Exact arithmetic at low cost
a case study in linear programming

Exact Arithmetic at Low Cost – a Case Study in Linear Programming *

Bernd Gärtner†

Abstract

We describe a new exact-arithmetic approach to linear programming when the number of variables n is much larger than the number of constraints m (or vice versa). The algorithm is an implementation of the simplex method which combines exact (multiple precision) arithmetic with inexact (floating point) arithmetic, where the number of exact arithmetic operations is small and usually bounded by a function of $\min(n, m)$. Combining this with a "partial pricing" scheme (based on a result by Clarkson [9]) which is particularly tuned for the problems under consideration, we obtain a correct and practically efficient algorithm that even competes with the inexact state-of-the-art solver CPLEX[1] for small values of $\min(n, m)$ and and is far superior to methods that use exact arithmetic in any operation. The main applications lie in computational geometry.

1 Introduction

Linear Programming (LP) – the problem of maximizing a linear objective function in n variables subject to m linear (in)equality constraints – is the most prominent optimization problem, and efficient methods have been devised to solve such problems in practice. The values of n and m for which solutions can nowadays be computed, range up to several millions for $\max(n, m)$ and several thousands for $\min(n, m)$. The *simplex method*, invented by G. Dantzig 50 years ago [10], is still among the most practical methods to solve linear programs, and state-of-the art solvers like CPLEX implement variants of it.

While the simplex method in its theoretical description smoothly works for any values of n and m, the typical values of these parameters encountered in an application greatly influence the way it is implemented best.

The scenario in which we work here is that $\min(n, m)$ is a small constant, at most 30, say, while $\max(n, m)$ can get very large. This is not the scenario usually encountered in operations research. The *NETLIB* collection, a popular set of benchmark LPs (see http://www.netlib.org/lp/data/) only features a few problems with $\min(n, m) < 100$, most of them also having $\max(n, m)$ relatively small.

However, in computational geometry (CG), our scenario is more common, and the applications we present below come from this area. Usually, problems arising in CG have to do with large point sets (or sets of other simple objects) in small-dimensional space and lead to small values of $\min(n, m)$ whenever they can be formulated as linear (or more general) optimization problems. Two examples (which don't look like LP at first glance) illustrate this.

*supported by the ESPRIT IV LTR project No. 21957 (CGAL) and by the Swiss Science Foundation (SNF), project No. 21-50647.97. A preliminary version of this paper appears in the Proceedings of the 9th Annual ACM Symposium on Discrete Algorithms (SODA) 1998, San Francisco

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

[1]trademark of CPLEX Optimization Inc.

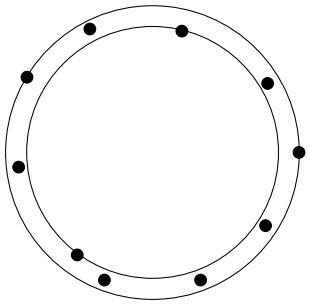

Figure 1: Smallest enclosing annulus

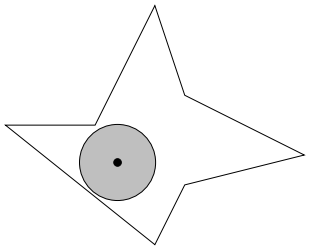

Figure 2: Largest disk in kernel

Problem 1.1 (Smallest enclosing annulus)
Given n points in the plane, find the annulus of smallest area covering all the points (see Figure 1).

This annulus is a device for testing 'roundness' of the point set. The problem can be formulated as LP in 4 variables and $2n$ constraints. A suitable generalization to dimension d leads to an LP in $d+2$ variables and $2n$ constraints – we come back to this problem in the last section where we use it as our major test problem.

Problem 1.2 (Largest disk in kernel)
Given a simple n-vertex polygon in the plane, find the largest disk in its kernel*, that is the region of the polygon from which all vertices are visible (see Figure 2).*

This problem is LP with 3 variables and n constraints. By solving it, one can in particular test the polygon for being starshaped, and if so, obtain a 'central' point of its kernel.

A major issue any serious LP solver must deal with is numerical accuracy, for two reasons. First, the program must not crash due to numerical problems, and second, the computed result should be correct. While the first reason is indisputable, the second one needs further consideration. Namely, what correctness means, as well as what the best way is to achieve correctness, depends on the application.

If, for example, the input values are already approximations of true values obtained by measurements, it may suffice if the output satisfies certain tolerances. On the other hand, if the objective is to test whether a certain point lies inside a given 0-1-polytope (a problem which can be formulated as LP), a wrong answer to this decision problem might have disastrous effects in an ambient algorithm, or lead to theoretical 'insights' which are none. This also applies

if the LP under consideration is a relaxation of a more difficult integer linear program (ILP), supposed to yield an upper bound for the optimal solution of the latter. Here, solution values of 16.998 and 17.001 make a tremendous difference. In general, if combinatorial rather than just numerical information has to be extracted from the input, there is a need for exact computations. Good examples are vertex- or facet-enumeration algorithms – they typically offer exact arithmetic [2, 7, 14].

Also, in choosing the means of achieving correct results, properties of the input are important. For example, if we know the problem to be nondegenerate, with intermediate solutions well separated (as it is typically the case in randomly generated problems), numerical stability becomes much less an issue.

The point is that a general purpose LP solver *must* be able to handle any problem but *may* take advantage of inputs it typically expects to be confronted with. It seems that existing solvers either ignore the 'may' part (we call their strategy *'expecting the worst'*) or neglect the 'must' part (in which case they follow the paradigm *'hoping for the best'*).

Expecting the worst. This strategy avoids numerical errors altogether by performing all computations in rational arithmetic over an exact multiprecision number type. If both n and m are small, this is certainly the method of choice, and for $\max(n, m)$ not too large, one can still obtain solutions in reasonable time (see our tests below). The strategy is implemented, for example, in the LP solvers that are part of the vertex-enumeration codes [2, 14] mentioned above.

However, the approach is in no case competitive with solvers like CPLEX – it is too pessimistic in the sense that floating point operations are assumed to go wrong all the time, where in practice, they work fine most of the time.

Hoping for the best. This strategy – used e.g. by CPLEX – tries do do as well as possible, purely with floating point arithmetic. Although this is fast and will in most cases compute the correct optimal solution, it will fail on some problems. Checking the result with exact arithmetic is possible, of course, but really helps only if the result is actually correct. Otherwise, a postoptimization phase has to be started (which is not an obvious task if the computed solution is neither primal nor dual feasible).

The approach is too optimistic in the sense that all problems are assumed to be well-behaved (with respect to the numerical techniques that are applied), where in practice, only most of them are.

Summarizing, *expecting the worst* is always correct but also always slow, while *hoping for the best* is always fast and usually correct. For our particular scenario, we propose a mixed strategy, combining floating point and exact arithmetic, which will always be correct and usually fast.

Our solver accepts floating point input, and most arithmetic operations are done in fast floating point arithmetic. Assuming $m < n$, in most cases only $O(m^2)$ out of the $\Theta(nm)$ arithmetic operations performed in a single iteration of the simplex method, need to be done exactly. Finding a pivot is done completely in floating point, verifying (or rejecting) it requires the additional evaluation of a simple semi-static error bound (and $O(m)$ exact computations if the bound does not suffice); performing the actual update step takes $O(m^2)$ exact operations. Since the number of iterations required to solve an LP by the simplex method usually depends only on m, the overhead we get for exact arithmetic is then just an additive constant depending on m. As our tests show, this constant is very reasonable for small values of m.

In 'bad' cases, many pivots might get rejected under exact arithmetic, before a suitable candidate is found, and a single iteration might require $\Theta(nm)$ exact operations. In such situations, however, pure floating point implementations are more likely to fail, either (our tests below demonstrate this).

The idea of combining floating point with exact arithmetic is not new, and floating point filters have successfully been applied in computational geometry before [13]. Although the motivation is similar (avoid exact arithmetic whenever floating point computations suffice), our approach is different in two respects.

Classical filter techniques apply *interval arithmetic*, i.e. they maintain for each floating point value computed during the algorithm an interval which is guaranteed to contain it. In any arithmetic operation, the result's interval is computed from the intervals of the operands. Exact comparisons of two values can then be done fast if their intervals do not overlap.

First of all, this approach already leads to a constant-factor slowdown, even if the problem is of the most well-behaved kind (Näher and Mehlhorn report a floating point filter operation to be about four times slower than the underlying floating point operation [17]). Our approach incurs only an additive overhead in this case.

Second, interval arithmetic works only for expressions of low arithmetic degree. Already in the process of solving an $m \times m$ linear system (a routine similar to this appears in any simplex implementation), the error bounds obtained from interval arithmetic are typically a gross overestimate, making them too large to be useful, even for small values of m. However, Brönnimann *et al.* [5] have recently described a technique to obtain good error bounds for the solution of a linear system, using a posteriori error analysis of an approximate (floating point) inverse of the matrix defining the linear system. The intuition is that if this inverse is a good approximation of the real inverse (as one expects in most practical cases), the error in the solution will be small. In this situation, the semi-static error bounds necessary to verify the chosen pivot will still work well under this small additional error. Empirical evidence for the efficiency of this scheme does not yet exist.

The rest of the paper is organized as follows. In Section 2 we give a brief description of the simplex method and sketch the main ideas of our implementation. Section 3 contains the more technical part describing details of the implementation. In Section 4 we give test results.

2 Linear Programs and the Simplex Method

We consider problems of maximizing a linear function in n variables subject to m linear equality constraints, where the variables must assume nonnegative values. Such problems can be written as

$$\begin{array}{lll} \text{(LP)} & \text{maximize} & c^T x \\ & \text{subject to} & Ax = b, \\ & & x \geq 0, \end{array} \qquad (1)$$

where c is an n-vectors, A an $m \times n$-matrix, b an m-vector, and x an n-vector of variables.

If a vector $x^* = (x_1^*, \ldots, x_n^*)$ exists that satisfies all the constraints, the problem is called *feasible*, otherwise *infeasible*. In the former case x^* is a *feasible solution* (FS). If the *objective function* $z = c^T x$ is bounded from above on the set of feasible solutions x^*, the problem is called *bounded*, otherwise *unbounded*.

The restriction to equality constraints is no loss of generality, because *slack variables* can be introduced to turn inequalities into equalities. Moreover, the simplex method (and our code) can

handle explicit bounds $l_j \le x_j \le u_j$ on the variables. However, for ease of exposition, we restrict attention to the *standard form* as given by (1).

Tableaus and basic feasible solutions. For an ordered subset $J = \{j_1, \ldots, j_k\} \subseteq [n]$, let x_J denote the k-vector $(x_{j_1}, \ldots, x_{j_k})$. If $n \ge m$, a tableau for (1) is a set of equations

$$\begin{array}{rcl} x_B &=& \beta - \Lambda x_N \\ \hline z &=& z_0 + \gamma^T x_N, \end{array} \tag{2}$$

$B, N \subseteq [n], |B| = m, |N| = n - m, B \cup N = [n]$, Λ an $m \times (n-m)$-matrix, γ an $(n-m)$-vector, β an m-vector, z_0 a number, such that the equations above the solid line are equivalent to the system $Ax = b$, expressing the m-vector x_B of *basic variables* in terms of the $(n-m)$-vector of *nonbasic variables* x_N. The last row stores the objective function value z as a (linear) function of the nonbasic variables. A feasible solution x^* arises from the tableau by assigning nonnegative values to the nonbasic variables x_N in such a way that the implied values of x_B are nonnegative as well. x^* is a *basic feasible solution* (BFS) if any nonbasic variable assumes the value 0.

If the matrix A of (1) does not have full (row) rank, there is no tableau for (1), so even if the problem is feasible, it need not have a BFS. However, if it has a BFS at all, then it must have a BFS which is an optimal FS for (1), unless the problem is unbounded.

Let A_B resp. A_N collect the columns of A corresponding to the basic resp. nonbasic variables (c_B and c_N are defined similarly). Then the tableau is uniquely determined via

$$\begin{array}{rcl} \beta &=& A_B^{-1} b, \\ \Lambda &=& A_B^{-1} A_N, \\ z_0 &=& c_B^T A_B^{-1} b, \\ \gamma^T &=& c_N^T - c_B^T A_B^{-1} A_N. \end{array} \tag{3}$$

We refer to A_B as the *basis matrix*. Λ is the *tableau matrix*, γ the vector of *reduced costs*, B the *basis* and N the *nonbasis*.

A BFS is therefore uniquely specified by the basis B. In particular, there are only finitely many BFS. For proofs and further details we refer to Chvátal's book [8].

The Simplex Method.

The simplex method consists of two phases, commonly called *phase I* and *phase II*. Phase I takes as input the initial LP (1) and either reports that the problem is infeasible, or generates an equivalent problem (with $n \ge m$), along with a tableau and a BFS x^* associated with it. This is done by solving an auxiliary problem for which an initial tableau is easily constructed [8].

Given a tableau and associated BFS x^*, an iteration of phase II either asserts that x^* is optimal for (1), reports that the problem is unbounded, or constructs a new tableau with a corresponding BFS y^* satisfying $c^T y^* \ge c^T x^*$. In the latter case, the process is repeated. Since there are only finitely many BFS, phase II finally terminates, unless a sequence of tableaus repeats itself forever. This phenomenon – known as cycling – can occur but there are techniques to avoid it [8]. In any case, it can happen that although the basis changes, the BFS x^* remains the same for several iterations. Such iterations are called *degenerate*.

Contrary to the *standard* simplex method, the *revised* simplex method does not explicitly maintain the tableau (2) but retrieves all necessary information from the implicit representation given by (3), the key quantity being the basis matrix A_B.

An iteration, also known as a *pivot step* consists of three main parts, the *pricing*, the *ratio test* and the *update*. In the following we assume $n \geq m$.

Pricing. From the tablau (2), we can immediately deduce that if the reduced cost vector γ satisfies $\gamma \leq 0$, then the associated BFS x^* is optimal. Namely, $x_N \geq 0$ then implies $z = z_0 + \gamma^T x_N \leq z_0$, where z_0 is the objective function value associated with x^*. The pricing step evaluates the vector γ and eiter certifies $\gamma \leq 0$ or delivers an index j with $\gamma_j > 0$. According to (3), the computational primitives in the pricing step are the following.

(i) computation of $v^T := c_B^T A_B^{-1}$,

(ii) evaluation of reduced cost values $c_j - v^T A_j$, $j \in N$.

Assuming that A_B^{-1} is available, (i) can be done in time $O(m^2)$, while (ii) costs $O(nm)$, if all γ_j are examined. In case of $n \gg m$, (ii) dominates the runtime of the pricing step and should therefore be done fast. Our strategy will be to compute v^T in exact arithmetic but evaluate the inner products $v^T A_j$ in floating point arithmetic, using a floating point approximation $\tilde{v}$ of v. However, we make sure that the index j that is finally returned truly satisfies $\gamma_j > 0$. In this case, x_j is called the *entering variable*.

Another important ingredient of our pricing scheme will be *partial pricing*, where we only search for the index j among a small subset $S \subseteq N$. Only if no candidate is found among S, we enlarge S according to a certain rule. The larger the ratio n/m is, the more effective partial pricing becomes. Section 3 contains the details.

Ratio Test. Given a variable x_j with $\gamma_j > 0$, it is clear from (2) that increasing its value by $t > 0$ increases the objective function value by $\gamma_j t$ and gives rise to a solution $x^*(t)$, where $x_j^*(t) = t$ and

$$x_B^*(t) = x_B^* - t\Lambda_j,$$

Λ_j the tableau column corresponding to the variable x_j. The ratio test consists of finding the largest value of t such that $x^*(t)$ is still feasible, equivalently, $x_B^*(t) \geq 0$. If no such value exists, the problem is obviously unbounded. Otherwise, let t_0 denote this maximum value. Then there is $i \in B$ such that $x_i^*(t_0) = 0$. x_i is called the *leaving variable*. Solving the i-th tableau equation for x_j and substituting the resulting expression for x_j into the other tableau equations, we obtain a new tableau in which x_j is basic and x_i nonbasic. If $t_0 > 0$, the objective function value has increased, otherwise we have performed a degenerate iteration. The computational primitives in the ratio test are

(i) computation of the *pivot colum* $\Lambda_j = A_B^{-1} A_j$,

(ii) solution of linear equations $x_i^*(t) = 0$, for all $i \in B$.

The cost of the ratio test only depends on m, and we completely do it in exact arithmetic. As in the pricing, (i) can be done in $O(m^2)$ time if A_B^{-1} is available, (ii) in $O(m)$.

Update. In the revised simplex method, we do not explicitly perform the tableau update mentioned in the ratio test. Rather, we just replace B with $B' := B \cup \{j\} \setminus \{i\}$, and by (3), this would suffice to have all necessary information available in the next iteration. However, it is crucial that the computations of $c_B^T A_B^{-1}$ and $A_B^{-1} A_j$ are done efficiently, and this requires some

preprocessing to bring A_B into a suitable format. Many formats are possible, but the chosen one should at least be easy to update when A_B changes to $A_{B'}$, the update being substantially cheaper than preprocessing $A_{B'}$ from scratch. As already suggested above, we explicitly keep the exact inverse of A_B^{-1}. As we show in Section 3, this *basis inverse* is easily updated in time $O(m^2)$.

In standard inexact solvers dealing with large sparse problems, one rather stores some factorization of A_B, because if A_B is sparse, there is a chance that also a sparse factorization is found, even if A_B^{-1} is dense. Moreover, this factorization usually has better numerical properties than the inverse. Because we deal with relatively small values of m, sparsity is not an issue (but consider the remark on this in the conclusion), and because we compute exactly, numerical stability need not be taken into account, either. But then the simplicity of the update routine is in favor of the explicit inverse.

3 An Exact Implementation of the Simplex Method

In the previous section we have roughly indicated which computations are to be done exactly and which ones in floating point arithmetic. Before we go any further, let us specify the requirements for the exact number type T, assuming that the floating point arithmetic is done in a floating point type F.

First of all, because our solver accepts floating point input, T must contain F as a subset (where overflows and exceptional values need not be considered). Apart from that, T is a subset of the real numbers and must form a ring, so addition, subtraction and multiplication must be defined. Moreover, we assume division to be available for operands whose quotient is an element of T again. In Section 4 we present a type that fulfills these requirements.

Now we can describe our concrete realizations of the *pricing* and *update* step (the *ratio test* is straightforward and works over T, as described above). Let us start with the representation format for A_B^{-1} and its update if column i of A_B is replaced with a new column A_j.

Maintaining the Basis Inverse.

If $M = A_B$ contains entries from T, Cramer's rule states that the entries of M^{-1} are rational numbers over T, with common denominator $\det(M)$, so that M^{-1} can also be stored as a matrix over T, keeping the denominator separately.

Now assume that the i-th column of M is replaced by a new column A_j, the resulting matrix being $\tilde{M} = A_{B'}$. Defining $\lambda := M^{-1}A_j$, we have

$$\tilde{M} = M \begin{pmatrix} 1 & & \lambda_1 & & \\ & \ddots & \vdots & & \\ & & \lambda_i & & \\ & & \vdots & \ddots & \\ & & \lambda_m & & 1 \end{pmatrix}, \tag{4}$$

where λ_k can be written as $\lambda_k = \lambda'_k / \det(M)$, λ'_k of type T, for $k = 1 \ldots m$. It follows that

$$\tilde{M}^{-1} = \frac{1}{\lambda'_i} \begin{pmatrix} \lambda'_i & & -\lambda'_1 & & \\ & \ddots & \vdots & & \\ & & \det(M) & & \\ & & \vdots & \ddots & \\ & & -\lambda'_m & & \lambda'_i \end{pmatrix} M^{-1},$$

so the entries of $\tilde{M}^{-1}$ can be written as rational numbers over T with common denominator $\lambda'_i \det(M)$, and performing the matrix multiplication gives the corresponding numerators $a_{k\ell}$,

$$\tilde{M}^{-1}_{k\ell} = \frac{a_{k\ell}}{\lambda'_i \det(M)}.$$

Note that the values λ'_k are readily available, because $\lambda = M^{-1}A_j$ is nothing else than the pivot column Λ_j already computed in the ratio test prior to this update (see previous section).

On the other hand, we know that the entries of $\tilde{M}^{-1}$ have a rational representation

$$\tilde{M}^{-1}_{k\ell} = \frac{b_{k\ell}}{\det(\tilde{M})},$$

which is the one we are actually interested in. From (4) we get $\det(\tilde{M}) = \det(M)\lambda_i = \lambda'_i$. Consequently,

$$b_{k\ell} = \frac{a_{k\ell} \det(\tilde{M})}{\lambda'_i \det(M)} = \frac{a_{k\ell}}{\det(M)},$$

and the divsion must be without remainder over T.

The whole update step can be performed in time $O(m^2)$ (for technical reasons, we always keep the absolute value $|\det(A_B)|$ as the denominator). This technique of updating the basis inverse has been discovered before by J. Edmonds and termed 'Q-pivoting' [11]. It has been implemented, for example, by D. Avis in his vertex enumeration algorithm `lrs` [2].

Pricing.

In the process of finding the entering variable, i.e. an index j with reduced cost $\gamma_j > 0$, we almost exclusively apply floating point arithmetic. Pricing is the most flexible step in the simplex method, because typically many indices j qualify, in which case we are free to choose. The actual choice is then made according to a *pivot rule*. We start (without referring to arithmetic) by specifying the rule we use, called *partial reduced cost pricing*. This rule accesses and manipulates a global subset $S \subseteq N$ of *active* indices, initially chosen to be relatively small (see below).

For an ordered index set $I \subseteq N$, define $\max(I)$ as the first index $j \in I$ with $\gamma_j = \max\{\gamma_k, k \in I\}$.

Algorithm 3.1 *(partial reduced cost pricing)*

(* returns j with $\gamma_j > 0$ or `optimal` *)
$j := \max(S)$
`IF` $\gamma_j > 0$ `THEN`
    `RETURN` j
`ELSE`

```
    V := {k ∈ N \ S | γ_k > 0}
    IF V = ∅ THEN
       RETURN optimal
    ELSE
       S := S ∪ V
       RETURN max(V)
    END
END
```

If subsequently the basis B is updated to $B' = B \cup \{j\} \setminus \{i\}$, S is set to $S \cup \{i\} \setminus \{j\}$ in the next iteration.

The choice of $\max(I)$ (resp. $\max(V)$) as return index is *Dantzig's rule*. The idea is that variables with large reduced cost values will lead to a fast increase in objective function value when chosen as entering variable.

The intuition behind the partial pricing scheme is that S and V are always small and that S is augmented only a few times. In this case, most iterations are cheap, because they operate on a small index set, while only a few ones run through the whole nonbasis N to find the set V. Exactly this intuition lies also behind Clarkson's LP algorithm [9] that works in a dual setting (few variables, many constraints) and can easily be formulated as a dual simplex method. The interpretation of Clarkson's algorithm as a dual partial pricing scheme has already been suggested by Adler and Shamir [1]. A related technique known in operations research is 'column generation' which is typically used to solve large problems that do not fit into main memory. Applying a technique quite similar to partial reduced cost pricing, Bixby et al. have been able to handle very large-scale LP [4].

To prove the above intuition rigorously, we need an assumption about the LP that is unfortunately not always satisfied. While in theory, we would then abandon the rule, in practice, we keep applying it and find that it still works well. The purpose of the theoretical result in this case is not to supply a proof of efficiency in a worst-case scenario but to give an idea how to reach efficiency in practice.

Lemma 3.2 *(Clarkson [9])*
If the LP is nondegenerate, the following holds in Algorithm 3.1.

(i) If S is a random subset of N of size r, then the expected size of V is at most $m(n-r)/(r+1)$.

(ii) If $V \neq \emptyset$, V contains at least one element of any optimal basis B.

(ii) implies that S is augmented at most m times, while (i) yields an estimate of $|V|$, at least when we enter the pricing step for the first time with a random subset S. For example, if we choose $|S| = m\sqrt{n}$, the expected size of V will be no more than $\sqrt{n}$. One can even prove that V is that small in the subsequent augmentation steps [15], so that in total, no more than $2m\sqrt{n}$ indices are ever expected to appear in S. In our implemetation, we use $m\sqrt{n}$ as the intial size of S.

Arithmetic. The steps in Algorithm 3.1 that require arithmetic are the computations of $\max(S)$ (and $\max(V)$ if necessary). Thus, we need to compute the index j with largest value

$$\gamma_j = c_j - v^T A_j, v^T = c_B A_B^{-1}.$$

Recall that we store the rational entries of A_B^{-1} only as their numerators and keep the denominator $D := |\det(A_B)|$ separately. In this situation, the vector $w^T = Dc_B A_B^{-1}$ has entries in T, and we are better off considering

$$\gamma'_k := D\gamma_k = Dc_k - w^T A_k, k \in S, \tag{5}$$

which is an expression over T. Now, instead of evaluating (5) exactly, we compute floating point approximations

$$\tilde{\gamma}'_k = \tilde{D} \otimes c_k - \tilde{w}^T \odot A_k, k \in S, \tag{6}$$

where $\tilde{D}, \tilde{w}$ are floating point approximations of D and w that are computed once in the beginning of the pricing step, $\otimes$ and $\odot$ the floating point multiplication resp. dot product. The obvious candidate for $\max(S)$ is then the index j with largest value $\tilde{\gamma}'_j$. (In case D or an entry of w is larger than the largest representable floating point number, both D and w are scaled by a suitable power of two before computing the approximation.)

Note that for the correctness of the method, it does not matter whether $j = \max(S)$ really holds, the important property is that γ_j is positive. This, however, can be checked with exact arithmetic at cost $O(m)$. The benefit of *Dantzig's rule* in this context is that γ_j is actually very likely to be positive, because it has been found to be largest under floating point computations.

If the check succeeds, j is returned. Otherwise, we just proceed with the computation of V. Note that we might have missed a candidate $j \in S$ with $\gamma_j > 0$ due to inexact computations, but we ignore that in the hope of finding another one later in $N \setminus S$.

Only if also from $N \setminus S$ (which is handled as S before), we cannot retrieve any candidate in this way, we need to do more work (just declaring the whole problem as optimally solved, would be a blunder, of course).

The straightforward solution would be to recompute all reduced costs again in exact arithmetic to check whether a candidate for the entering variable has been missed. This, however, is not necessary in most cases. We know that all inexact values $\tilde{\gamma}'_k, k \in N$ are nonpositive at this stage, and a candidate has been missed if and only if some exact value γ'_k is still positive. The following lemma develops an error bound on $\tilde{\gamma}'_k$ that also lets us deduce $\gamma'_k \leq 0$ if $\tilde{\gamma}'_k$ is sufficiently far below zero. Only the indices k which can not be decided using the error bound, need to be handed over to exact arithmetic. In typical cases, these are very few.

Lemma 3.3 *Let*

$$\begin{aligned} C_k &:= \max(|c_k|, \max_{i=1}^{m} |(A_k)_i|), k \in [n], \\ R_i &:= \max_{k=1}^{n} |(A_k)_i|, i \in [m], \\ R_0 &:= \max_{k=1}^{n} |c_k|. \end{aligned}$$

(The C_k and R_i are the column and row maxima of the LP). Furthermore, define

$$U := \max\left(\tilde{D} \otimes R_0, \max_{i=1}^{m}(|\tilde{w}_i| \otimes R_i)\right)$$

and

$$W := \max\left(\tilde{D}, \max_{i=1}^{m} |\tilde{w}_i|\right).$$

If the floating point type F has p bits of precision, then

$$|\tilde{\gamma}'_k - \gamma'_k| \leq \min(U \otimes q, W \otimes q \otimes C_k), k \in N,$$

where $q = (1 + 1/64)(m + 1)(m + 2)2^{-p}$.

It is important to note that this bound is an expression which can exactly be evaluated in floating point arithmetic, so that no errors occur in computing the error bound. The bound is usually very good, because q is quite small. A typical value for p is 53, as in the `C++` type `double`.

Proof. We first show the following general estimate. Let x, y be vectors of length ℓ, $\mu := 2^{-p}$. Then

$$|\tilde{x} \odot y - xy| \leq (1 + 1/64)\ell(\ell + 1)\mu \max_{1 \leq i \leq \ell} |\tilde{x}_i \otimes y_i|. \tag{7}$$

We derive this from classical results of Forsythe and Moler [12], stating that

$$\begin{aligned} \tilde{x}_i &= x_i(1 + \delta_i), \\ x_i &= \tilde{x}_i(1 + \epsilon_i), \end{aligned}$$

$|\delta_i|, |\epsilon_i| \leq \mu, i = 1 \ldots \ell$, and if $\ell\mu < 0.01$,

$$\tilde{x} \odot y = \sum_{i=1}^{\ell} \tilde{x}_i y_i \left(1 + 1.01\ell\Theta_i\mu\right), \ |\Theta_i| \leq 1, i = 1 \ldots \ell.$$

Hence we get

$$\tilde{x} \odot y = \sum_{i=1}^{\ell} x_i y_i \left(1 + \delta_i\right)\left(1 + 1.01\ell\Theta_i\mu\right).$$

This further gives

$$\begin{aligned} |\tilde{x} \odot y - xy| &= |\sum_{i=1}^{\ell} x_i y_i (1.01\ell\Theta_i\mu + \delta_i + 1.01\ell\delta_i\Theta_i\mu)| \\ &= |\sum_{i=1}^{\ell} \tilde{x}_i y_i (1 + \epsilon_i)(1.01\ell\Theta_i\mu + \delta_i + 1.01\ell\delta_i\Theta_i\mu)| \\ &\leq |\sum_{i=1}^{\ell} \tilde{x}_i y_i|(1 + \mu)(1.01\ell\mu + \mu + 1.01\ell\mu^2) \\ &\leq \ell \max_i |\tilde{x}_i y_i|(1 + \mu)(1.01\ell\mu + \mu + 1.01\ell\mu^2). \end{aligned}$$

We want to transform this bound into a bound that depends on computable values. To this end we note that [12]

$$\tilde{x}_i y_i = \tilde{x}_i \otimes y_i(1 + \eta_i), |\eta_i| \leq \mu, i = 1 \ldots \ell.$$

Then we can further estimate

$$\begin{aligned} |\tilde{x} \odot y - xy| &\leq \max_i |\tilde{x}_i \otimes y_i|(1 + \mu)^2(1.01\ell^2\mu + \ell\mu + 1.01\ell^2\mu^2) \\ &\leq \max_i |\tilde{x}_i \otimes y_i| 1.01\ell(\ell + 1)\mu, \end{aligned}$$

for any practical value of ℓ, if $\mu = 2^{-53}$. Finally, we would like to majorize the constant 1.01 by a constant that is exactly evaluated over the floating point numbers. To this end we observe that

$$1.01 = (1 + 1/64)(1 - 9/1625),$$

and the estimate (7) follows. There are two different ways to upper bound the maximum, and this finally implies the lemma when we put $x^T := (D, -w^T), y^T := (c_k, A_k^T)$. Namely, on the one hand we have

$$\max_i |\tilde{x}_i \otimes y_i| \leq \max_i (|\tilde{x}_i| \otimes \max_y |y_i|),$$

y running over all vectors we consider during the pricing. On the other hand,

$$\max_i |\tilde{x}_i \otimes y_i| \leq \max_i |\tilde{x}_i| \otimes \max_i |y_i|.$$

□

To apply the bound of the Lemma, we first check whether $\tilde{\gamma}'_k$ is separated from zero by more than $U \otimes q$. This bound is independent from k and can therefore be computed once in the beginning of the pricing step. If this bound does not suffice to tell the sign of γ'_k, we apply the second bound, which involves one extra multiplication for each k. If after applying both bounds, the sign of γ'_k is still undecided, we resort to exact arithmetic.

With this scheme, we either find still one more index j satisfying $\gamma_j > 0$ (and return it), or we certify that $V = \emptyset$ in Algorithm 3.1, as claimed by the inexact computations. In this case, returning the value `optimal` is correct.

4 Test Results

We have tested our implementation on various instances of Problem 1.1 (smallest enclosing annulus) and its generalization to higher dimensions. We have compared its performance to that of the primal simplex solver of CPLEX 4.0.9. and to a version of our code using exact arithmetic in any operation. We also have done tests on the three NETLIB problems with highest variable-to-constraint ratio.

The program was written in `C++`, compiled with GNU's `g++`, Version 2.7.2.1 (optimization level `-O3`) and run on a SUN Ultra 1. The floating point type F is `double`. In a first version, we have used the number type `bigfloat` from LEDA [18] as the exact number type T. A `bigfloat` stores numbers of the form $s \cdot 2^e$, where s is a multiprecision LEDA `integer` and e an integer exponent. However, since `bigfloat` is a true superset of `double`, also able to handle overflows etc., arithmetic operations have quite some overhead we do not want to spend. Moreover, `bigfloat` does not offer a division operator as we need it. Therefore, we have developed our own version of the type, providing just the required functionality. (We have also experimented with GNU multiprecision integers (see http://cpw.math.columbia.edu/online/gmp_toc.html) instead of LEDA integers but found the latter to be faster in our context.)

Before we give the results, let us formally introduce the d-dimensional version of Problem 1.1. This is the problem of finding the annulus (region between two concentric spheres) with minimal difference between the squared radii, that covers an n-point set $P = \{p_1, \ldots, p_n\}$ in d-space (for $d = 2$, this is equivalent to minimizing the area).

If $\underline{r}, \overline{r}$ denote the small and large radius of the annulus, c the annulus's center, then the objective is to minimize $\overline{r}^2 - \underline{r}^2$ subject to the constraints

$$\underline{r} \leq \|p_i - c\| \leq \overline{r}, i = 1 \ldots n.$$

d	partial	full	exact	CPLEX
2	2.2 s	3.2 s	17.4 s	5.4 s
5	2.6 s	6.3 s	29.6 s	6.5 s
10	6.0 s	20.9 s	1:53 min	9.2 s
15	25.2 s	47.4 s	5:41 min	13.6 s
20	72.0 s	2:12 min	13:30 min	20.4 s

Table 1: Runtimes on random annulus problems, $n = 50,000$

If $p_i = (x_1^i, \ldots, x_d^i)$, $c = (c_1, \ldots, c_d)$ we can equivalently write this as

$$\underline{r}^2 \leq (x_1^i - c_1)^2 + \cdots + (x_d^i - c_d)^2 \leq \overline{r}^2, i = 1 \ldots n.$$

Defining

$$\alpha := \underline{r}^2 - c_1^2 - \cdots - c_d^2, , \quad \beta := \overline{r}^2 - c_1^2 - \cdots - c_d^2,$$

we can easily formulate the problem as an LP with $2n$ constraints and $d+2$ variables $\alpha, \beta, c_1, \ldots c_d$. Its dual can be written as a problem in $2n$ variables $\lambda := (\lambda_1, \ldots, \lambda_n), \mu := (\mu_1, \ldots, \mu_n)$ and $d+2$ constraints, as follows.

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^n ((x_1^i)^2 + \cdots + (x_d^i)^2)\mu_i \\ & -\sum_{i=1}^n ((x_1^i)^2 + \cdots + (x_d^i)^2)\lambda_i, \\ \text{subject to} & \sum_{i=1}^n \lambda_i = 1, \\ & \sum_{i=1}^n -\mu_i = -1, \\ & \sum_{i=1}^n 2x_1^i\lambda_i - \sum_{i=1}^n 2x_1^i\mu_i = 0, \\ & \vdots \\ & \sum_{i=1}^n 2x_d^i\lambda_i - \sum_{i=1}^n 2x_d^i\mu_i = 0, \\ & \lambda, \mu \geq 0. \end{array} \tag{8}$$

This problem immediately fits into the form of (1), and from an optimal solution for it, an optimal solution to the primal problem and thus an optimal annulus can easily be reconstructed.

The first set of test problems consists of smallest enclosing annulus problems over $n = 50,000$ randomly generated 24-bit integer points in dimensions $2, 5, 10, 15$ and 20 (the machine reached its memory limit at $nm \approx 1,000,000$). Table 1 gives the runtimes obtained by our method, with partial pricing and *full* pricing (initial value of $S = N$, see Section 3), compared with our method (partial pricing version) using exact arithmetic for any operation ('exact'), and CPLEX.

In any case, the correct result was obtained by all solvers. As was to be expected for random input, no candidate delivered from the pricing was ever rejected by the exact check of our method, and the error bounds were in all cases sufficient to verify optimality in the final iteration.

It is remarkable that we still win against CPLEX in dimension 10, because of our partial pricing scheme. In dimensions still higher, the exact arithmetic starts to contribute considerably (although the contribution is still tolerable). The full exact version is not as slow as one might have expected. One reason is that, during the pricing, any exact multiplication has one operand of small bitlength, namely an input number. In the basis inverse update, this is not the case.

Table 2 depicts the number of simplex iterations taken by the different methods on the same problems. Although partial pricing takes more iterations than full pricing, it is faster because the individual iterations are cheaper ('exact' behaves like 'partial' here).

d	partial	full	CPLEX
2	18	11	11
5	40	20	14
10	82	61	35
15	185	98	83
20	242	184	163

Table 2: Number of iterations on random annulus problems, $n = 50,000$

partial	full	CPLEX
20.4 s	29.6 s	116.9 s

Table 3: Runtimes on random annulus problem, $d = 2, n = 500,000$

Here it is interesting to note that the number of CPLEX iterations is much smaller than even the number of full pricing iterations in our method. This is due to the fact that *Dantzig's rule* is usually inferior to other pivot rules like *DEVEX* or *steepest descent* as applied by CPLEX. We have used *Dantzig's rule* mainly because it nicely works together with our inexact pricing scheme. In the future, other rules will be tested as well.

Table 3 shows how our code performs for $d = 2, n = 500,000$ (because of memory limitations, this was the largest n we could test). CPLEX apparantly has problems with this input and is outperformed by a factor of almost six.(The full exact version is not of interest for us in this and the subsequent tests.)

In connection with the random problems, we observed an interesting phenomenon. When generating random annulus problems with `double` entries from the random number generator `drand48`, the runtime of our method is spectacularly good, even for large dimensions. The mystery is solved when one observes that the numerators and the denominator $|\det(A_B)|$ stored in the basis inverse A_B^{-1}, have very small encoding lengths in the format $s2^e$ in this case, so that exact arithmetic is fairly cheap. The reason for this is that matrices generated with `drand48` (or any other random number generator based on the linear congruential algorithm), exhibit extreme dependencies among the rows. In generating the random problems tested above, care was taken that at least dependency patterns that lead to unusually small numbers were avoided.

It is also quite clear that the runtime of our method depends on the bitlengths of the input point coordinates. Above, we have tested with 24-bit coordinates in order to make sure that problem (8) can be set up in `double` format without rounding errors up to dimension $d = 20$. If 48-bit coordinates are used (leading to an inexact internal representation of (8)), the runtimes are about the same up to dimension 10, 50% more in dimension 15 and about twice as large in dimension 20.

To test degenerate inputs, special 2-dimensional annulus problems were generated that have all points exactly on a circle. A first such example features $6,144$ integer points on a common circle with squared radius $3,728,702,916,375,125$, a value that can still be represented in `double`, so (8) is set up without rounding errors. The first row of Table 4 gives the results for partial pricing and CPLEX (full pricing makes basically no difference here and is not tested).

While our method needs 5 iterations (four in phase I and one in phase II, just to check optimality), CPLEX gets away with no iteration at all (it still does something, as the time

	partial	CPLEX
original	0.8 s	1.54 s
perturbed	0.29 s	3.18 s

Table 4: Runtimes on small degenerate annulus problem, $d = 2, n = 6,144$, original and perturbed

partial	CPLEX
1.95 s	3.14 s

Table 5: Runtimes on large degenerate annulus problem (rounded), $d = 2, n = 13,824$

suggests). Because all reduced cost values are zero in the final iteration, this time all of them were processed with exact arithmetic by our method. Both solvers came up with the correct optimal value 0.

The next test problem was generated by slightly perturbing the points so that they are no longer cocircular. To do this, a random value in $\{-1, 1\}$ was added to each coordinate. The results are given in the second row of Table 4.

Although our method now takes 24 iterations, it has become faster, because the small perturbation already suffices to make the error bounds work effectively: no reduced cost value was passed to exact arithmetic in the final iteration. CPLEX becomes slower and computes a solution that deviates from the correct one in the 9-th significant digit.

The final annulus problem features 13,824 integer points on a common circle with squared radius $1,128,305,502,495,112,825$, a value that no longer fits into a `double`. It follows that in setting up the LP (8), rounding errors are made. However, considering the rounded problem as the correct input, we ran the solvers on it. Table 5 collects the results.

Our code takes 13 iterations, but needs to check all reduced costs excatly in the final iteration (although they are not zero, no single error bound suffices). The optimal solution is 128, CPLEX comes up with the value 408 (note that this is not a large deviation, considering the size of the input numbers).

Finally, we have tested our solver on standard benchmark problems from the NETLIB collection, choosing the three problems which are most suitable for our method in the sense that they have relatively few constraints and relatively many variables. Table 6 gives the statistics on these problem, Table 7 the runtimes we achieve, again distinguished between partial and full pricing (in case of SCSD1, partial and full pricing coincide). CPLEX was run for comparison.

It is clear that we cannot compete with CPLEX on these NETLIB problems, although they are probably the ones from the collection on which we still do best. Note that for FIT1D, partial pricing brings about no benefit. If the number of variables is larger (as in FIT2D), we again profit

Problem	n	m
FIT1D	1026	24
FIT2D	10500	25
SCSD1	760	77

Table 6: Statistics on NETLIB problems

Problem	partial	full	CPLEX
FIT1D	96 s	98 s	0.83 s
FIT2D	19:59 min	24:12 min	13.6 s
SCSD1	42:13 min	42:13 min	0.07 s

Table 7: Runtimes on NETLIB problems

Problem	partial	full	CPLEX
FIT1D	1734	1482	1137
FIT2D	15549	13478	14168
SCSD1	499	499	201

Table 8: Number of iterations on NETLIB problems

(a little) from our pricing scheme. Compared to the annulus problems before, a large number of iterations is required, indicating that the problems here are more difficult (see Table 8). SCSD1 is the worst example for our code but the best example for CPLEX. There are several reasons why our code is much worse than CPLEX on the NETLIB problems, the most important one being that these are sparse problems (unlike the annulus problems before). In this case CPLEX profits from its very efficient sparsity handling, while we completely ignore sparsity. In case of SCSD1, it is also the overhead for exact arithmetic (recall that $m = 77$ here) that becomes overwhelming.

We have included this statistics mainly in order not to create the impression that our method is superior to CPLEX. It is particularly tuned for a class of problems (dense, few constraints), for which CPLEX is not tuned, and our method makes full sense only on such problems.

5 Conclusion

We have described a *correct* implementation of the simplex method with low overhead for exact computations, if the number of constraints or variables is small. The algorithm can in some cases compete with CPLEX and beats the full exact solver by far. The applications mostly lie in computational geometry, and it is planned to incorporate the solver into the *Computational Geometry Algorithms Library* CGAL, a joint project of seven European sites (see http://www.cs.ruu.nl/CGAL/).

A more tuned implementation of exact arithmetic would be a natural next step. The scheme based on interval arithmetic that was already mentioned in the introduction might lead to another substantial speedup [5]. Moreover, Brönnimann *et al.* [6] have shown that *modular arithmetic* can yield much faster computation of determinants; the technique might also apply to the matrix operations considered here. However, under such techniques, the simplicity of the current implementation will probably not persist.

Another issue has already been addressed above. In explicitly maintaining the basis inverse, we ignore sparsity of the matrix A_B. This might seem justified if we store the LP in dense format anyway. However, we can profit from sparsity effects because of the exaxt arithmetic. Namely, exact operations involving zero values are cheap (and neglegible in comparison with other operations), so that even without an explicit sparse format, we implicitly handle the matrix operations as if the matrix was in sparse representation. It follows that it would pay off to abandon

the format of the explicit inverse in favor of a factorization that respects sparsity of the input.

Finally, we would like to mention that the techniques introduced here are not restricted to linear programming but can be applied to a variety of other optimization problems. In computational geometry, important examples are the smallest enclosing ball of a point set, or the distance between two polytopes. These are in particular quadratic programming problems, for which a simplex-type solution method exists [19]. More general, the abstract class of *LP-type problems* as introduced by Sharir and Welzl [16] can – slightly modified – be handled by our approach of combining exact and floating point computations. To illustrate the point, we briefly recall the basics.

An LP-type problem is a pair (H, w), H a set of "constraints", w an "objective function" assigning to each subset $G \subseteq H$ a value $w(G)$ (the smallest solution subject to the subset of constraints in G), with the following properties.

- $w(F) \leq w(G)$ for $F \subseteq G$ (*monotonicity*)
- if $F \subseteq G$ with $w(F) = w(G)$ and $w(G) < w(G \cup \{h\})$ for some $h \in H \setminus G$, then also $w(F) < w(F \cup \{h\})$ (*locality*)

A basis of $G \subseteq H$ is an inclusion-minimal subset B of G such that $w(B) = w(G)$. The *combinatorial dimension* of (H, w) is the maximum cardinality of any basis. To solve the problem means to find $w(H)$ and a basis of H. As an example, consider the problem of finding the smallest enclosing ball (SMB) of a set of points $P \in \mathbb{R}^d$. Because this ball is determined by at most $d+1$ of the input points, the combinatorial dimension of SMB is $d + 1$.

Sharir and Welzl's randomized algorithm to solve LP-type problems needs two problem-specific primitives.

- *violation test*: for a basis B and $h \in H \setminus B$, decide whether $w(B) < w(B \cup \{h\})$ holds.
- *basis computation*: if $w(B) < w(B \cup \{h\})$, compute a basis B' of $B \cup \{h\}$.

When we interpret the simplex method in terms of these primitives, the reduced cost computations performed during the pricing step play the role of violation tests, while the ratio test and update step together form the basis computation. The crucial point is that the reduced cost computation in the simplex method is more than just a violation test, namely it computes an *amount* of violation. Exactly the availability of such a *quantitative* violation test makes our whole method work. In all known practical applications of the LP-type framework, one actually has such a stronger violation test, where the amount of violation is (like in the simplex method) not a fixed quantity but a value usually determined by some sensible heuristic. Summarizing, the methods of this paper apply in principle to general LP-type problems for which the following primitive exists.

- *quantitative violation test*: for a basis B and $h \in H \setminus B$, return an amount $a(B, h) \in \mathbb{R}$ of violation which is positive exactly if $w(B) < w(B \cup \{h\})$.

What remains in a concrete application is to develop problem-specific error bounds which can decide the sign of $a(B, h)$ from a floating point approximation of it in most cases.

Acknowledgment

Parts of the code come from a previous implementation of an inexact solver developed together with Sven Schönherr, who in particular wrote the routine for reading (NETLIB) problems in MPS-format. Torsten Thiele provided the 'points-on-a-circle' examples. Komei Fukuda's suggestions substantially improved the presentation, and his exact LP solver – part of the vertex/facet enumeration algorithm `cddr+` – helped to verify the correctness of our code. Finally, many discussions with Emo Welzl have contributed to this paper.

References

[1] I. Adler and R. Shamir. A randomized scheme for speeding up algorithms for linear and convex programming with high constraints-to-variable ratio. *Math. Programming*, 61:39–52, 1993.

[2] D. Avis. A `C` implementation of the reverse search vertex enumeration algorithm. URL: ftp:// mutt.cs.mcgill.ca/pub/C/.

[3] D. Avis, K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discr. Comput. Geom.*, 8:295–313, 1992.

[4] R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno. Very large-scale linear programming: a case study in combining interior point and simplex methods. *Operations Research*, 40(5):885–897, 1992.

[5] H. Brönnimann, C. Burnikel, S. Pion. Interval arithmetic yields efficient arithmetic filters for computational geometry. Manuscript, 1997.

[6] H. Brönnimann, I. Z. Emiris, V. Y. Pan, S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[7] A. Löbel, T. Christof. Porta – polyhedron representation transformation algorithm. URL: http://www.zib.de/ Optimization/Software/Porta/.

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

[9] K. L. Clarkson. A Las Vegas algorithm for linear programming when the dimen sion is small. *J. ACM*, 42(2):488–499, 1995.

[10] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[11] J. Edmonds and J.-F. Maurras. Note sur les Q-matrices d'Edmonds. *Recherche Opérationnelle (RAIRO)*, 31(2):203–209, 1997.

[12] G. Forsythe and C. Moler. *Computer Solutions of Linear Algebar Systems*. Prentice Hall, 1967.

[13] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[14] K. Fukuda. `cdd+` reference manual. URL: http://www.ifor.math.ethz.ch/ifor/staff/fukuda/cddplus_man/cddman.html.

[15] B. Gärtner and E. Welzl. An analysis of Clarkson's sampling lemma. Mansuscript, 1997.

[16] J. Matoušek, M. Sharir and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

[17] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Fachber. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1989.

[18] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[19] P. Wolfe. The simplex method for quadratic programming. *Econometrica*, 27:382–398, 1959.