# Combining Precision Boosting with LP Iterative Refinement for Exact Linear Optimization*

Leon Eifler[1] Jules Nicolas-Thouvenin Ambros Gleixner[1,2]

[1]Zuse Institute Berlin
[2]HTW Berlin

[3] November 13, 2023

**Abstract**

This article studies a combination of the two state-of-the-art algorithms for the exact solution of linear programs (LPs) over the rational numbers, i.e., without any roundoff errors or numerical tolerances. By integrating the method of *precision boosting* inside an *LP iterative refinement* loop, the combined algorithm is able to leverage the strengths of both methods: the speed of LP iterative refinement, in particular in the majority of cases when a double-precision floating-point solver is able to compute approximate solutions with small errors, and the robustness of precision boosting whenever extended levels of precision become necessary. We compare the practical performance of the resulting algorithm with both pure methods on a large set of LPs and mixed-integer programs (MIPs). The results show that the combined algorithm solves more instances than a pure LP iterative refinement approach, while being faster than pure precision boosting. When embedded in an exact branch-and-cut framework for MIPs, the combined algorithm is able to reduce the number of failed calls to the exact LP solver to zero, while maintaining the speed of the pure LP iterative refinement approach.

## 1 Introduction

Linear programming (LP) is a fundamental optimization technique widely used in various fields, including operations research, engineering, economics, and finance. In practice, linear programming solvers rely on fast floating-point arithmetic, coupled with the careful use of error tolerances to efficiently compute accurate solutions. However, the use of floating-point arithmetic can lead to numerical inaccuracies, especially for problems with large coefficient ranges, which in turn can result in inaccurate solutions or incorrect claims of optimality or infeasibility. Exact linear programming algorithms aim to solve LPs exactly over the rational numbers, i.e., without any numerical inaccuracies or error tolerances. Such exact solvers are needed as a subroutine for exact mixed integer programming (MIP) [6, 8], but can also directly be used to investigate numerically challenging LPs or to establish theoretical results [17, 22, 4, 5, 10, 19, 21, 25].

The naïve approach of performing a simplex method in exact arithmetic was observed to be prohibitively slow in many practical applications by Espinoza [11]. Also the idea of using limited-precision arithmetic at a fixed, but sufficiently high level in order to obtain theoretical guarantees of convergence to an exact solution is limited in its

*The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (BMBF grant numbers 05M14ZAM, 05M20ZBM).

practical applicability. This holds even for algorithms with polynomial runtime, such as the algorithm described by Grötschel, Lovász, and Schrijver [16], which is itself based on the ellipsoid method of Khachiyan [20], see [13]. In this article, we focus purely on simplex-based methods. Their warm-starting capabilities also align well with one of over main motivations to us exact LP solvers as a subroutine in exact branch-and-bound solvers.

Among simplex-based methods for solving LPs exactly, the more successful approaches rely on combining floating-point arithmetic and exact arithmetic in some way. An early attempt was presented by Dhiflaoui et al. [7]. They first solve the LP approximately in double-precision arithmetic and try to prove the optimality of the found solution by symbolically factorizing the returned basis matrix. If this approach fails, they continue with an exact rational simplex, warm-started from the final floating-point basis. This approach was refined by Applegate et al [11, 1] in an algorithm called incremental precision boosting, which is implemented in the solver QSopt_ex. In each iteration of the algorithm, the floating-point precision is increased until the basis can be proven to be exactly optimal. Another state-of-the-art algorithm for solving LPs exactly is based on LP iterative refinement [14, 13] and is implemented in the LP solver SoPlex. It avoids higher-precision LP solves by instead solving a series of error-correcting LPs in double-precision in order to produce a sequence of primal-dual solutions with residual errors converging to zero.

Although LP iterative refinement was shown to outperform precision boosting in the majority of cases, the precision boosting algorithm is more robust on numerically difficult instances. The LP iterative refinement procedure has no reliable way of recovering if the floating-point LP subroutine aborts with a failure due to numerical issues. This has been observed in practice both on pure LPs [14, 13] and for LP relaxations of exact MIP subproblems during branch-and-cut [9].

In this article, we propose a natural combination of these two algorithms that profits from the speed of LP iterative refinement, but can use precision boosting as a fallback to overcome numerical issues. We show that this combination is more robust and faster than either of the algorithms individually.

The remainder of this paper is structured as follows. In Section 2 we introduce exact LP solving formally and give a brief review of the two base algorithms. In Section 3, we present the combined algorithm, giving details on when and how the precision boosting technique is used. In Section 4, we conduct a computational study, evaluating the different algorithms both in the context of pure exact LP, as well as in experiments with an exact MIP framework. Finally, in Section 5, we conclude our findings and give an outlook on future work.

# 2 Existing Methods to Solve LPs Exactly

We aim to solve a linear program

$$\min\{c^T x | Ax = b, x \geq \ell\} \tag{1}$$

where $A \in \mathbb{Q}^{m \times n}$ is a rational matrix of full row rank with $m \leq n$, $c \in \mathbb{Q}^n$ is the objective function, and $\ell \in \mathbb{Q}^m$ is the lower bound vector. Note that we choose this formulation to keep the notation simple. More general formulations are possible and discussed in detail, e.g., in [14].

Our goal is to solve this LP exactly over the rational numbers, i.e., any feasibility or optimality tolerances that are often used in LP solvers are set to zero. As discussed in Section 1, we focus on algorithms based on the simplex method. Both LP iterative refinement and precision boosting iteratively produce a sequence $(x_n, y_n)$ of approximate solutions that are more and more accurate. With any such algorithm, we can then use the methods described in [13] to obtain an exact solution, either by reconstructing it from an approximate solution as described in [13], or by solving the linear system defining the current basis exactly.

In the following, we describe the two existing methods that we combine in this paper. We refer to [11] for a more detailed description of precision boosting and to [14, 13] for a more detailed description of LP iterative refinement.

## 2.1 Incremental precision boosting

This algorithm first computes an approximate solution using a floating-point simplex implementation in double-precision arithmetic and checks the resulting basis for exact primal and dual feasibility. In the original algorithm [11] this check is always performed by means of a rational LU factorization of the basis matrix and subsequent triangular solves in rational arithmetic. If the basis is detected as not optimal, the arithmetic precision is increased and all floating-point tolerances are decreased. After double precision, the first level of extended precision uses 128 bits (quad precision); subsequently, the precision is grown by a factor of 1.5 in each boosting step.

The tolerances are decreased at the same rate as the precision is increased. Given a value of $2^a$ for some $a \in \mathbb{Q}$ as the tolerance's default value in double precision and $p$ bits for the mantissa in higher precision, the corresponding tolerance value is set to $2^{a\frac{p}{64}}$. Note that this leaves a "buffer", as the mantissa in double precision only has 53 bits. As an example, a tolerance value of $10^{-6}$ in double precision would be scaled to approximately $3 \cdot 10^{-11}$ in quad precision.

If an iteration returns that the LP is infeasible, the algorithm attempts to turn the approximate Farkas proof into an exact proof of infeasibility. A short algorithmic description for a feasible LP can be found in Algorithm 1.

**Algorithm 1** Incremental precision boosting

**Input**: $c, \ell \in \mathbb{Q}^n, b \in \mathbb{Q}^m, A \in \mathbb{Q}^{n\times m}$
**Output**: primal-dual solution $(x^*, y^*) \in \mathbb{Q}^{n+m}$ of (1), basis $\mathcal{B}$
**for** $p \leftarrow 64, 128, 192, 288, \ldots$ **do**
  load $\bar{A}, \bar{b}, \bar{c}, \bar{\ell}$ with precision $p$
  decrease floating-point tolerances
  solve $\min\{\bar{c}^T x | \bar{A}x = \bar{b}, x \geq \bar{\ell}\}$ in precision $p$
  $\mathcal{B} \leftarrow$ returned basis
  symbolically compute solution $(x^*, y^*)$ corresponding to $\mathcal{B}$
  **if** $(x^*, y^*)$ is primal and dual feasible **then**
    **return** $(x^*, y^*), \mathcal{B}$
  **end if**
**end for**

The precision boosting algorithm is often very efficient, as it has been observed that the basis returned by the first floating-point LP solve is often already exactly optimal [7]. Furthermore, the higher numeric precision on successive iterates makes this algorithm very robust on problems that are numerically difficult, even in the presence of large coefficient ranges or ill-conditioned basis matrices. Its downside is that the computations in higher precision can be time-consuming, and the exact factorization of the basis matrix after every iteration can pose an additional bottleneck in some problem instances.

## 2.2 LP iterative refinement

In an attempt to overcome these issues, the *LP iterative refinement* algorithm was introduced [14], which is based on iterative refinement for linear systems [26]. Instead of increasing the numerical precision of the floating-point computations, LP iterative refinement computes the violations in the reduced costs, right-hand sides, and the lower bounds. Then, those residuals are scaled and inserted in place of the original objective, right-hand sides, and lower bounds, respectively.

At each iteration, this transformed problem is solved in double-precision floating-point arithmetic. Afterwards, the original solution is updated by adding an unscaled version of the transformed solution. This correction is performed in rational arithmetic, and the resulting solution is then used as a starting point for the next iteration. The algorithm for a feasible LP is provided in Algorithm 1.

---

**Algorithm 2** Iterative refinement for linear programming

---

**Input**: $c, \ell \in \mathbb{Q}^n, b \in \mathbb{Q}^m, A \in \mathbb{Q}^{n\times m}$, scaling limit $\alpha$
**Output**: primal-dual solution $(x^*, y^*) \in \mathbb{Q}^{n+m}$ of (1), basis $\mathcal{B}$

```
δ_P^1 ← 1, δ_D^1 ← 1
load Ā, b̄, c̄, ℓ̄ in double precision                                    {initial solve}
solve min{c̄^T x | Āx = b̄, x ≥ ℓ̄} approximately
(x^1, y^1) ← returned approximate solution, B ← returned Basis
for k ← 1, 2, ... do
  symbolically compute solution (x*, y*) corresponding to B            {check
  termination}
  or reconstruct (x*, y*) from (x^k, y^k)
  if (x*, y*) is primal and dual feasible then
      return (x*, y*), B
  else
      b̂ ← b − A x_k, ℓ̂ ← ℓ − x_k                                         {compute error}
      ĉ ← c − y_k^T A
      δ_P^k ← max{‖b̂‖_∞, ‖ℓ̂‖_∞}
      δ_D^k ← max{0, max{−ĉ_i | i = 1, ..., n}}
      Δ_P^{k+1} ← 1/ max{δ_P^k, (α Δ_P^k)^{−1}}                          {compute scaling}
      Δ_D^{k+1} ← 1/ max{δ_D^k, (α Δ_D^k)^{−1}}
      b̄ = Δ_P^{k+1} b̂, ℓ̄ = Δ_P^{k+1} ℓ̂, c̄ = Δ_D^{k+1} ĉ                   {solve transformed}
      solve min{c̄^T x | Āx = b̄, x ≥ ℓ̄} approximately
      (x̂, ŷ) ← returned approximate solution
      x_{k+1} ← x_k + 1/Δ_P^{k+1} x̂                                      {update solution}
      y_{k+1} ← y_k + 1/Δ_D^{k+1} ŷ
  end if
end for
```

---

In the case that an iteration detects floating-point infeasibility, the LP iterative refinement algorithm solves the auxiliary *feasibility problem* [14]

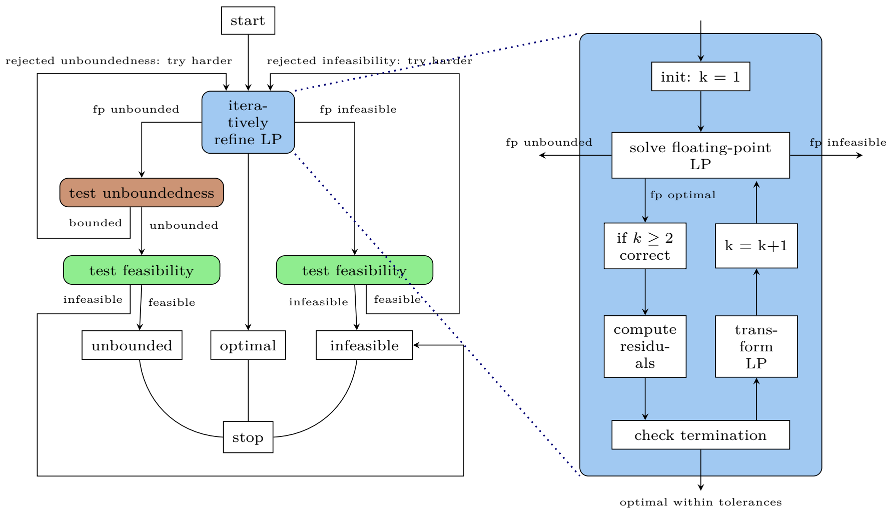
$$\max\{\tau \mid A\xi - (b - A\ell)\tau = 0,\ \xi \geq 0,\ \tau \leq 1\}. \tag{2}$$

exactly with the previosuly described LP iterative refinement algorithm. If unboundedness is detected during an iteration, both a primal feasible solution, as well as an unbounded direction of improvement need to be computed. A primal feasible solution can be computed by solving the feasibility problem (2) as described above. To compute an unbounded ray, the *unboundedness problem*

$$Av = 0,\ c^T v = -1,\ v \geq 0. \tag{3}$$

is solved. A flowchart describing the full algorithm can be seen in Figure 1.

Figure 1: Iterative refinement for exact linear programming, reprinted from [14]




The strength of LP iterative refinement is that it can perform all simplex solves in fast double-precision arithmetic and requires symbolic computations only to recover exact solutions and to create the transformed problems. This results in faster running times on instances that can be solved by LP iterative refinement [14]. The downside is that it cannot reliably recover when the floating-point solver fails due to numerical difficulties. Furthermore, if the floating-point solver reports infeasibility but the feasibility problem (2) is feasible, there is also no reliable recovery mechanism. In those cases, the pure LP iterative refinement algorithm attempts to overcome the difficulties by trying different setting combinations of presolving, scaling, ratio testing, pricing, increasing the Markowitz threshold, as well as changing tolerances.

In order to harness the individual strengths of both algorithms, we propose a combination, described in the following section.

# 3 Combining Precision Boosting and LP iterative refinement

As the LP iterative refinement algorithm tends to be faster whenever it succeeds, we use it as the main algorithm in our approach, and use precision boosting only whenever LP iterative refinement fails. Concretely, this can happen in three cases.

## 3.1 Possible failures of pure LP iterative refinement

**Failure due to numerical troubles.** The first case occurs when the floating-point solver, executed inside LP iterative refinement, fails to terminate with an approximately optimal solution. Reasons for this can be that a linear system $Bx = b$ needs to be solved where $B$ is numerically singular, or that cycling occurs, i.e., the same few variables enter/leave the basis repeatedly. In these cases, the floating-point solver will return correctly with a non-optimal solution status. A different case of numerical difficulties arises when the floating-point solver claims to have reached an optimal basis, but in fact returns a solution with large residual errors, i.e., close to or even exceeding one. Then, the LP iterative refinement algorithm can not be guaranteed to converge to an optimal solution [13].

**Failure due to stalling.** The second case occurs when the LP iterative refinement method fails to significantly reduce violations for too many iterations in a row. By default, LP iterative refinement aborts after two consecutive iterations where the maximum violation was decreased by a factor less than $2^4$.

**Failure due to incorrect status.** The last two cases are when the floating-point solver incorrectly detects an instance known to be feasible as infeasible, or an instance known to be bounded as unbounded. Firstly, this situation can occur when the auxiliary LPs (2) for checking feasibility or (3) for checking boundedness are solved exactly by LP iterative refinement: By construction, (2) is feasible and bounded, and (3) is bounded. Secondly, this can happen for the original LP after feasibility and boundedness has been established by exact solution of one or both of these auxiliary LPs.

In the original LP iterative refinement method presented in [14], running into numerical problems or incorrect status claims would enable a recovery mechanism that modifies certain settings such as presolving, scaling, tolerances, etc. Since this mechanism involves restarting the solving process from scratch, it can be very time-consuming and is disabled in our combined approach. Instead, whenever one of these problems occurs, we boost the precision and restart the LP iterative refinement procedure in higher precision.

## 3.2 Boosting the precision

In the following, we explain important details of the combined algorithm. A flowchart illustrating the algorithm can be seen in Figure 2.

The precision boosting step can be split into three parts: First, the default arithmetic precision of all operations and data structures is increased to the new precision, then the LP is approximated in the increased precision from the rational LP, and finally the tolerances of the solver are decreased.

**Increase precision.** We increase the precision very similarly to QSopt_ex [11], which means first solving in double precision, then with $128, 192, 288, \ldots$ bits. Due to implementation details we currently do not support quad precision (128 bits), but instead directly increase to 192 bits after the double-precision solve. A second difference is that we set the maximal precision limit at 1000 bits, because above this precision some tolerances in SoPlex (that are expressed in double precision) would automatically be rounded to zero. This is smaller than the 3164 bit limit of QSopt_ex, but note that the maximal precision is never reached in any of our experiments.

**Load LP from rational LP.** Computing a more accurate approximation of the rational LP after each precision boost is necessary. In fact, in some instances, the roundoff-errors introduced when approximating the rational LP in floating-point precision are the reason why the floating-point solver returned a wrong basis. Consequently, after each precision boosting step, the rational coefficients $A, b, \ell, c$ of LP (1) are approximated with the increasingly accurate floating-point coefficients $\bar{A}, \bar{b}, \bar{\ell}, \bar{c}$.

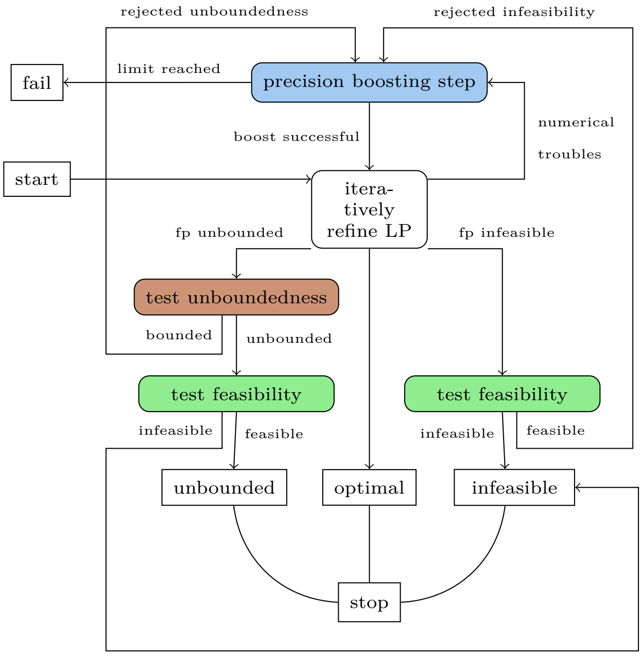



Figure 2: Flowchart illustrating the combined algorithm

**Decrease tolerances.** The last part of the precision boosting step is the update of the tolerances in the floating-point solver. Note that in contrast to a pure precision boosting approach, we are not required to change the feasibility and optimality tolerances of the floating-point solver, since this is handled by the LP iterative refinement procedure. However, the tolerances below which numbers are considered zero by the different components of the solver do need to be decreased. Similarly to QSopt_ex [11], we use constant factors to scale the tolerances in accordance with the current precision. Due to implementation details of the library we use to express higher-precision numbers, we express our tolerances in relation to the current precision, approximated in base 10. For a precision of $p$ bits, let $p'$ be the such that $2^{-p} = 10^{-p'}$, i.e., $p' = p\frac{log(2)}{log(10)}$. Then we set the tolerances as $10^{-\lfloor p' * c \rfloor}$, where $c$ is a constant that can be different for each tolerance, e.g., $c = 1$ for the tolerance below which values are considered zero during the solve and $c = 0.625$ for the tolerance below which values are considered as zero during pivot element selection. The values of the constants are chosen to be consistent with the tolerance values in double precision.

After the precision has been increased, we restart the LP iterative refinement procedure. It is clear that it is advantageous to not solve from scratch with higher-precision but rather to warm-start the solving process from an advanced basis. If precision boosting was performed because of stalling, it is clear that we can restart from the last floating-point optimal basis. However, if precision boosting was performed because of numerical difficulties, it is desirable to restart with a more stable basis. To achieve this, we store the basis at a geometrically increasing frequency: if the number of iterations is a power of two, and at least all 10000 iterations.

If unboundedness or infeasibility is detected by the floating-point solver during the solution process, first one of the auxiliary LPs (2) or (3) is created and solved exactly with the combined algorithm. If feasibility respectively boundedness has already been established by solving such an auxiliary problem, then any future claim of infeasibility respectively unboundedness immediately leads to a precision boosting step. Furthermore, we store separate advanced bases for the original, the feasibility, and the unboundedness problem.

### 3.3 Convergence guarantees of the combined algorithm

The convergence proof of LP iterative refinement relies on the notion of a *limited-precision LP oracle* [13]. An oracle is a limited-precision LP oracle, if there exist constants $\sigma > 0$ and $0 < \eta < 1$, such that the oracle can, for any LP (1), produce an approximate primal-dual floating-point solution $(\bar{x}, \bar{y})$ with

$$\|A\bar{x} - b\|_\infty \leq \eta, \tag{4a}$$

$$\bar{x} \geq \ell - \eta\mathbb{1}, \tag{4b}$$

$$c - A^T\bar{y} \geq -\eta\mathbb{1}, \tag{4c}$$

$$|(\bar{x} - \ell)^T(c - A^T\bar{y})| \leq \sigma. \tag{4d}$$

In other words, all primal and dual violations are bounded by $\eta$, and complementary slackness violations are bounded by $\sigma$.

Since our combined algorithm still uses LP iterative refinement, it is clear that any convergence guarantee for the pure LP iterative refinement approach immediately transfers to the new algorithm. If our base algorithm to solve the LP is restricted to double precision, then it is unreasonable to expect it to be able to act as a limited-precision LP oracle for all possible inputs. Hence, LP iterative refinement currently lacks a self-contained convergence guarantee.

This is where our combination with precision boosting does not only provide a *practical*, but also a *theoretical* contribution. If we allow to increase the precision to any arbitrary number, then it becomes possible to construct such an oracle and prove an unconditional convergence guarantee.

A detailed proof of convergence to a solution that satisfies (4a) – (4d) is beyond the scope of this paper, but intuitively it is clear that a solution with a primal and dual violation of at most $\eta < 1$ can be found, if the simplex can be executed in arbitrarily high precision and the largest numerical errors encountered during the entire course of the algorithm tend towards zero as the precision increases. However, note that any such statement must depend on the implementation of the underlying numerical linear algebra routines, in particular the LU update. Many modern simplex implementations use Forrest-Tomlin updates [12] in order to update the LU factorization of the basis matrix. While computationally very efficient due to its sparsity-preserving properties, this update is not proven to be backward stable. For a convergence proof, it would suffice to use the Bartels-Golub update [3] instead, which is known to be backward stable [2].

In this regard, note that to our knowledge the literature currently lacks a theoretical proof of convergence for pure precision boosting. Such a proof would require special attention in decreasing the primal and dual feasibility tolerances. The current implementation of QSopt_ex [11], e.g., decreases the primal and dual feasibility tolerance by the same order of magnitude by which the precision is increased, i.e., the relative difference between target tolerance and numerical accuracy remains unchanged. This seems to work well in practice. However, in order to prove convergence to an optimal basis theoretically, this relative difference would need to grow with every boosting step, i.e., tolerances would need to decrease at a slower rate than the rate at which precision increases. The results in [24] suggest that numerical stability is achieved as long as the tolerances exceed the forward error of the computed solution.

By contrast, our proof of convergence for LP iterative refinement with precision boosting does not suffer from such interdependencies. The reason is that precision boosting is only used to force the primal and dual violations below $\eta < 1$, and this $\eta$ can remain fixed.

## 4 Computational Study

We investigate the performance of our proposed combined algorithm, comparing it to both previously existing methods on a large set of LPs and MIPs. In particular, we are interested in the following questions. First, in the context of pure LP, *how does the combined algorithm compare to pure LP iterative refinement and pure precision boosting, and what are the strengths of the individual methods?* From the existing literature, we expect LP iterative refinement to be faster on instances where it works, while precision boosting is supposed to handle numerically challenging problems with more consistency. As this is the first time both methods were implemented inside the same LP solver, we want to verify if this is still the case. Second, we want to determine on which sets of instances the respective algorithms perform better, and see if the combined algorithm can leverage the strengths of both parts. This is discussed in Section 4.2. Third, in the context of MIP, we want to determine *if the performance improvements from pure LP experiments translate to MIP solving, and if the combined algorithm can help to reduce the number of failed calls to the exact LP solver.* Especially the last part is of interest, since failing exact LPs was an issue when introducing numerical cutting planes to an exact MIP solver [9], where custom techniques were developed to make cuts numerically easier for the exact LP solver. This is discussed in Section 4.3.

### 4.1 Setup and test set

The experiments were all performed on a cluster of Intel Xeon Gold 5122 CPUs with 3.6 GHz and 96 GB main memory. For all symbolic computations, we use the GNU Multiple Precision Library (GMP) 6.1.2 [15]. All compared algorithms are implemented within SoPlex 6.0.3, and are freely available on GitHub[1]. For exact MIP experiments we use a development version of SCIP, which is also publically available[2] and which uses PaPILO 2.0.1 [18] for exact rational presolving.

For the pure LP tests, we use two different test sets. The first, which we call LPLIB, is a collection of instances from [14], containing instances from the Netlib LP test set including the kennington folder, Hans Mittelmann's benchmark instances, Csaba Mészáros's LP collection, the LP relaxations of the COR@L mixed-integer programming test set, and the LP relaxations of the MIPs from MIPLIB instances up to and including MIPLIB 2010. The second test set, which we call CUTLIB, is comprised of 100 instances that all stem from subproblems encountered by exact SCIP for instances of the MIPLIB 2017 benchmark set that proved difficult for the exact LP solver [9].

For MIP experiments, we use the MIPLIB 2017 benchmark instances; in order to save computational effort, we exclude all those that could not be solved by the floating-point default version of SCIP 8.0 within two hours. We use three random seeds for the remaining 132 instances, making the size of our test set 396. Note that for the pure LP experiments, we ran each instance only once since performance variability is not as pronounced in pure LP experiments. The time limit was set to 7200 seconds for all experiments. For the LP experiments we report aggregated times in shifted geometric mean with a shift of 0.1 seconds, and LP iterations in shifted geometric mean with a shift of 10. For the MIP experiments, we use a shift of 1 second for the time and 100 nodes for the number of nodes.

[1] https://github.com/scipopt/soplex

[2] It can be obtained from https://github.com/scipopt/scip/tree/exact-rational.

## 4.2 Pure LP experiments

We compare the following three settings:

- pure LP iterative refinement with precision-boosting disabled, referred to as IR-DOUBLE,
- pure precision boosting with LP iterative refinement disabled, referred to as BOOSTING-PURE, and
- the proposed combination of both, referred to as IR-BOOSTING.

In all three settings, we use a rational factorization of the final basis matrix to test for exact optimality. Although there exist instances where the reconstruction approach presented in [14] performs better, in preliminary experiments the factorization approach clearly outperformed the reconstruction approach on average for all three settings. Consequently, on instances where the final basis of the initial double-precision simplex solve is already optimal, there is no difference between the settings. We therefore only report results for instances where the final basis of the initial double-precision simplex solve is not optimal. This leaves us with 84 instances from LPLIB and 91 instances from CUTLIB. Furthermore, the time and iterations for the initial floating-point LP solve are the same for all three settings, since they all use the same floating-point simplex solver. We therefore only report the time and iterations *after* the initial floating-point LP solve has finished.

As reported in Table 2, IR-BOOSTING as well as BOOSTING-PURE, were able to solve 79 of the 84 instances in LPLIB, with 5 timeouts, while IR-DOUBLE solved 44 instances, and had to abort on the remaining 40 instances. On CUTLIB, IR-BOOSTING and BOOSTING-PURE were able to solve all 91 instances, while IR-DOUBLE solved 26 instances. This demonstrates that some form of precision boosting is crucial for solving these numerically more difficult instances.

In terms of solving time, we compare all three settings on the subset of instances that were solved to optimality by all three settings, since the instances on which IR-DOUBLE aborts unsuccessfully are not useful for comparison.

Table 1 reports aggregate results on both test sets. Comparing IR-BOOSTING and IR-DOUBLE, we observe that IR-BOOSTING is slightly faster on LPLIB (by 3.5%) and slower on CUTLIB (by 16.2%). On these instances, BOOSTING-PURE performs the worst, with a slowdown of 54.8% on LPLIB, and 100% on CUTLIB, compared to IR-BOOSTING. Some of this can be explained by the selection of instances, which include the instances where IR-DOUBLE successfully recovers from a faulty claim of infeasibility. More explicitly, the first floating-point solve claims infeasibility, but the feasibility test (2) shows that the instance is feasible. In that case, IR-DOUBLE can sometimes recover by restarting, and those cases appear in the test set of all optimal instances, while the cases where the recovery is not successful are excluded. In that sense, looking at the subset of all optimal instances is biased in favor of IR-DOUBLE. Nevertheless, it is clear that IR-DOUBLE is faster than BOOSTING-PURE on instances that can be solved to optimality by both settings, while being comparable with IR-BOOSTING.

When the basis of the initial floating-point LP solve is not optimal, BOOSTING-PURE and IR-BOOSTING solve the same set of instances. Hence, we can compare them fairly using the aggregate results from Table 2, which includes instances that fail with some setting. On LPLIB, IR-BOOSTING is 17.3% faster than IR-DOUBLE, while on CUTLIB, IR-BOOSTING is 5.5% slower than IR-DOUBLE. This indicates that on sets of instances where the LP iterative refinement algorithm already works well, it is beneficial to avoid precision boosting by running LP iterative refinement first. On the other hand, in 71% of the instances of CUTLIB, LP iterative refinement fails and the IR-BOOSTING algorithm

Table 1: Comparison of the different variants on the subset of instances that where solved to optimality by all solvers, and where the final basis of the initial (double) precision LP solve is not optimal. Column "initial" shows the number of simplex iterations in initial precision, column "boosted" shows the number of iterations after the first precision boost.

| | | | | iterations | |
|---|---|---|---|---|---|
| Test set | setting | size | time | initial | boosted |
| LPLIB | IR-BOOSTING | 44 | 1.15 | 25.72 | - |
| | IR-DOUBLE | 44 | 1.19 | 25.72 | - |
| | BOOSTING-PURE | 44 | 1.78 | 26.00 | 17.70 |
| CUTLIB | IR-BOOSTING | 26 | 0.37 | 26.91 | 0.71 |
| | IR-DOUBLE | 26 | 0.31 | 26.57 | - |
| | BOOSTING-PURE | 26 | 0.74 | 18.30 | 22.26 |

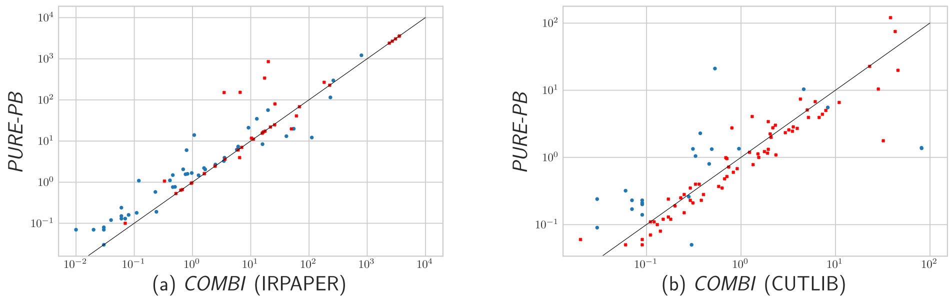
Table 2: Comparison of the different variants on the subset of instances where the final basis of the initial (double) precision LP solve is not optimal. Column "initial" shows the number of simplex iterations in initial precision, column "boosted" shows the number of iterations after the first precision boost.

| | | | | | iterations | |
|---|---|---|---|---|---|---|
| Test set | setting | size | solved | time | initial | boosted |
| LPLIB | IR-BOOSTING | 84 | 79 | 4.12 | 59.78 | 13.63 |
| | IR-DOUBLE | 84 | 44 | 0.71 | 25.85 | - |
| | BOOSTING-PURE | 84 | 79 | 5.98 | 52.19 | 44.66 |
| CUTLIB | IR-BOOSTING | 91 | 91 | 0.96 | 108.98 | 8.95 |
| | IR-DOUBLE | 91 | 26 | 0.39 | 89.03 | - |
| | BOOSTING-PURE | 91 | 91 | 0.91 | 37.75 | 36.79 |

needs to boost the precision. In these cases, simply performing precision boosting right from the start is often faster than trying LP iterative refinement first. However, if we split the union of both test sets into the subset of instances where IR-BOOSTING performed at least one precision boost, and the subset of instances where no precision boost was performed, we observe that IR-BOOSTING is 20.9% faster on the instances where no precision boost was performed, while only being 2.1% faster on the other subset.

To summarize, overall IR-BOOSTING performs best among all settings, displaying a good tradeoff between speed on numerically easier instances and robustness on more difficult ones. To further illustrate this point, Figure 3 shows a scatter plot of solving times for IR-BOOSTING vs. BOOSTING-PURE. Points above the diagonal are instances where IR-BOOSTING is faster than BOOSTING-PURE, points below the diagonal are instances where BOOSTING-PURE is faster than IR-BOOSTING. Instances where IR-BOOSTING performed at least one precision boost are marked in red. Although there are outliers, it is clearly visible that IR-BOOSTING is faster than BOOSTING-PURE on the blue instances (where no precision boost was performed), while being slower on the red instances, especially on CUTLIB (right plot).

Figure 3: Comparison of solving times for BOOSTING-PURE vs. IR-BOOSTING. The left plot shows the results for LPLIB, the right plot shows the results for CUTLIB. Instances where IR-BOOSTING performed at least one precision boost are marked in red.




## 4.3 Experiments in an exact MIP solver

Exact LP solving is an essential subroutine for solving MIPs exactly, certainly when following the bybrid-precision LP-based branch-and-bound approach proposed in [6], and whenever continuous variables are present. Although our new proposed variant IR-BOOSTING performed well in pure LP experiments, it is not clear whether this will translate to exact MIP solving. On the one hand, the additional overhead of precision boosting might not be worthwhile in the context of MIP solving, where branching is an alternative to precision boosting. On the other hand, IR-BOOSTING may help to remove the need for custom-made cut-weakening techniques that were developed in [9] in order to reduce the encoding length of coefficents in the LP relaxation.

To investigate these question, we performed experiments using SCIP on the MIPLIB 2017 benchmark instances with four different SCIP settings:

- the default best setting determined in [9], precision boosting disabled, referred to as DENSMALL+IR-DOUBLE
- the same setting, precision boosting enabled (in the IR-BOOSTING variant), referred to as DENSMALL+IR-BOOSTING
- cut-weakening disabled, precision boosting disabled, referred to as SAFEGMI+IR-DOUBLE
- cut-weakening disabled, precision boosting enabled, referred to as SAFEGMI+IR-BOOSTING

To reduce the amount of performance variability as much as possible, we restrict our analysis to the subset of 146 instances where at least one precision-boosting step was performed by at least one of the settings. The results for all of these instances are reported in Table 3.

We observe that enabling precision boosting in the DENSMALL+IR-DOUBLE setting leads to a speedup of 3.8%, with one more instance solved to optimality (3 gained, 2 lost). In the SAFEGMI+IR-DOUBLE setting it leads to a speedup of 4.4% with one less instance solved (4 lost, 3 gained). This very small difference in solving times is due to the large offset incurred from the large number of timeouts. For comparing solving times, Table 4 is more meaningful, since it only includes instances that were solved by at least one. Here, we observe a speedup of 10.9% for DENSMALL+IR-BOOSTING over DENSMALL+IR-DOUBLE, and of 12.8% of SAFEGMI+IR-BOOSTING over SAFEGMI+IR-DOUBLE.

Table 3: Comparison of the four MIP settings on the MIPLIB 2017 benchmark instances. Only those instances are included for which at least one precision boost was performed for one of the settings. Column “nboost” shows the shifted geometric mean of the number of precision boosts performed by the respective setting, column “exlpfails” shows the shifted geometric mean of the number of times the exact LP solver failed to solve an exact LP.

| Setting | size | solved | time | exlptime | nboost | exlpfails |
|---|---|---|---|---|---|---|
| DENSMALL+IR-DOUBLE | 146 | 44 | 4669.29 | 63.1 | 0.0 | 5.4 |
| DENSMALL+IR-BOOSTING | 146 | 45 | 4491.30 | 77.9 | 13.5 | 0.0 |
| SAFEGMI+IR-DOUBLE | 146 | 36 | 5153.33 | 208.9 | 0.0 | 50.8 |
| SAFEGMI+IR-BOOSTING | 146 | 35 | 4924.84 | 204.7 | 73.0 | 0.0 |

Table 4: Comparison of the four MIP settings on the MIPLIB 2017 benchmark instances. Only instances where at least one precision boost was performed for one of the settings, and where at least one settings managed to solve to optimality are included.

| Setting | size | solved | time | nodes | exlptime |
|---|---|---|---|---|---|
| DENSMALL+IR-DOUBLE | 49 | 44 | 1973.50 | 76975.1 | 26.9 |
| DENSMALL+IR-BOOSTING | 49 | 45 | 1757.81 | 72551.4 | 29.7 |
| SAFEGMI+IR-DOUBLE | 49 | 36 | 2646.87 | 82332.1 | 178.3 |
| SAFEGMI+IR-BOOSTING | 49 | 35 | 2306.36 | 68470.1 | 153.8 |

Finally, let us compare the performance of both settings that use precision boosting: DENSMALL+IR-BOOSTING and SAFEGMI+IR-BOOSTING. Although the number of failed exact LPs is zero in both versions, the number of necessary precision boosts is 5.4 times higher in SAFEGMI+IR-BOOSTING, leading to an increase of 162.8% in exact LP time and of 9.6% in overall solving time, as reported in Table 3. We note that the large number of timeouts gives a strong offset to the average solving times given in this table. If we compare only on the subset of instances that could be solved to optimality by at least one setting, then the speedup of DENSMALL+IR-BOOSTING over SAFEGMI+IR-BOOSTING is 31.2%. This is comparable to the speedup of 34.1% of DENSMALL+IR-DOUBLE over SAFEGMI+IR-DOUBLE, which shows that precision boosting does not impact the importance of cut-weakening techniques, but is rather a complementary technique.

# 5 Conclusion

In this article, we presented an improvement to the LP iterative refinement algorithm for solving LPs exactly over the rational numbers. By integrating the precision boosting method inside an outer LP iterative refinement loop, the combined algorithm is more robust on numerically challenging problems and does not suffer from any slowdown on numerically well-behaved instances. This addresses the major shortcoming of LP iterative refinement for exact LP solving, namely the absence of a reliable recovery mechanism in case of numerical difficulties.

We analyze the performance of our new combined algorithm on two different LP testsets, comparing it with pure LP iterative refinement, as well as pure precision boosting. The results show that the combined algorithm outperforms the other methods for solving LPs exactly. It solves more instances than a pure LP iterative refinement approach, and is faster than pure precision boosting. Furthermore, we show that using the combined algorithm as a subroutine for solving exact MIPs, we are able to reduce the number of failed exact LP calls to zero, while remaining as fast as the pure LP iterative refinement approach.

We see two directions for future improvement. First, and most importantly, the rational LU factorization that is used to verify the exact optimality of a final basis can become a major bottleneck on some problem instances. A more involved roundoff error-free LU factorization algorithm, such as the one proposed by [23], could help to alleviate this problem. Second, the missing intermediate step of increasing the precision to hardware-supported 128 bit quad precision could speed up the combined algorithm, as it makes the first boosted iteration more efficient, especially since one precision boost is sufficient in most cases.


## Acknowledgments

We would like to thank Martin Sidaway and his supervisor Paolo Zuliani for their valuable insights into the theoretical convergence of pure precision boosting.


## References


[1] David Applegate, William Cook, Sanjeeb Dash, and Daniel G. Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693 – 699, 2007.

[2] Richard H. Bartels. A stabilization of the simplex method. *Numer. Math.*, 16(5):414–434, feb 1971.

[3] Richard H. Bartels and Gene H. Golub. The simplex method of linear programming using LU decomposition. *Commun. ACM*, 12(5):266–268, 1969.

[4] Miquel Bofill, Felip Manyà, Amanda Vidal, and Mateu Villaret. New complexity results for Łukasiewicz logic. *Soft Computing*, 23:2187–2197, 2019.

[5] Benjamin A. Burton and Melih Ozlen. Computing the crosscap number of a knot using integer programming and normal surfaces. *ACM Transactions on Mathematical Software*, 39(1), 2012.

[6] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305 – 344, 2013.

[7] Marcel Dhiflaoui, Stefan Funke, Carsten Kwappik, Kurt Mehlhorn, Michael Seel, Elmar Schömer, Ralph Schulte, and Dennis Weber. Certifying and repairing solutions to large lps how good are lp-solvers? In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 255–256, USA, 2003. Society for Industrial and Applied Mathematics.

[8] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 2022.

[9] Leon Eifler and Ambros Gleixner. Safe and verified gomory mixed integer cuts in a rational mip framework. *SIAM Journal on Optimization*, 2023. under review.

[10] Leon Eifler, Ambros Gleixner, and Jonad Pulaj. A safe computational framework for integer programming applied to chvátal's conjecture. *ACM Transactions on Mathematical Software*, 48(2), 2022.

[11] Daniel G. Espinoza. *On Linear Programming, Integer Programming and Cutting Planes*. PhD thesis, Georgia Institute of Technology, 2006.

[12] John J. Forrest and John A. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, Feb 1972.

[13] Ambros Gleixner and Daniel E. Steffy. Linear programming using limited-precision oracles. *Mathematical Programming*, 183:525–554, 2020.

[14] Ambros Gleixner, Daniel E. Steffy, and Kati Wolter. Iterative refinement for linear programming. *INFORMS Journal on Computing*, 28(3):449–464, 2016.

[15] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, London, GBR, 2015.

[16] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.

[17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, and et al. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017.

[18] Alexander Hoen and Leona Gottwald. scipopt/papilo: v2.0.0, April 2022.

[19] Franklin Kenter and Daphne Skipper. Integer-programming bounds on pebbling numbers of cartesian-product graphs. In Donghyun Kim, R. N. Uma, and Alexander Zelikovsky, editors, *Combinatorial Optimization and Applications*, pages 681–695, 2018.

[20] L.G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[21] Giuseppe Lancia, Eleonora Pippia, and Franca Rinaldi. Using integer programming to search for counterexamples: A case study. In Alexander Kononov, Michael Khachay, Valery A Kalyagin, and Panos Pardalos, editors, *Mathematical Optimization Theory and Operations Research*, pages 69–84, 2020.

[22] Joshua A. Lerman, Daniel R. Hyduke, Haythem Latif, Vasiliy A. Portnoy, Nathan E. Lewis, Jeffrey D. Orth, Alexandra C. Schrimpe-Rutledge, Richard D. Smith, Joshua N. Adkins, Karsten Zengler, and Bernhard O. Palsson. In silico method for modelling metabolism and gene product expression at genome scale. *Nature Communications*, 3(1):929, Jul 2012.

[23] Christopher Lourenco, Adolfo R. Escobedo, Erick Moreno-Centeno, and Timothy A. Davis. Exact solution of sparse linear systems via left-looking roundoff-error-free lu factorization in time proportional to arithmetic work. *SIAM Journal on Matrix Analysis and Applications*, 40(2):609–638, 2019.

[24] W. Ogryczak. The simplex method is not always well behaved. *Linear Algebra and Its Applications - LINEAR ALGEBRA APPL*, 109:41–57, 10 1988.

[25] Jonad Pulaj. Cutting planes for families implying Frankl's conjecture. *Mathematics of Computation*, 89(322):829–857, 2020.

[26] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Inc., New York, 1994.