \{p^k\})$ inside the algorithm `pivot_mb` 3.2, 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 '$p^i \notin$ 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 $\mathrm{MB}(L_s, \{p^k\})$. Because of (7), 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 $\overline{\mathrm{MB}}(B \cup \{p^k\})$ 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

$$\frac{e}{r_0^2} \approx \sqrt{2\varepsilon}$$

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

$$\frac{e_0}{r_0^2} \approx \sqrt{2\varepsilon}, \tag{9}$$

where e_0 is the excess of p w.r.t. the final ball mb_0, as desired. As already mentioned, 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 below, we set ε to 10^{-32}, reflecting the observation that the accuracy value of the final ball is even for well-behaved inputs usually larger than 10^{-16}; in that case, the error made in misclassifying points due to failing push operations does not contribute to the overall error in magnitude.

6.2 Checking

The computed ball is the smallest enclosing ball of its support points if and only if its center is in the convex hull of the support points. In the terminology of (1), this is equivalent to $\lambda_1, \ldots, \lambda_{m-1}$ and $\lambda_0 := 1 - \sum_{i=1}^{m-1} \lambda_i$ being nonnegative. Unfortunately, we don't have immediate access to these coefficients. If we had A_B^{-1} available, we could easily get them by a matrix multiplication; from M_B, however, these values cannot be deduced. Instead we proceed as follows: 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 of the linear span will be the set of (pairwise orthogonal) vectors $Q_1 - \bar{Q}_1, \ldots, Q_{m-1} - \bar{Q}_{m-1}$. From the update formula for the center (6) 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 obtained when pushing q_i. This means, the coordinates of C in the alternative basis are the values $(f_1, \ldots, f_{m-1})$.

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 the Lemma above. Here, z_k denotes the value z we got in adding 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

$$M_{B^i} := \sum_{k=1}^{i} \frac{2}{z_k}(Q_k - \bar{Q}_k)(Q_k - \bar{Q}_k)^T \tag{10}$$

we get

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

(after adding q_i to B, Q_i projects to itself). 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 $\alpha_{ii} = 1$.

Here comes the punch line: if we represent M_B according to (10) and evaluate the product $M_B Q_m$ (that we need to compute $\bar{Q}_m$) according to this expansion, we have already computed α_{ik} in the course of adding q_i to the basis! Thus, all we need to do is store these values when they are computed.

The coefficients λ_i in the representation

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

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

$$\lambda_i = f_i - \sum_{k=i+1}^{m-1} \alpha_{ki}\lambda_k. \tag{11}$$

Finally, we set

$$\lambda_0 = 1 - \sum_{k=1}^{m-1} \lambda_k, \tag{12}$$

and check whether all these values are nonnegative. In case no support point is redundant (which should be the case if everything went fine), the coeeficients should even be strictly positive.

7 Class template `Basis<d>`: Interface

In the algorithm, $B = \{q_0, \ldots, q_{m-1}\}$ is an ordered set (elements are ordered by time of addition to B). q_0 is explicitly kept in the basis. Furthermore, for any j in the range $1 \leq j \leq m-1$, we store

- the values z_j and f_j, which are the values of z resp. e/z obtained when pushing q_j.
- the vector $v_j := Q_j - \bar{Q}_j$,
- the values $\alpha_{jk}, 1 \leq k < j$,
- the centers c_j and the squared radii r_j^2, which denote the parameters of the sphere corresponding to $\{q_0, \ldots, q_j\}$. Note that unlike the other values, these ones are also defined and stored for $j = 0$.

The basis has public access methods to retrieve center, squared radius, the size, the number of support points and the excess of a given point.

Methods to manipulate the basis are the method `reset` (to create an empty sphere) as well as the methods `push` and `pop`. `push` may return `false` in case the push operation was rejected due to a too small value of z/r^2, see Section 6.1.

Finally, we have the `slack` method to compute 'how far' the ball is from the optimal one for the given set of support points. The slack is defined to be zero if all the λ_i in the representation of C by the Q_i are nonnegative, and it will be the negative of the smallest λ_i otherwise, see Section 6.2.

Here is the complete interface of the class `Basis`.

Basis interface [13] $\equiv$ {

```
template <int d>
class Basis {
    private:
        // types
        typedef Wrapped_array<d>            Point;

        // data members
        int                 m, s;   // size and number of support points
        double              q0[d];

        double              z[d+1];
        double              f[d+1];
        double              v[d+1][d];
        double              a[d+1][d];

        double              c[d+1][d];
        double              sqr_r[d+1];

        double*             current_c;      // points to some c[j]
        double              current_sqr_r;

        double              sqr (double r) const {return r*r;}

    public:
        Basis();

        // access
        const double*       center() const;
```

```
        double              squared_radius() const;
        int                 size() const;
        int                 support_size() const;
        double              excess (const Point& p) const;

        // modification
        void                reset(); // generates empty sphere with m=s=0
        bool                push (const Point& p);
        void                pop ();

        // checking
        double              slack() const;
    };
```

}
This macro is invoked in definition 28.

8 Class template `Basis<d>`: Implementation

8.1 Access

Basis access methods [14] $+\equiv$ {

```
    template <int d>
    const double* Basis<d>::center () const
    {
        return current_c;
    }

    template <int d>
    double Basis<d>::squared_radius() const
    {
        return current_sqr_r;
    }

    template <int d>
    int Basis<d>::size() const
    {
        return m;
    }

    template <int d>
    int Basis<d>::support_size() const
    {
        return s;
    }

    template <int d>
    double Basis<d>::excess (const Point& p) const
    {
        double e = -current_sqr_r;
        for (int k=0; k<d; ++k)
           e += sqr(p[k]-current_c[k]);
```

```
        return e;
    }

}
```
This macro is defined in definitions 14.
This macro is invoked in definition 29.

8.2 Update

The reset method initializes the basis to be the empty ball (size and number of support points zero). The center is set to zero and the squared radius to -1 in order to make the `excess` method work without case distinction.

Basis modification methods [15] $+\equiv$ {
```
    template <int d>
    void Basis<d>::reset ()
    {
        m = s = 0;
        // we misuse c[0] for the center of the empty sphere
        for (int j=0; j<d; ++j)
            c[0][j]=0;
        current_c = c[0];
        current_sqr_r = -1;
    }

}
```
This macro is defined in definitions 15, 16, 17, and 18.
This macro is invoked in definition 29.

The default constructor calls the reset method.

Basis modification methods [16] $+\equiv$ {
```
    template <int d>
    Basis<d>::Basis ()
    {
        reset();
    }

}
```
This macro is defined in definitions 15, 16, 17, and 18.
This macro is invoked in definition 29.

The `pop` method just decreases the current basis size m. Note that number of support points, the center and the squared radius do *not* change (in particular, they do not revert to their previous values), because the current ball itself is not affected by the pop operation.

Basis modification methods [17] $+\equiv$ {
```
    template <int d>
    void Basis<d>::pop ()
    {
        --m;
    }

}
```

This macro is defined in definitions 15, 16, 17, and 18.
This macro is invoked in definition 29.

In the `push` operation, there are two cases. If $m = 0$, we store q_0 and set the values c_0 and r_0^2 directly; the other values (z, f, v and α) are not defined for index 0. If $m > 0$, we perform the update according to subsection 6.1. The push is ignored if the value z divided by the current squared radius in the update turns out to be too small. As argued above, the value 10^{-32} is a reasonable threshold.

Basis modification methods [18] $+\equiv$ {

```
template <int d>
bool Basis<d>::push (const Point& p)
{
    int i, j;
    double eps = 1e-32;
    if (m==0) {
        for (i=0; i<d; ++i)
            q0[i] = p[i];
        for (i=0; i<d; ++i)
            c[0][i] = q0[i];
        sqr_r[0] = 0;
    } else {
       // set v_m to Q_m
       for (i=0; i<d; ++i)
            v[m][i] = p[i]-q0[i];

       // compute the a_{m,i}, i< m
       for (i=1; i<m; ++i) {
            a[m][i] = 0;
            for (j=0; j<d; ++j)
                a[m][i] += v[i][j] * v[m][j];
            a[m][i]*=(2/z[i]);
       }

       // update v_m to Q_m-\bar{Q}_m
       for (i=1; i<m; ++i) {
            for (j=0; j<d; ++j)
                v[m][j] -= a[m][i]*v[i][j];
       }

       // compute z_m
       z[m]=0;
       for (j=0; j<d; ++j)
            z[m] += sqr(v[m][j]);
       z[m]*=2;

       // reject push if z_m too small
       if (z[m]<eps*current_sqr_r) {
            return false;
       }

       // update c, sqr_r
       double e = -sqr_r[m-1];
       for (i=0; i<d; ++i)
            e += sqr(p[i]-c[m-1][i]);
       f[m]=e/z[m];
```

```
            for (i=0; i<d; ++i)
                 c[m][i] = c[m-1][i]+f[m]*v[m][i];
            sqr_r[m] = sqr_r[m-1] + e*f[m]/2;
        }
        current_c = c[m];
        current_sqr_r = sqr_r[m];
        s = ++m;
        return true;
    }

}
```

This macro is defined in definitions 15, 16, 17, and 18.
This macro is invoked in definition 29.

8.3 Checking

The checking proceeds as described in equations (11) and (12). The return value is either zero (if everything went fine) or the negative of the smallest λ_i.

Basis check method [19] $\equiv$ {

```
    template <int d>
    double Basis<d>::slack () const
    {
        double l[d+1], min_l=0;
        l[0] = 1;
        for (int i=s-1; i>0; --i) {
            l[i] = f[i];
            for (int k=s-1; k>i; --k)
                l[i]-=a[k][i]*l[k];
            if (l[i] < min_l) min_l = l[i];
            l[0] -= l[i];
        }
        if (l[0] < min_l) min_l = l[0];
        return ( (min_l < 0) ? -min_l : 0);
    }

}
```

This macro is invoked in definition 29.

9 Class template `Wrapped_array<d>`

A wrapped array just encapsulates a fixed-size array of `double`. We implement this trivial wrapper class right in the interface. The construction from an array allows the user of the program to work with arrays instead of wrapped arrays—they get automatically converted then.

We also provide a basic output operator.

Wrapped_array interface and implementation [20] $\equiv$ {

```
    template <int d>
    class Wrapped_array {
        private:
```

```
        double coord [d];

    public:
        // default
        Wrapped_array()
        {}

        // copy from Wrapped_array
        Wrapped_array (const Wrapped_array& p)
        {
            for (int i=0; i<d; ++i)
                coord[i] = p.coord[i];
        }

        // copy from double*
        Wrapped_array (const double* p)
        {
            for (int i=0; i<d; ++i)
                coord[i] = p[i];
        }

        // assignment
        Wrapped_array& operator = (const Wrapped_array& p)
        {
            for (int i=0; i<d; ++i)
                coord[i] = p.coord[i];
            return *this;
        }

        // coordinate access
        double& operator [] (int i)
        {
            return coord[i];
        }
        const double& operator [] (int i) const
        {
            return coord[i];
        }
        const double* begin() const
        {
            return coord;
        }
        const double* end() const
        {
            return coord+d;
        }
};

// Output

#ifndef MINIBALL_NO_STD_NAMESPACE
    template <int d>
    std::ostream& operator << (std::ostream& os, const Wrapped_array<d>& p)
    {
```

```
            os << "(";
            for (int i=0; i<d-1; ++i)
                os << p[i] << ", ";
            os << p[d-1] << ")";
            return os;
        }
    #else
        template <int d>
        ostream& operator << (ostream& os, const Wrapped_array<d>& p)
        {
            os << "(";
            for (int i=0; i<d-1; ++i)
                os << p[i] << ", ";
            os << p[d-1] << ")";
            return os;
        }
    #endif

}
```
This macro is invoked in definition 30.

10 Usage example

Here is a program that shows how to use the code. It generates a set of random points and checks them in, then constructs their ball and outputs the parameters of this ball.

miniball usage example [21] $\equiv$ {
```
    int main ()
    {
    #ifndef MINIBALL_NO_STD_NAMESPACE
            using std::cout;
            using std::endl;
    #endif

            const int       d = 5;
            const int       n = 100000;
            Miniball<d>     mb;

            // generate random points and check them in
            // ----------------------------------------
            Miniball<d>::Point p;
            random_seed (1999);
            for (int i=0; i<n; ++i) {
                for (int j=0; j<d; ++j)
                    p[j] = random_double();
                mb.check_in(p);
            }

            // construct ball, using the pivoting method
            // -----------------------------------------
            cout << "Constructing miniball..."; cout.flush();
            mb.build();
```

```
        cout << "done." << endl << endl;

        // output center and squared radius
        // --------------------------------
        cout << "Center:         " << mb.center() << endl;
        cout << "Squared radius: " << mb.squared_radius() << endl << endl;

        // output number of support points
        // --------------------------------
        cout << mb.nr_support_points() << " support points: " << endl << endl;

        // output support points
        // ---------------------
        Miniball<d>::Cit it;
        for (it=mb.support_points_begin(); it!=mb.support_points_end(); ++it)
            cout << *it << endl;
        cout << endl;

        // output accuracy
        // ---------------
        double slack;
        cout << "Relative accuracy: " << mb.accuracy (slack) << endl;
        cout << "Optimality slack:  " << slack << endl;

        return 0;
    }

}
```
This macro is invoked in definition 31.

11 Test suite

We set up a test suite to check the code on a predefined collection of point sets read from files. These are (almost) cospherical points, random points, points forming regular simplices and cubes. These point sets are tested as they come, and in various modifications, obtained by

- adding several copies of each point,
- embedding the points into a higher dimensional space
- randomly perturbing the coordinates by different relative amounts.

All these modifications are realized by choosing appropriate parameters in the following basic test routine. The parameters of the classes `Dim<...>` and `Dim2<...>` enable the compiler to deduce the template parameters.

The test suite currently encounters problems only for one test set, namely the cocircular points with small radius. Here, embedding the points into 4-space and slightly perturbing the coordinates leads in some cases to the wrong result. Try running the test suite with different random seeds to see the effect. All other examples work nicely, although sometimes the accuracy value is as low as 10^{-8}, where 10^{-16} is the usual order of magnitude. However, you can still consider this as being ok.

classes Dim and Dim2 [22] ≡ {

```
    template <int d>
        class Dim {};
    template <int d, int D>
        class Dim2 {};
```

}

This macro is invoked in definition 32.

basic test for P [23] ≡ {

```
    template <int d, int D, class Point> // embedding from d-space into D-space
    void basic_test (const std::vector<Point>& P,
                     Dim2<d,D> dim,      // enables the compiler to deduce d,D
                     double amount = 0,  // absolute amount of perturbation
                     int copies = 1)     // number of copies of each point
    {
    #ifndef MINIBALL_NO_STD_NAMESPACE
            using std::cout;
    #endif

            Miniball<D> mb;
            typedef typename Miniball<D>::Point PointD;
            PointD p;
            for (int i=0; i<P.size(); ++i) {
                for (int k=0; k<copies; ++k) {

                    // first d coordinates are taken from P
                    int j;
                    for (j=0; j<d; ++j)
                        p[j] = P[i][j] + (random_double()-0.5)*amount;

                    // last D-d coordinates are assumed to be zero
                    for (j=d; j<D; ++j)
                        p[j] = (random_double()-0.5)*amount;
                    mb.check_in(p);
                }
            }

            // construction
            mb.build();

            // validity check
            double slack, accuracy = mb.accuracy(slack);
            char s;
            if (slack > 0)
                s = '#';
            else {
                if (accuracy > 1e-15) s = '*'; else s = ' ';
            }
            cout << s << "(" << accuracy << ", " << slack << ")" << s << " ";
        }
```

}

This macro is invoked in definition 32.

The actual test routine for P calls the basic test several times, for P itself, for P as a multiset with 10 copies of each point, and for P embedded in $2d$-space. In each case, we test five different amounts of perturbation. For this, we need a parameter `magnitude` specifying the order of magnitude of the input coordinates (assuming this magnitude is about the same for all coordinates). All these tests are performed three times.

test set for P [24] $\equiv$ {

```
template <int d, int D, class Point>
void test_set (const std::vector<Point>& P,
               Dim2<d,D> dim, double magnitude)
{
#ifndef MINIBALL_NO_STD_NAMESPACE
        using std::cout;
        using std::endl;
#endif

        double    tiny =         magnitude*(1e-50),
                  footnotesize = magnitude*(1e-30),
                  small =        magnitude*(1e-10),
                  large =        magnitude*(1e-3);

        for (int tries=0; tries<3; ++tries) {
            cout << " Try: " << tries << endl;

            cout << "  Set (d):      ";
            basic_test (P, Dim2<d,d>(), 0);
            basic_test (P, Dim2<d,d>(), tiny);
            basic_test (P, Dim2<d,d>(), footnotesize);
            basic_test (P, Dim2<d,d>(), small);
            basic_test (P, Dim2<d,d>(), large);
            cout << endl;

            cout << "  Multiset (d): ";
            basic_test (P, Dim2<d,d>(), 0, 10);
            basic_test (P, Dim2<d,d>(), tiny, 10);
            basic_test (P, Dim2<d,d>(), footnotesize, 10);
            basic_test (P, Dim2<d,d>(), small, 10);
            basic_test (P, Dim2<d,d>(), large, 10);
            cout << endl;

            cout << "  Set (D):      ";
            basic_test (P, Dim2<d,D>(), 0);
            basic_test (P, Dim2<d,D>(), tiny);
            basic_test (P, Dim2<d,D>(), footnotesize);
            basic_test (P, Dim2<d,D>(), small);
            basic_test (P, Dim2<d,D>(), large);
            cout << endl;

            cout << "  Multiset (D): ";
            basic_test (P, Dim2<d,D>(), 0, 10);
            basic_test (P, Dim2<d,D>(), tiny, 10);
            basic_test (P, Dim2<d,D>(), footnotesize, 10);
            basic_test (P, Dim2<d,D>(), small, 10);
            basic_test (P, Dim2<d,D>(), large, 10);
```

```
            cout << endl;
        }
        cout << endl;
    }

}
```
This macro is invoked in definition 32.

The complete test reads a set P from a file and calls the above test set for it.

test for set P obtained from a file [25] $\equiv$ {
```
    template <int d>
    void test (char* file, Dim<d> dim, double magnitude)
    {
    #ifndef MINIBALL_NO_STD_NAMESPACE
            using std::cout;
            using std::endl;
            using std::ifstream;
    #endif

            typedef typename Miniball<d>::Point Point;

            Point                   p;
            std::vector<Point>      P;

            cout << file << "...";
            ifstream from (file);
            int n; from >> n;
            for (int i=0; i<n; ++i) {
                for (int j=0; j<d; ++j)
                    from >> p[j];
                P.push_back(p);
            }
            cout << endl;

            test_set (P, Dim2<d, 2*d>(), magnitude);
        }

}
```
This macro is invoked in definition 32.

Finally, the testsuite calls the main test routine for several files. Every file used here has a comment section at the end where a description of the point set it contains is given.

test suite [26] $\equiv$ {
```
    int main (int argc, char* argv[])
    {
    #ifndef MINIBALL_NO_STD_NAMESPACE
            using std::cout;
            using std::endl;
            // using std::atoi; commented out because Visual C++ doesn't like it
    #endif
            if (argc != 2) {
                cout << "Usage: test_suite <seed>" << endl;
                exit(1);
```

```
    } else
        random_seed (atoi(argv[1]));

    cout << "Miniball testsuite" << endl
         << "------------------" << endl
         << "Note: Results in asterisks *(...)* are usually of lower"<< endl
         << "      accuracy, while results in hash marks #(...)# may"<< endl
         << "      really be wrong. Please consult the documentation"<< endl
         << "      for details." << endl
         << "-----------------------------------------------------"<< endl
         << "Running testsuite (this may take a while)..." << endl << endl;

    cout << "columns: input perturbations 0 | 1e-50 | 1e-30 | 1e-10 | 1e-3"
         << endl << endl;

    test ("cocircular_points_small_radius_2.data", Dim<2>(), 1e8);
    test ("cocircular_points_large_radius_2.data", Dim<2>(), 1e9);

    test ("almost_cospherical_points_3.data", Dim<3>(),1);
    test ("almost_cospherical_points_10.data", Dim<10>(),1);

    test ("longitude_latitude_model_3.data", Dim<3>(),1);

    test ("random_points_3.data", Dim<3>(),1);
    test ("random_points_5.data", Dim<5>(),1);
    test ("random_points_10.data", Dim<10>(),1);

    test ("simplex_10.data", Dim<10>(),1);
    test ("simplex_15.data", Dim<15>(),1);

    test ("cube_10.data", Dim<10>(),1);
    test ("cube_12.data", Dim<12>(),1);

    return 0;

}
```

}

This macro is invoked in definition 32.

References

[1] V. Chvátal. *Linear Programming*. W. H. Freeman, New York, NY, 1983.

[2] B. Gärtner. Fast and robust smallest enclosing balls. In *Proc. ESA'99*, volume ??? of *Lecture Notes in Computer Science*, page ???, 1999.

[3] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[4] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

12 Files

12.1 miniball_config.H

We start with a configuration file that defines some compiler dependent macros and functions. Currently, Visual C++ and EGCS support standard namespaces. For these compilers, all standard identifiers get qualified over using directives; for the other ones, the using directives are ignored. We also define functions that return random doubles, depending on the compiler.

miniball_config.H [27] ≡ {

version and license [33]

```
#ifndef MINIBALL_CONFIG_H
    #define MINIBALL_CONFIG_H
    #if defined(__sgi) && !defined(__GNUC__)              // assume MIPS-IRIX
        #define MINIBALL_NO_STD_NAMESPACE
    #endif

    #if defined(__GNUC__) && (__GNUC_MINOR__<=90)         // assume old GNU
        #define MINIBALL_NO_STD_NAMESPACE
    #endif

    #if !defined(__sgi) && !defined(__GNUC__)             // assume Visual C++
        #include<cstdlib>
        inline void random_seed (unsigned int seed) {srand(seed);}
        inline double random_double () {return double(rand())/RAND_MAX;}
    #else                                                 // no Visual C++
        #ifndef MINIBALL_NO_STD_NAMESPACE
            #include<cstdlib>
            inline void random_seed (unsigned int seed){std::srand48(seed);}
            inline double random_double () {return std::drand48();}
        #else
            #include<stdlib.h>
            inline void random_seed (unsigned int seed){srand48(seed);}
            inline double random_double () {return drand48();}
        #endif
    #endif
#endif
```

}

This macro is attached to an output file.

12.2 miniball.H

This file contains the interfaces of the two classes `Miniball<d>` and `Basis<d>`. We do not assume automatic template code inclusion, so the file `miniball.C` is included in the end.

miniball.H [28] ≡ {

version and license [33]

```
#ifdef MINIBALL_NO_STD_NAMESPACE
    #include <list.h>
#else
    #include <list>
#endif
```

```
#include "wrapped_array.H"

template <int d> class Miniball;
template <int d> class Basis;

// Miniball
// --------
```
Miniball interface [1]

```
// Basis
// -----
```
Basis interface [13]

```
#include "miniball.C"
```

}
This macro is attached to an output file.

12.3 miniball.C

`miniball.C` just contains the implementations of the template classes `Miniball<d>` and `Basis<d>`.

miniball.C [29] ≡ {
version and license [33]

```
#ifdef MINIBALL_NO_STD_NAMESPACE
   #include<assert.h>
#else
   #include<cassert>
#endif

   // Miniball
   // --------
```
Miniball construction methods [2]
Miniball access methods [8]
Miniball check method [11]

```
   // Basis
   // -----
```
Basis access methods [14]
Basis modification methods [15]
Basis check method [19]

}
This macro is attached to an output file.

12.4 wrapped_array.H

The file `wrapped_array.H` contains the complete primitive point class `Wrapped_array<d>`.

wrapped_array.H [30] ≡ {
version and license [33]

```
   #include "miniball_config.H"

#ifdef MINIBALL_NO_STD_NAMESPACE
   #include <iostream.h>
#else
   #include <iostream>
#endif
```

Wrapped_array interface and implementation [20]

}
This macro is attached to an output file.

12.5 miniball_example.C

Then, we have the example file.

miniball_example.C [31] ≡ {
version and license [33]

```
   #include "miniball_config.H"
#ifdef MINIBALL_NO_STD_NAMESPACE
   #include <stdlib.h>
   #include <iostream.h>
#else
   #include <cstdlib>
   #include <iostream>
#endif

   #include "miniball.H"
```

miniball usage example [21]

}
This macro is attached to an output file.

12.6 miniball_test_suite.C

Finally, there is the test suite.

miniball_test_suite.C [32] ≡ {
version and license [33]

```
   #include "miniball_config.H"

#ifdef MINIBALL_NO_STD_NAMESPACE
   #include <stdlib.h>
   #include <math.h>
   #include <iostream.h>
   #include <fstream.h>
   #include <vector.h>
#else
   #include <cstdlib>
   #include <cmath>
   #include <iostream>
```

```
   #include <fstream>
   #include <vector>
#endif

   #include "miniball.H"

   classes Dim and Dim2 [22]
   basic test for P [23]
   test set for P [24]
   test for set P obtained from a file [25]
   test suite [26]

}
```
This macro is attached to an output file.

12.7 License and Version text

Here, we define macros that include the GPL header and the version number of the code into all files that are generated.

version and license [33] $\mathbf{M} \equiv$ {
```
    //    Copright (C) 1999
    //    $Revision: 1.4 $
    //    $Date: 1999/07/19 14:09:21 $
    //
    //    This program is free software; you can redistribute it and/or modify
    //    it under the terms of the GNU General Public License as published by
    //    the Free Software Foundation; either version 2 of the License, or
    //    (at your option) any later version.
    //
    //    This program is distributed in the hope that it will be useful,
    //    but WITHOUT ANY WARRANTY; without even the implied warranty of
    //    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    //    GNU General Public License for more details.
    //
    //    You should have received a copy of the GNU General Public License
    //    along with this program; if not, write to the Free Software
    //    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA,
    //    or download the License terms from prep.ai.mit.edu/pub/gnu/COPYING-2.0.
    //
    //    Contact:
    //    --------
    //    Bernd Gaertner
    //    Institut f. Informatik
    //    ETH Zuerich
    //    ETH-Zentrum
    //    CH-8092 Zuerich, Switzerland
    //    http://www.inf.ethz.ch/personal/gaertner
    //

}
```
This macro is invoked in definitions 27, 28, 29, 30, 31, and 32.