Solving Very Sparse Rational Systems of Equations

William Cook and Daniel E. Steffy

School of Industrial and Systems Engineering

Georgia Institute of Technology

Efficient methods for solving linear-programming problems in exact precision rely on the solution of sparse systems of linear equations over the rational numbers. We consider a test set of instances arising from exact-precision linear programming and use this test set to compare the performance of several techniques designed for symbolic sparse linear-system solving. We compare a direct exact solver based on LU factorization, Wiedemann's method for black-box linear algebra, Dixon's p-adic-lifting algorithm, and the use of iterative numerical methods and rational reconstruction as developed by Wan.

---

1. INTRODUCTION

Solving systems of linear equations is a fundamental mathematical problem. Several methods have been proposed to efficiently solve rational or integer systems of linear equations exactly [Dixon 1982; Kaltofen and Saunders 1991; Mulders and Storjohann 1999; Dumas et al. 2002; Chen 2005; Chen and Storjohann 2005; Eberly et al. 2006; Wan 2006]. Such methods rely on the ability to reconstruct rational numbers from high-accuracy floating-point solutions, or from modular solutions.

In this study we seek to determine which exact method is best suited for the type of linear systems of equations arising in the solution of real-world linear-programming (LP) problems. Such problems tend to be very sparse and they have been the focus of much research, due to the wide-ranging application of linear and integer programming. Until recently, software developed to solve LP problems has provided approximate floating-point solutions; commercial LP packages, such as CPLEX [ILOG 2007], attempt to find solutions within fixed error tolerances. For many practical applications, such approximate solutions are sufficient. Nevertheless, there is a demand for exact solutions to LP problems. Exact solutions allow researchers to use linear programming in computer-assisted proofs, allow for sub-routines in exact-precision integer and mixed-integer programming, and are used in

---

other applications requiring certifiably correct solutions. An important example of the use of exact LP methods is Hales' proof of the Kepler Conjecture [Hales 2005].

An effective approach for solving LP problems exactly is to perform the simplex algorithm using inexact floating-point precision, then use symbolic computation to construct, check, and correct the final solution [Kwappik 1998; Dhiflaoui et al. 2003; Koch 2004; Applegate et al. 2007]. Our description of the technique follows that adopted in the QSopt_ex code of Applegate et al. [2007]. After performing the simplex algorithm in floating-point precision, the LP solution includes a basis, providing a square system of linear equations that defines that solution in terms of the original input data. The primal and dual solutions associated with this basis can be computed in full rational precision and checked to make sure they satisfy the LP optimality conditions. If the solution is certified as optimal, it is returned. Otherwise, the floating-point precision is increased on the fly and more simplex pivots are carried out to find another solution, then the process is repeated. A similar procedure is followed if a certificate for unboundedness or infeasibility is returned. This incremental strategy is more efficient than carrying out all computations in rational precision throughout the entire simplex method. The exact solution of linear systems in this procedure is a bottleneck; solving these systems quickly can have a large influence on the solve times. Finding a fast and robust method for this setting is the objective of this study.

The focus of this paper is a comparison of four solution procedures for rational linear systems. Our starting point is the LU-factorization routine developed in the QSopt [Applegate et al. 2005] linear-programming code. This routine is engineered specifically for the type of sparse matrices that arise in LP applications. We adopt the QSopt routine in a direct LU-based solver, as well as an implementations of Dixon's p-adic-lifting algorithm [Dixon 1982] and Wan's iterative-refinement method [Wan 2006]. We also consider a rational solver based on the black-box algorithm of Wiedemann [1986]. All four methods are tested on a large collection of instances arising in the exact solution of LP problems.

The paper is structured as follows. The testbed of problem instances is described in Section 2. The four solution methods we consider are described in Section 3. Results from our computational study are presented in Section 4, and conclusions are summarized in Section 5. The testbed of rational linear systems and the computer codes for the rational solvers are freely available at

<div align="center">

`www.isye.gatech.edu/~dsteffy/rational`

</div>

for any research purposes.

We refer the reader to [von zur Gathen and Gerhard 2003] for background material in computer algebra and to [Chvátal 1983; Vanderbei 2001] for material on the simplex algorithm.

2.   TEST INSTANCES FROM LP APPLICATIONS

The linear-programming research community is fortunate to have several publicly-available libraries of test instances. In our study we collected these instances together into a single testbed. The set includes the instances from NETLIB [Gay 1985], MIPLIB 3.0 [Bixby et al. 1998], MIPLIB 2003 [Achterberg et al. 2006], the miscellaneous, problematic, and stochastic collections of [Mészáros 2006], the

collection of [Mittelmann 2006], and a collection of traveling-salesman relaxations [Applegate et al. 2006] from the TSPLIB [Reinelt 1991]. These collections are comprised of instances gathered from business and industrial applications, and from academic studies. The problems range in size from several variables up to over one million variables.

The 695 instances in our testbed were given to QSopt_ex. For each instance that was solved by the code, the optimal basis matrix was recorded. In several cases an optimal solution was not found within 24 hours of computing time. For these examples, the basis from the last exact rational solve employed by QSopt_ex was recorded.

When examining the resulting linear systems, we found groups of instances with very similar characteristics. In these cases, we chose a representative system and deleted the other similar instances. For example, the 37 instances delf000 up to delf036 in the miscellaneous collection of [Mészáros 2006] were replaced by the single instance delf000.

We also ran a pre-processing algorithm to repeatedly remove rows and columns having a single nonzero component. Many such examples existed in our systems, due to the inclusion of slack variables in the LP models. In the resulting collection of reduced problems, we deleted all instances having dimension less than 100.

The final problem set contains 276 instances, with dimensions ranging from 100 to over 50,000. For each instance we have both the square basis matrix and the corresponding right-hand-side vector. Within the computational results section, Table VI includes information on the problem-set characteristics and Table VIII includes details for selected instances.

3. SOLUTION METHODS

3.1 Direct Methods

The QSopt_ex code is based on the floating-point LP solver QSopt [Applegate et al. 2005], which adopts the LU-factorization methods described in [Suhl and Suhl 1990]. We refer to the QSopt double-precision floating-point equation solver as *QSLU_double*. This solver was adapted by Espinoza [2006] to solve over alternate data types, including rational numbers, using the GNU Multiple Precision Library [GMP 2008], and it is included in the QSopt_ex code. We refer to this rational solver as *QSLU_rational*. We created a version of the code to solve over word-sized prime-order fields using a data type with optimized operations; we call this finite-field solver *QSLU_ffield*. Figure 1 gives a performance profile[1] comparing the speed of solving all instances in our test set using these three solvers. Table I gives the geometric mean of the solve-time ratios, normalized by the solve time of QSLU_double.

While the double-precision and finite-field solvers are close in time, solving over the rational numbers is considerably slower. This comes as no surprise, since storing and performing operations on full-precision rational numbers is computationally expensive. This supports the idea that techniques for solving rational systems of

---

[1]A performance profile plots the number of instances solved within a factor x of the fastest method time. The vertical axis represents the number of instances. The horizontal axis gives the solve-time ratios.
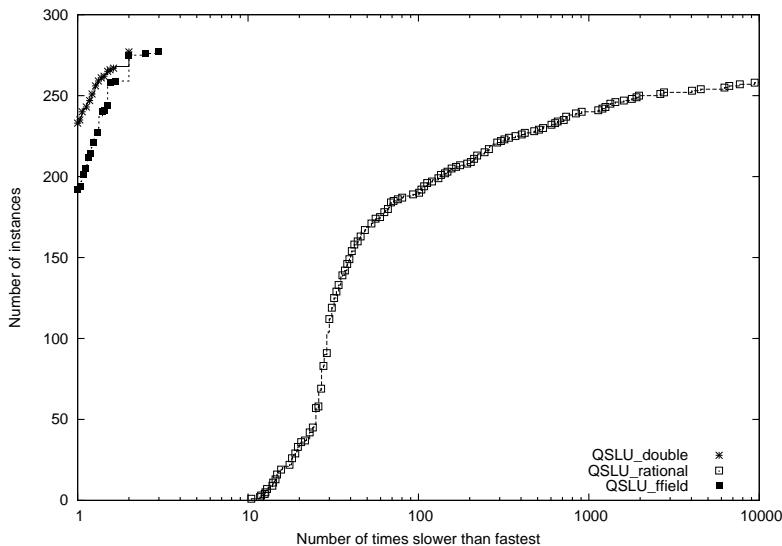
Fig. 1.   Comparison of QSLU Solvers

Table I.   Relative Speed of QSLU Solvers

Solver	Time Ratio
QSLU_double	1.00
QSLU_ffield	1.05
QSLU_rational	89.72

equations that rely on fixed-precision solvers as subroutines could have advantages over direct exact methods. We also tested a direct rational solver using the conjugate gradient method and the GMP library, programmed by Sanjeeb Dash of IBM, and found it much slower than the rational LU-factorization method.

There were many characteristics of individual problems which influenced the deficit in speed between the double-precision solver and the exact rational solver, including the dimension and the complexity of the solutions. The ratio of 89.72 presented in the table can be seen as a characteristic of our problem set and a bound on how much we could possibly hope to improve our speed over the rational solver by using the double-precision and finite-field solvers as subroutines. From the performance profile we also observe that the finite-field solver was the fastest solver on some instances. This may seem surprising as finite-field operations are generally slower than double-precision operations; one possible explanation is the simplification of some operations in the code afforded by exact finite-field computation over

numerical computation, such as checking if an element is zero.

Direct exact-precision methods are not usually thought of as being among the fastest methods for solving systems of linear equations exactly. However, we experienced a reasonable level of success in the sparse setting with QSLU_rational, in some cases outperforming the other methods presented in this paper. A considerable amount of the computational effort of a sparse LU solver is spent finding permutations of the system to reduce the fill-in and to reduce the number of arithmetic operations that must be performed. Such computation depends only on the nonzero structure of the matrix, and this helps QSLU_rational avoid many full-precision arithmetic operations. In the dense setting, we expect a larger performance gap between a direct rational solver and fixed-precision solvers. For dense systems, BLAS routines [Lawson et al. 1979; Dongarra et al. 1990] can be used for fast floating-point linear algebra, and Dumas et al. [2008] have introduced a comparably fast system for dense linear algebra over finite fields.

3.2 Rational Reconstruction

3.2.1 *Basic Results.* The following well-known result is stated as it appears in [Schrijver 1986] as Corollary 6.3a.

THEOREM 3.1. *There exists a polynomial-time algorithm which, for a given rational number α and natural number B_d, tests if there exists a rational number p/q with $1 \le q \le B_d$ and $|\alpha - p/q| < \frac{1}{2B_d^2}$, and, if so, finds this (unique) rational number.*

This theorem provides a powerful tool for calculating exact solutions to rational systems of equations $Ax = b$. If an upper bound B_d is computed for the denominators of the components of x and a vector $\hat{x}$ satisfying $|\hat{x} - x|_\infty < 1/2B_d^2$ is computed, then the theorem can be applied component-wise to $\hat{x}$ to compute the exact solution x. There is an analogous result using modular arithmetic.

THEOREM 3.2. *There exists a polynomial-time algorithm which, for given natural numbers n, M, B_n, B_d, with $2B_nB_d \le M$, tests if there exists a rational number p/q with $gcd(p, q) = 1$, $|p| < B_n$, and $1 \le q < B_d$, such that $p = nq \mod M$, and, if so, finds this (unique) rational number.*

This result is a simplified restatement of Theorem 5.26 from [von zur Gathen and Gerhard 2003]. Using this, a rational system of equations can be solved by scaling the system to be integral, bounding the numerator and denominator of the solution vector, computing a solution to the system modulo an appropriate integer M, and reconstructing the exact rational solution. The remaining techniques to solve rational systems of equations discussed in this paper will rely on these two theorems. The methods of determining exact solutions to rational systems of equations by the corresponding algorithms will be referred to as floating-point rational reconstruction and modular rational reconstruction, respectively. The first result is also often referred to as Diophantine approximation.

The polynomial-time algorithms cited in these theorems are based on applying the Extended Euclidean Algorithm (EEA) to find selected continued-fraction convergents. Section 5.10 of [von zur Gathen and Gerhard 2003] gives a description of modular rational reconstruction, including numerical examples, another description

appears in [Kaltofen and Rolletschek 1989]. The EEA appears as Algorithm 3.6 in [von zur Gathen and Gerhard 2003]. Some methods have been studied to accelerate rational reconstruction, including [Pan and Wang 2003; 2004; Monagan 2004; Lichtblau 2005; Khodadad and Monagan 2006]. In our implementation, we use a technique, sometimes referred to as Lehmer's GCD algorithm, to accelerate our computations (see Algorithm 1.3.7 in [Cohen 2000]). The EEA involves successively performing integer divisions. Lehmer's algorithm accelerates the EEA computationally by performing the integer divisions on approximations of the numbers instead of on large integers. Specifically, in our routines we replace extended-precision integer division with floating-point number inversion when possible, carrying out several steps of the EEA based on truncated data and then synchronizing and updating the full-precision data. We found this to speed up the rational reconstruction by at least a factor of two, and more with large inputs. Collins and Encarnación [1995] also used Lehmer's algorithm to speed up rational reconstruction and experienced comparable levels of success.

3.2.2 *Reconstruction Bounds.* The *bitsize* of a nonzero rational number p/q is $log(|pq|)$, and the bitsize of a rational vector is defined to be the maximum bitsize of its components. For a given instance, the exact bitsize of the solution is not known without solving the system, but it can be bounded using Cramer's rule. Cramer's rule states that for a square nonsingular system $Ax = b$, the ith component of the solution vector is determined by $x_i = \frac{\det(A_i)}{\det(A)}$ where A_i is constructed by replacing the ith column of A with b. Computing determinants exactly is computationally expensive, so the Hadamard bound is typically used to provide an upper bound. The Hadamard determinant bound states that $det(A) \leq \|A\|_2^n \leq n^{\frac{n}{2}} \|A\|_{\max}^n$. This gives $log(\|A\|_2^{2n-1} \|b\|_2)$ as a bound on the bitsize of x. Specifically, $B_n = \|A\|_2^{n-1} \|b\|_2$ and $B_d = \|A\|_2^n$ give upper bounds on the numerator and denominator of the solution of $Ax = b$ that are valid for Theorems 3.1 and 3.2. Table II shows the bound on solution bitsize generated by the Hadamard bound, along with the actual bitsize of the solution, for several of the larger instances in our test set.

Table II.   Actual Solution Size vs. Hadamard Bound

Problem	Solution Bitsize	Hadamard Bound
cont11_l	1263	4570016
gen1	439931	1046612
momentum3	159597	11521199

The examples in Table II illustrate that the bitsize of the solution can be much lower than the Hadamard-based bound. This suggests that computation of modular and floating-point solutions based on the Hadamard bound can lead to unnecessary computation and memory use. For the problem cont11_l, computing an approximate solution with 4,570,016 bits for each component would require over 31 gigabytes of memory for storage alone, when a solution with 1,000 times fewer digits gives sufficient information to successfully reconstruct the exact solution. For a

more in depth investigation into the tightness of the Hadamard bound, see [Abbott et al. 1999; Abbott and Mulders 2001].

As an alternative to the Hadamard bound, we can use smaller but possibly incorrect bounds, then verify the results that are obtained. In this scheme, we attempt rational reconstruction on an approximate solution corresponding to the guessed bound, and check the resulting exact solution for correctness. If it is correct, we can terminate, and otherwise we increase the bound and repeat. Correctness of a candidate solution can be easily certified by evaluating the linear equations. In [Chen 2005; Chen and Storjohann 2005], this technique is referred to as *output-sensitive lifting*. This technique is used in a different context by Dumas et al. [2001] where it is referred to as early termination. This method is made especially practical because while computing high-precision solutions by iterative methods, less-precise solutions are encountered at intermediate steps without any extra computation, giving an opportunity to try rational reconstruction. Chen and Storjohann also provide a simple formula to certify solutions obtained via modular rational reconstruction without evaluating all equations.

3.2.3   *Vector Reconstruction.* Reconstructing the solution vector of a system of equations can be achieved by applying Theorem 3.1 or Theorem 3.2 component-wise to approximate or modular solution vectors. Considering information from the entire system of equations can lead to faster methods for reconstructing a solution vector. We discuss two such techniques, one using the relationship of the denominators of the solution components to accelerate rational reconstruction, the second using the equations to deduce some values without reconstruction.

For many systems of equations, the denominators of the components of the solution vector share common factors. The first method we look at exploits this situation. This method is discussed in [Kaltofen and Saunders 1991; Chen and Storjohann 2005] for use in modular rational reconstruction and we call it the *DLCM.* The DLCM technique for modular reconstruction is well known and currently used in other software such as LinBox and NTL [Dumas et al. 2002; Shoup 2008]. Let Δ be the least common multiple of the denominators of the components that have been reconstructed so far. Suppose the next component of the solution we reconstruct is p/q, from n, M, B_n, B_d, as in Theorem 3.2. Compute $n' = \Delta n$ mod M, then reconstruct p'/q' from n', M using bounds $B_n, \lfloor B_d/\Delta \rfloor$, and assign

$$\frac{p}{q} := \frac{p'}{q'\Delta}.$$

Fixing Δ as a factor of the denominator, and then reconstructing the remaining factors of the denominator and the numerator, accelerates the routine because rational reconstruction terminates faster when the denominator bound B_d is lower. In fact, if q divides Δ then p/q can be immediately identified by the rational-reconstruction routine without any steps of the EEA. It is possible to reduce M to a value $M' \geq M/\Delta$, as described in [Kaltofen and Saunders 1991], to further accelerate this procedure.

The DLCM technique can also be applied to accelerate floating-point rational reconstruction. Suppose a component of the solution p/q is to be reconstructed from an approximation α, and a common denominator Δ of other components is

known. Rational reconstruction is applied to find a rational number p'/q' that best approximates $\Delta\alpha$ with denominator less than B_d/Δ, and then the assignment

$$\frac{p}{q} := \frac{p'}{q'\Delta}$$

can be made. Again, by assigning some factors of the denominator ahead of time, we reduce the calculations in the rational-reconstruction routine.

We mention one possible drawback of this technique. For DLCM to work correctly, the bound B_d must be an upper bound on the size of the common denominator of the entire solution vector, while component-wise rational reconstruction only requires B_d to be a bound on the individual denominators of the solution vector's components. The Hadamard bound given in the previous section will always bound the common denominator. However, a smaller bound that is valid for each component individually, but less than the common denominator of the components, can cause this technique to fail. From the statements of the theorems, we see that increasing the bound B_d necessitates the computation of approximate solutions with more digits of accuracy, or solutions modulo a larger number. The following example illustrates this possible drawback.

EXAMPLE 3.1. *If the solution to $Ax = b$ is $x = (1/2, 1/3, \ldots, 1/p_n)$ where p_n is the nth prime number, then a bound of $B_d = p_n$ will suffice for component-wise rational reconstruction. However, the DLCM method requires the bound B_d to be at least $2 \times 3 \times \cdots \times p_n$ to terminate properly.*

Despite the possible drawback highlighted in Example 3.1, this characteristic could also prove advantageous in an output-sensitive lifting algorithm. Suppose output-sensitive lifting is applied with some insufficiently large bounds B_n, B_d. The DLCM method would recognize failure and terminate early after the common denominator Δ grows larger than B_d. Therefore, that rational-reconstruction attempt with a bound too small to determine the actual solution would terminate before reconstructing every component of the solution vector. This can avoid a significant amount of computation that would otherwise be spent in failed vector reconstruction attempts. This strategy is used by Chen and Storjohann [2005] to reduce the overall time spent on rational reconstruction in their rational solver. When reconstructing a vector component-wise a similar early stopping criterion can be set by maintaining a common denominator of the reconstructed components and terminating if it becomes larger than B_d.

The second technique we explored is the use of the equations from the system to deduce some components of the solution vector. Once part of the solution vector is reconstructed, it is possible that the known components, along with the equations, will directly imply the values of unknown components. If the equations are sparse, evaluating an equation to determine the exact rational value of an unknown solution component could be faster than performing rational reconstruction to determine that component. We call this method of reconstructing some components and then deducing all implied components the *ELIM* technique. To apply this technique, the primary challenge is to determine an order in which to consider components for reconstruction, and to determine which equations to use for deducing values.

A matrix A is said to have *lower bandwidth* of p if $a_{ij} = 0$ whenever $i > j + p$

[Golub and van Loan 1983]. The lower-bandwidth minimization problem is the problem of performing row and column permutations on a matrix to minimize its lower bandwidth. If the $n \times n$ matrix A defining a system of equations has lower bandwidth of p, and the final p components of the solution vector are known exactly, then all remaining components can be determined by solving a $n-p$ lower-triangular system of equations by backwards substitution. Determining the minimum number of components of the solution vector that must be reconstructed, in order to deduce the remaining portion of the solution vector from the equations, is equivalent to the lower-bandwidth minimization problem.

We use a greedy heuristic, Algorithm 1, to determine the variable ordering. The algorithm partitions the columns of A into a set R and an ordered list E. Variables corresponding to columns in R will be reconstructed and variables corresponding to columns in E will be deduced using equations from the system. A list $L(i)$ for $i \in E$ is constructed such that $L(i)$ gives an index to a row of A that has a nonzero element in its ith column and has zeros in every column j appearing after i in the list E. Thus, $R \cup E$ gives an ordering to reconstruct the variables, where every variable in R is obtained by rational reconstruction and each variable i in E can be deduced using constraint $L(i)$ and the preceding variables. We use A_j to denote the jth column of A, and we use a_i to denote the ith row of A. The algorithm is described in terms of deleting rows and columns of the matrix; after such deletions, we maintain the original labeling of the remaining rows and columns.

Algorithm 1 Variable Ordering Algorithm

Input: Matrix A {From $Ax = b$}
Initialize: $R = \emptyset$, $E = \emptyset$
while $A \neq \emptyset$ **do**
    Remove any all zero rows a_i from A
    if $\exists i$ such that a_i has a unique nonzero element a_{ij} **then**
        $E := E \cup \{j\}$ {Variable j can be eliminated}
        $L(j) := i$ {Implied by constraint i}
        Remove A_j, a_i from A
    else
        Choose A_j with the maximum number of nonzeros
        $R := R \cup \{j\}$ {Mark column with most nonzeros for rational reconstruction}
        Remove A_j from A
    end if
end while
Return: R, E, L

We found this heuristic effective in reducing the number of variables to be reconstructed. Table III shows the number of variables that could be eliminated by the routine, that is, the number of variables in the list E.

To illustrate the overall effectiveness of the DLCM and ELIM methods, Table IV compares the solve times and loop count of an exact solver based on Dixon's method (introduced later) on our entire problem set. Details of the output-sensitive

Table III.   Percent of Variables Eliminated

Elimination %	Instances (out of 276)
70%+	219
80%+	145
90%+	66
95%+	35

lifting used can be found in Section 4.1. The solve times are expressed as geometric means of the ratios with the time needed for Dixon's method using component-wise rational reconstruction. It also shows the geometric mean of the ratios of how many loops each method performed to achieve the final solution. The ratios presented here compare the total solve times, of which the reconstruction is just a part. This measure is used to consider the variation in solve times because the ELIM method and the component-wise reconstruction are in some cases able to construct a solution with less information than the DLCM method. We also tested a combination of the two techniques, listed as "DLCM and ELIM" in the table. In this case we applied the ELIM routine as it is described but applied the DLCM method to reconstruct the first p components of the solution vector.

From this table we observe that both methods reduce the overall solve time by approximately 60%. The loop ratio here indicates how many loops of p-adic lifting (described in Section 3.4) were required to identify the correct solution by using output-sensitive lifting. As we can see, the DLCM method required a geometric average of 16 % more loops than the component-wise reconstruction, indicating that the phenomenon shown in Example 3.1 does occur to some degree, although much less dramatic than the worst case. Despite these extra loops, the huge speedup and possibility of early termination gained at each reconstruction by the DLCM method, over the component-wise computation, still allows the DLCM to finish much faster. We also see that the ELIM method is able to finish in fewer loops than the component-wise reconstruction in some cases; this would occur when some of the components that are deduced by the elimination routine have representation too large to have been reconstructed component-wise. When combining DLCM and ELIM we see that the number of loops again increases as the DLCM computation terminates the procedure early, resulting in a slight slowdown over the pure ELIM routine.

Table IV.   Improvement Using Vector Reconstruction

Vector RR Method	Solve Time Ratio	Loops Count Ratio
Component-wise	1.00	1.00
DLCM	0.39	1.16
ELIM	0.40	0.97
DLCM and ELIM	0.41	1.15

From these tests see that the DLCM method is fastest for our set of instances. We

therefore use DLCM throughout our modular and floating-point rational-reconstruction routines for the remainder of the paper. Note, however, that the ELIM method is nearly as fast, and on certain classes of sparse problems it may be faster. Dixon's method uses modular rational reconstruction, but in our tests we noticed similar acceleration of the floating-point reconstruction routine using these techniques.

We also performed tests comparing the speed of the component-wise and DCLM methods for reconstructing solution vectors, excluding the time required to solve the linear system. To make this comparison we considered the solution vector from each problem. We determined a bound for the component-wise reconstruction to terminate correctly, which was $B_1 = 2B^2$ where B is the largest numerator or denominator in any component of the solution. We also computed a bound valid to reconstruct the vector using the DLCM method, which is $B_2 = 2B^2$ where B is the largest of the numerator in any component of the solution or the least common denominator for the solution vector, whichever is larger. We found that on 129 out of 276 problems the bounds were the same, and that the geometric average of $\log(B_2)/\log(B_1)$ was 1.21, which corresponds to the loop ratio in Table IV.[2] As we saw in Table IV, DLCM was still a faster overall strategy despite these extra loops, but there were some instances where this ratio was quite large and effected speed. On some problems this ratio was as high as 66, and the Dixon solver using output-sensitive lifting and component-wise reconstruction was over 100 times faster than the DLCM-based Dixon solver. We also considered the time required to reconstruct the solution by component-wise reconstruction at B_1, divided by the time to use DLCM vector reconstruction at bound B_2; we found the geometric mean of these ratios to be 7.80. To have a pure direct comparison of these we also compared component-wise reconstruction and DLCM vector reconstruction both at B_2 and found their ratio to be 8.92, and found DLCM to be as much as 75 times faster on individual problems. These final numbers indicate that, although component-wise reconstruction can be faster on some problems, the DLCM method can provide a huge speedup overall.

3.3 Iterative Refinement

Applying Theorem 3.1, a rational system of equations can be solved exactly given a bound on the size of the denominators of the solution and an approximate solution within a required degree of accuracy. The previous section addressed how to reconstruct rational solutions and how to choose a proper bound. Here we discuss a method to determine the approximate solution. When the number of digits of accuracy required of the approximate solution is large, solving the system in extended-precision floating-point arithmetic can be as slow as solving the system directly in rational precision, or slower. The iterative refinement procedure, described below, allows us to use repeated approximate floating-point solves to construct an extended-precision solution, taking advantage of the speed of a floating-point LU

---

[2]One would expect this number to match the 1.16 in Table IV. The difference occurs because the table is generated by timings on our Dixon solver which uses an output-sensitive lifting scheme that will not attempt rational reconstruction at every loop. When the bounds B_1, B_2 are close to each other the solvers will often finish at the same time, even if the bounds are not equal. As we are taking the geometric mean in both cases this explains the gap; the arithmetic mean would be expected to be the same.

solver.

Iterative refinement is the process of finding and refining an approximate solution. Once a system is solved approximately, the exact error of the approximate solution can be determined, and further approximate solves can be used to help correct the error. Repeating this process gives solutions that are increasingly accurate. State of the art floating-point solvers typically perform iterative refinement to refine a double-precision solution so that the backwards error is close to machine epsilon.

Ursic and Patarra [1983] adopted iterative refinement to obtain high-accuracy approximate solutions, and combined this with rational reconstruction to solve linear systems of equations exactly. Wan [2006] introduced an improved version of this algorithm, reducing the number of extended-precision operations that are required. Wan's method works with systems of equations that are integer; rational systems are handled by scaling the entries to be integral. An analysis of the complexity of this method, and comparison with other techniques, is presented in [Wan 2006].

An outline of Wan's method is given in Algorithm 2. A solution x/D, where x is integral and D is an integer common denominator, is constructed and refined to become more accurate. At each step, a scaled measure of the error is maintained: $\Delta = (b - A(x/D))D$. The key advantage to this algorithm, over previous ones, is that by scaling and rounding the approximate solutions to be integral at each step, Δ is computed and updated exactly without using extended-precision arithmetic. A scale factor α is chosen at each step, determining the factor by which the common denominator D will be increased. The factor α is chosen relative to the accuracy of the correction $\hat{x}$ chosen at that step, and it is made as large as possible while allowing the newly computed error measure Δ to be stored exactly in standard precision.

Algorithm 2 Iterative Refinement

Input: A, b	$\{Ax = b$ is system to be solved$\}$
Compute a numerical LU factorization of A	
$x := 0$	$\{$Numerator of the solution vector$\}$
$D := 1$	$\{$Common denominator of the solution vector$\}$
$\Delta := b$	$\{$Error measure of solution at each step$\}$
while Desired Accuracy Not Achieved **do**	
Compute $\hat{x} :\approx A^{-1}\Delta$	$\{$Using numerical LU factorization$\}$
Choose an integer $\alpha < \frac{\|\Delta\|_\infty}{\|\Delta - A\hat{x}\|_\infty}$	$\{$This is the scale factor$\}$
Set $[\hat{x}] \approx \alpha\hat{x}$	$\{$Round to the nearest integer$\}$
$\Delta := \alpha\Delta - A[\hat{x}]$	$\{$Update the residual$\}$
$D := D \times \alpha$	$\{$Update the denominator$\}$
$x := x \times \alpha + [\hat{x}]$	$\{x/D$ becomes an increasingly accurate approximation$\}$
end while	
Return: x/D	$\{$Floating-point approximation meeting required accuracy$\}$

We implemented a version of iterative refinement using Wan's strategy and the solver QSLU_double. Incorrect choices of α can quickly cause the algorithm to fail, so computing an error measure of the approximate solution $\hat{x}$ at each step to guide the selection of α is necessary. Note that scaling a rational problem to

be integral can create difficulties for numerical LU-factorization solvers, since some entries can become very large. This problem is avoided by performing the numerical LU factorization on the original unscaled form of the problem, then using the scaled integral matrix only in the refinement steps of the algorithm.

Iterative refinement and rational reconstruction is a very effective method for solving systems of equations exactly, often performing the fastest on our test set. The drawback to this method is its vulnerability to numerical difficulties with floating-point computations. We did experience some trouble on a small subset of examples that were numerically unstable; other methods considered in this paper do not share this problem.

3.4 Dixon's Method

Dixon's method [Wang 1981; Dixon 1982] for solving exact rational systems of equations relies on Theorem 3.2. This algorithm is stated in terms of integer systems of equations, so we first scale a rational system to be integer. In order to determine a solution modulo a large number M, Dixon [1982] uses the p-adic-lifting procedure, which constructs a solution modulo p^k by successively solving systems of equations modulo p. Algorithm 3 gives a description of p-adic lifting. This procedure can be thought of as computing a base p representation of x from the bottom up, one digit at a time.

Algorithm 3 p-adic Lifting

    Input: A, b, p, k $\{Ax = b$ is system to be solved mod $p^k\}$
    Set $x := 0, \quad d := b$
    for $i = 0, \ldots, k-1$ **do**
      $y := A^{-1}d \mod p$ $\{$Solve system$\}$
      $x := x + yp^i$ $\{$This will set $x = A^{-1}b \mod p^{i+1}\}$
      $d := \left(\frac{d - Ay}{p}\right)$ $\{$Compute residual of x in p-adic representation$\}$
    end for
    Return: x $\{x = A^{-1}b \mod p^k\}$

Dixon's method for solving integral systems of equations can use any subroutine to solve systems modulo a prime number p, and then apply p-adic lifting and rational reconstruction to determine the exact solution. In our implementation, we use QSLU_ffield for the finite-field solves. For a nonsingular integer matrix A, A mod p is nonsingular for a prime p if and only if p does not divide $det(A)$. Instead of computing $det(A)$ to guide the choice of p, we can guess different primes until an LU factorization is successful. The prime p is chosen small enough so that all numbers can be stored as machine-precision integers.

The p-adic-lifting procedure can be thought of as an analogue to the iterative-refinement method, since they both use fixed-precision solve routines iteratively to build extended-precision solutions. One advantage Dixon's method has over iterative refinement is that the finite-field elements are stored exactly, leaving no chance for numerical problems when performing calculations.

For further detail and analysis of the complexity of Dixon's method see [Dixon 1982; Mulders and Storjohann 1999; Chen 2005; Chen and Storjohann 2005].

3.5 Wiedemann's Method

Wiedemann's method for solving systems of equations over finite fields was introduced in [Wiedemann 1986]. His method can calculate the minimum polynomial of a matrix A, or of a sequence $\{A^i b\}_{i=0}^{\infty}$, over a finite field using a randomized algorithm based on the Berlekamp-Massey algorithm [Berlekamp 1967; Massey 1969]. This gives an explicit formula for solving $Ax = b$:

$$c_0 I + c_1 A + c_2 A^2 + \ldots c_m A^m = 0 \implies A^{-1} b = -c_0^{-1}(c_1 b + c_2 A b + \ldots c_m A^{m-1} b).$$

In both the computation and evaluation of the minimum polynomial, access to the matrix is only needed as a matrix-vector multiplication oracle. Therefore, Wiedemann's method is referred to as a black-box algorithm, and it is particularly suited for working with sparse matrices. A presentation of this technique is given in Section 12.4 of [von zur Gathen and Gerhard 2003].

Dixon's method for solving rational systems of equations relies on a finite-field solve routine for each p-adic-lifting step. Replacing QSLU_ffield with Wiedemann's method gives an alternative approach to solving systems of equations exactly. A detailed description of Wiedemann's method applied to solving rational systems of equations, including complexity analysis, is given in [Kaltofen and Saunders 1991].

4.   COMPUTATIONAL RESULTS

4.1   Implementation

We tested four methods in the C programming language to solve rational linear systems of equations: QSLU_rational, iterative refinement, Dixon, and Wiedemann. The rational-reconstruction routines used in the methods share a common structure, using the techniques detailed in Section 3.2. We implemented fast finite-field operations, storing the elements as integers, pre-computing inverse tables, using delayed modulus computation, and using floating-point operations to accelerate multiplications. These techniques are standard and they are employed by other software packages such as [Dumas et al. 2002; Shoup 2008].

When used, we attempted rational reconstruction with a frequency relative to the number of loops in the iterative refinement/p-adic-lifting procedure. In [Chen and Storjohann 2005], rational reconstruction is attempted every 10 loops. After some experimentation, we found it effective to attempt rational reconstruction with a geometric frequency; we choose specifically to attempt reconstruction when the loop number is a power of two. Using this strategy, if the approximate solution after k loops is sufficient to reconstruct the rational solution, then at most $log(k)$ unsuccessful attempts are made. We remark that if rational reconstruction is attempted before enough loops are completed to accurately reconstruct the solution, this can be recognized without reconstructing the entire vector, as mentioned in Section 3.2.3. Therefore, failed attempts at reconstruction will stop before reconstructing each component and thus have a considerably lower computational cost than the final reconstruction step in which each component is reconstructed. Despite this fact we still found it computationally faster to attempt reconstruction

less frequently. A more detailed analysis of our strategy for output-sensitive lifting can be found in [Steffy 2009]. We also modified the geometric strategy slightly by imposing a maximum number of loops between reconstruction attempts, giving improved solve times for our problem set. A similar strategy is also employed and analyzed by Chen and Monagan [2008] for solving linear systems over cyclotomic fields. A description and analysis of output-sensitive lifting applied to determinant computation can be found in [Kaltofen 2002]. A probabilistic output-sensitive lifting strategy is used in [Emiris 1998] where it is applied to the computation of integer hulls.

To verify the competitiveness of QSLU_double, which was used as a subroutine in iterative refinement, we compared it with the well-known solvers Pardiso [Schenk and Gärtner 2004] and SuperLU [Demmel et al. 1999]. In Table V, it can be seen that the QSLU_double code was faster on average over our testbed of LP instances, although Pardiso and SuperLU were faster on some individual problems. We may not expect QSLU_double to outperform Pardiso and SuperLU on other classes of instances, as it was developed using a method engineered specifically to solve very sparse bases arising in the solution of LP problems. In our use of QSLU_double, we perform two refinement steps in double precision to improve the double-precision solution. SuperLU contains a similar refinement scheme. We measured the backward relative error[3] of the final solutions and found SuperLU to produce more accurate solutions, with relative backward error average of 1.27e-16, compared to 1.58e-15 for QSLU_double. We found Pardiso to achieve comparable errors on many instances, but we experienced numerical difficulties on some examples, leading to unsatisfied constraints. (The performance of the Pardiso and SuperLU codes are compared with other numerical solvers in a nice computational study by [Gould et al. 2007], covering symmetric systems.)

Table V.   Numerical Sparse LU Solvers

Solver	Time Ratio
QSLU_double	1.00
SuperLU	2.36
Pardiso	2.50

We also compared our Wiedemann-based solver with the Wiedemann solver found in LinBox 1.1.6 [Dumas et al. 2002]. On the instances both codes completed, we found our new code to be 6.91 times faster by geometric mean, with ratios ranging from 0.13 to 17.30. We also compared our Wiedemann finite-field solver against the LinBox Wiedemann finite-field solver, which are used as subroutines in the rational solvers, and found our solver to be 1.84 times faster, with ratios ranging from 0.013 to 3.50 [4]. These results suggest that the speedup in our rational solver comes

---

[3]The backward relative error of a solution x is defined to be $\max_i \frac{|(Ax-b)_i|}{\sum_j |A_{ij}x_j|+|b_i|}$.

[4]A referee performed similar experiments on a smaller subset of our problems and observed our Wiedemann rational solver to be 2.72 times faster and our Wiedemann finite-field solver to be on average 62% slower on a 3.4Ghz Intel P4 computer.

Table VI.   Geometric Means of Relative Solve Times

Problems		Solver Time Ratios			
Subset of Probs.	Size of Subset	Dixon	Iter. Refine	QSLU_rational	Wied.
All Problems	276	1.0	**0.861**	3.247	38.370
Dim. 100-300	79	1.0	**0.859**	5.228	12.853
300-1,000	98	1.0	**0.835**	2.787	24.664
1,000-10,000	84	1.0	**0.889**	2.654	106.924
10,000+	15	1.0	**0.892**	2.219	703.087
Sol bitsize 0-100	92	1.0	**0.993**	10.744	36.892
100-1,000	79	1.0	**0.911**	4.069	57.008
1,000-10,000	55	1.0	**0.708**	0.971	39.443
10,000+	50	1.0	**0.751**	0.949	21.405
Nz./row 2-3	84	1.0	**0.966**	2.314	61.571
3-5	99	1.0	**0.814**	2.352	34.924
5-10	68	1.0	**0.825**	4.982	32.789
10+	25	1.0	**0.811**	11.342	17.433

from the rational-reconstruction techniques employed in our implementation. The purpose of this comparison is simply to demonstrate that our code is reasonably fast; LinBox is a much larger and more general software package written in C++, and our software was tuned to be as fast as possible on our specific problem set.

4.2   Rational System Results

Dixon's method, QSLU_rational, and Wiedemann's method were all able to solve all instances in our test set to completion. Iterative refinement was able to solve all but 5 of the problems, which failed for numerical reasons. Solve times varied greatly, ranging from fractions of a second to days. In Table VI we present a comparison of the solve times over all instances. Computations were performed on linux-based machines with 2.4GHz AMD Opteron 250 processors and 4 gigabytes of RAM. To avoid the slower instances outweighing all others, we normalized all solve times by dividing by the time for Dixon's method. We then computed the geometric means over the entire set of instances and also over selected subsets. Using the geometric mean instead of the arithmetic mean helps to prevent the results from being skewed by outliers. The five instances where iterative refinement failed are omitted from the averages in that column. The partitions of the problem set are based on the dimension of the instances, the bitsize of the final solutions, and the density of the instances, taken as the average number of nonzeros per row.

An immediate observation is that on the entire problem set, iterative refinement is the fastest method, followed by Dixon's method, which is nearly as fast. QSLU_rational is more than 3 times slower on average, and Wiedemann's method is on average nearly 40 times behind. By considering the various subsets of instances, we can identify patterns concerning the effect of problem characteristics on the solve times. One observation is that the dimension of the problem has the most significant relative effect on the Wiedemann method. The larger instances can have a minimum polynomial of a higher degree, requiring huge numbers of matrix-vector

multiplications to perform the finite-field solves.

We note that Dixon and iterative refinement have the best advantage over the QSLU_rational code on the instances with smaller solution bitsize; problems with a small solution bitsize can be computed with only a few steps of refinement/p-adic lifting. Relative to Dixon and iterative refinement, QSLU_rational becomes slower as the density increases, presumably because the LU factorization has more fill-in and computation, which is relatively more expensive using rational arithmetic. Wiedemann's method becomes relatively faster as the density increases; this is likely because increased density gives more work to the LU factorizations used by the other methods.

In Figure 2 we give a performance profile comparing the four methods on the full problem set. In this profile we can observe the close performance of Dixon and iterative refinement, the lag in speed of the QSLU_rational method, and the significantly slower speed of Wiedemann's method. We note the sharp edge in the curves for Dixon and iterative refinement, near the top where they cut far to the right quickly. This is caused by a small group of instances on which QSLU_rational is faster by a significant amount. Some the instances where QSLU_rational has a speed advantage are those having large solution bitsize; unfortunately the bitsize of a solution is not available before solving to aide in choosing a method.

Fig. 2.   Comparison of Rational Solvers
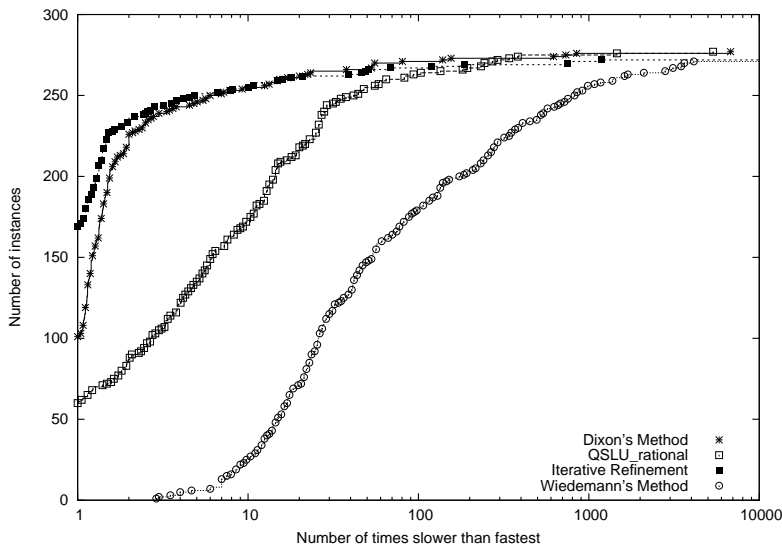
Another important observation we can make from our experiments is how each of the methods balances time between their internal subroutines. Table VII pro-

vides profiling data showing how the time was spent by each method, excluding QSLU_rational. This table was generated by considering the percent breakdown of the various stages of the algorithm by each method, for each instance, then the (arithmetic) average was taken over all instances. By *residual computation* we mean the time spent within each loop calculating the new right hand side for which the fixed precision solve will be computed and the radix conversion of the intermediate solution. Dixon and iterative refinement spent similar portions of their time on the LU factorizations and rational reconstruction, with a variation of time in their inner loops. The backsolves were faster for iterative refinement, but the computation of the residual for lifting was more expensive due to the additional work to compute the scaling factor α. We note that for Wiedemann's method, the largest portion of time was spent doing the first solve and then successive backsolves over finite fields; the large number of matrix-vector multiplications that must be done in these stages is relatively much slower than the LU factorization and solves. This table suggests that if Dixon's method and iterative refinement are to be made faster computationally, deeper investigation into accelerating rational reconstruction could be helpful, as it occupied nearly half of the solve time for these methods on average. The large portion of time spent on rational reconstruction may come as some surprise when one considers the worst-case complexity analysis of Dixon's method and iterative refinement, which do not have rational reconstruction as the dominating factor. We believe this is explained by the extreme sparsity of our matrices, which allows for faster than predicted computation in other portions of the algorithm, while not reducing the required computation for rational reconstruction.

Table VII. Profile of Time Spent

Solve Component	Dixon	Iter. Refine	Wiedemann
Factorization/First Solve	11.9 %	10.1 %	23.7 %
Backsolves	15.5 %	4.5 %	61.3 %
Residual Computation	25.6 %	39.8 %	7.7 %
Rational Reconstruction	47.0 %	45.6 %	7.3 %

Finally, Table VIII provides solve times and detailed information for select instances.

5. CONCLUSION

The results of our computational study provide a picture of how rational solver methods perform on a test set of very sparse real-world instances arising in linear-programming applications. By using variations of the QSopt factorization code and using common rational reconstruction strategies, we give a side by side comparison of these methods.

There are several conclusions we can make from our computations. The two methods we found to be the fastest were Dixon's method and iterative refinement. These two methods perform repeated fixed-precision solves to obtain high-accuracy solutions and apply rational reconstruction. Iterative refinement is approximately 15% faster overall, but Dixon's method has an advantage in numerical stability.

This agrees with the conclusion of Wan [Wan 2006] that his method is faster than Dixon's method on well-conditioned matrices and with the general knowledge that Dixon's method should be faster than a direct elimination method using rational arithmetic. Our conclusions also agree with a computational result in [Dumas and Villard 2002], who found that Wiedemann's method could be slower than direct elimination techniques on structured sparse matrices.

For such a speed difference we find Dixon's method to be the most attractive method for our application of exact-precision linear programming. An exact LP solver can call the exact linear system solver many times, making robustness very important. In other application areas, iterative refinement might be more attractive, especially if the systems are known to be numerically stable. We note also that in some exact LP solution schemes, a double-precision LU factorization of the basis matrix may be available at the end of a call to the simplex method. In such cases, the factorization can be used in the steps of the iterative-refinement method, resulting in a substantial savings in time. In our tests, the QSLU_rational code is faster than the other methods on a small subset of the instances. If multiple processors are available, a reasonable strategy is to run Dixon's method on one processor, and QSLU_rational on another.

We found Wiedemann's method to not be attractive for our LP test instances. Its black-box nature apparently does not make it competitive in this setting, as the LU factorizations for these very sparse problems can be computed very quickly. We believe that the QSLU codes benefited not only from the extreme sparsity of the matrices but from their structure as well. For other classes of sparse matrices for which LU factorizations are not possible without significant fill-in, we would expect Wiedemann's method to perform more competitively. In such cases dense solvers might also be competitive, especially on the problems with smaller dimension. In a study by Eberly et al. [2006] tests were performed on randomly generated sparse systems and it was found that an efficient dense solver was often able to beat their sparse solver unless the dimension was very large.

We have tried to use uniform standards as much as possible between our codes in order to give a fair comparison of the methods we are evaluating. However, there is always room for improvement in any implementation. We will make some comments on several improvements that could be made and how they could influence the results of our study. We thank the referees for some excellent suggestions in this light. The iterative solvers could benefit from better strategies for radix conversion, such as those used by Chen and Storjohann [2005] in their IML software; this strategy would take advantage of the asymptotically fast multiplication algorithms in GMP. Other gains for the iterative solves could come from implementing asymptotically faster strategies for rational number reconstruction such as the HalfGCD strategy outlined in [Lichtblau 2005]. For QSLU_rational a speedup could possibly be achieved by delaying canonicalization of the rational numbers, which might reduce the overall time spent on the frequent GCD computations that are made in association with arithmetic operations. Additional speedup in QSLU_rational might be achieved by applying additional effort in the LU factorization to maintain sparsity; more aggressive strategies than those currently used might pay off due to the high cost of rational arithmetic. If all of these suggestions were implemented

it would not effect the relative performance of the iterative solvers. We also conjecture that the iterative solvers could be improved more than the direct rational solver, which would not change our conclusions.

The methods we have used for rational reconstruction in this paper are also applicable to solving dense systems of equations using iterative refinement or Dixon's method. For dense systems of equations, both Wiedemann's method and direct rational solvers are not expected to perform competitively.

Acknowledgments

REFERENCES

ABBOTT, J., BRONSTEIN, M., AND MULDERS, T. 1999. Fast deterministic computation of determinants of dense matrices. In *ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA, 197–204.

ABBOTT, J. AND MULDERS, T. 2001. How tight is Hadamard's bound? *Experiment. Math. 10,* 3, 331–336.

ACHTERBERG, T., KOCH, T., AND MARTIN, A. 2006. MIPLIB 2003. *Operations Research Letters 34*, 361–372.

APPLEGATE, D., COOK, W., AND DASH, S. 2005. QSopt. http://www.isye.gatech.edu/~wcook/qsopt/.

APPLEGATE, D. L., BIXBY, R. E., CHVÁTAL, V., AND COOK, W. J. 2006. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, New Jersey, USA.

APPLEGATE, D. L., COOK, W., DASH, S., AND ESPINOZA, D. G. 2007. Exact solutions to linear programming problems. *Operations Research Letters 35*, 693–699.

BERLEKAMP, E. R. 1967. Factoring polynomials over finite fields. *Bell System Technical Journal 46*, 1853–1859.

BIXBY, R. E., CERIA, S., McZEAL, C. M., AND SAVELSBERGH, M. W. P. 1998. An updated mixed integer programming library: MIPLIB 3.0. *Optima 58*, 12–15.

CHEN, L. AND MONAGAN, M. 2008. Algorithms for solving linear systems over cyclotomic fields. *Submitted*.

CHEN, Z. 2005. A BLAS based C library for exact linear algebra over integer matrices. M.S. thesis, School of Computer Science, University of Waterloo.

CHEN, Z. AND STORJOHANN, A. 2005. A BLAS based C library for exact linear algebra on integer matrices. In *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*. ACM, New York, NY, USA, 92–99.

CHVÁTAL, V. 1983. *Linear Programming*. W. H. Freeman and Company, New York, NY, USA.

COHEN, H. 2000. *A Course in Computational Algebraic Number Theory*. Springer, New York, NY, USA.

COLLINS, G. E. AND ENCARNACIÓN, M. J. 1995. Efficient rational number reconstruction. *Journal of Symbolic Computation 20*, 287–297.

DEMMEL, J. W., EISENSTAT, S. C., GILBERT, J. R., LI, X. S., AND LIU, J. W. H. 1999. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications 20,* 3, 720–755.

DHIFLAOUI, M., FUNKE, S., KWAPPIK, C., MEHLHORN, K., SEEL, M., SCHÖMER, E., SCHULTE, R., AND WEBER, D. 2003. Certifying and repairing solutions to large LPs: How good are

LP-solvers? In *SODA '03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SIAM, Philadelphia, PA, USA, 255–256.

Dixon, J. D. 1982. Exact solution of linear equations using *p*-adic expansion. *Numerische Mathematik 40*, 137–141.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software 16,* 1, 1–17.

Dumas, J.-G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B. D., Turner, W. J., and Villard, G. 2002. LinBox: A generic library for exact linear algebra. In *Mathematical Software: ICMS 2002*, A. M. Cohen, X.-S. Gao, and N. Takayama, Eds. World Scientific, Singapore, 40–50.

Dumas, J.-G., Giorgi, P., and Pernet, C. 2008. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Transactions on Mathematical Software 35,* 3, Article 19.

Dumas, J.-G., Saunders, B. D., and Villard, G. 2001. On efficient sparse integer matrix smith normal form computations. *Jounal of Symbolic Computation 32*, 71–99.

Dumas, J.-G. and Villard, G. 2002. Computing the rank of large sparse matrices over finite fields. In *CASC'2002 : Computer Algebra in Scientific Computing.* 63–74.

Eberly, W., Giesbrecht, M., Giorgi, P., Storjohann, A., and Villard, G. 2006. Solving sparse rational linear systems. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation.* ACM, New York, NY, USA, 63–70.

Emiris, I. Z. 1998. A complete implementation for computing general dimensional convex hulls. *International Journal of Computational Geometry and Applications 8*, 223–253.

Espinoza, D. G. 2006. On linear programming, integer programming and cutting planes. Ph.D. thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology.

Gay, D. M. 1985. Electronic mail distribution of linear programming test problems. *COAL Newsletter 13*, 10–12.

GMP. 2008. GNU multiple precision arithmetic library, version 4.2. `http://gmplib.org`.

Golub, G. and van Loan, C. 1983. *Matrix Computations.* Johns Hopkins University Press, Baltimore, Maryland, USA.

Gould, N. I. M., Scott, J. A., and Hu, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software 33,* 2, Article 10.

Hales, T. C. 2005. A proof of the Kepler conjecture. *Annals of Mathematics 162*, 1065–1185.

ILOG. 2007. *User's Manual, ILOG CPLEX 11.0.* ILOG CPLEX Division, Incline Village, Nevada, USA.

Kaltofen, E. 2002. An output-sensitive variant of the baby steps/giant steps determinant algorithm. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation.* ACM, New York, NY, USA, 138–144.

Kaltofen, E. and Rolletschek, H. 1989. Computing greatest common divisors and factorizations in quadratic number fields. *Mathematics of Computation 53,* 188, 697–720.

Kaltofen, E. and Saunders, B. D. 1991. On Wiedemann's method of solving sparse linear systems. In *Proceedings of the Ninth International Symposium on Applied, Algebraic Algorithms, Error-Correcting Codes, Lecture Notes in Computer Science 539.* Springer, Heidelberg, Germany, 29–38.

Khodadad, S. and Monagan, M. 2006. Fast rational function reconstruction. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation.* ACM, New York, NY, USA, 184–190.

Koch, T. 2004. The final netlib-lp results. *Operations Research Letters 32*, 138–142.

Kwappik, C. 1998. Exact linear programming. M.S. thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software 5,* 3, 308–323.

LICHTBLAU, D. 2005. Half-gcd and fast rational recovery. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA, 231–236.

MASSEY, J. L. 1969. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory 15*, 122–127.

MÉSZÁROS, C. 2006. LPtestset. `http://www.sztaki.hu/~meszaros/public_ftp/lptestset/`.

MITTELMANN, H. D. 2006. LPtestset. `http://plato.asu.edu/ftp/lptestset/`.

MONAGAN, M. 2004. Maximal quotient rational reconstruction: an almost optimal algorithm for rational reconstruction. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA, 243–249.

MULDERS, T. AND STORJOHANN, A. 1999. Diophantine linear system solving. In *ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA, 181–188.

PAN, V. Y. AND WANG, X. 2003. Acceleration of Euclidean algorithm and rational number reconstruction. *SIAM Journal of Computing 32*, 548–556.

PAN, V. Y. AND WANG, X. 2004. On rational number reconstruction and approximation. *SIAM Journal of Computing 33*, 502–503.

REINELT, G. 1991. TSPLIB–a traveling salesman library. *ORSA Journal on Computing 3*, 376–384.

SCHENK, O. AND GÄRTNER, K. 2004. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems 3,* 20, 475–487.

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. Wiley, Chichester, UK.

SHOUP, V. 2008. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

STEFFY, D. 2009. Exact solutions to linear systems of equations using output sensitive lifting. `http://www.isye.gatech.edu/~dsteffy/`.

SUHL, U. H. AND SUHL, L. M. 1990. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing 2,* 4, 325–335.

URSIC, S. AND PATARRA, C. 1983. Exact solution of systems of linear equations with iterative methods. *SIAM Journal on Matrix Analysis and Applications 4,* 1, 111–115.

VANDERBEI, R. J. 2001. *Linear Programming: Foundations and Extensions*. Kluwer, Boston, Massachusetts, USA.

VON ZUR GATHEN, J. AND GERHARD, J. 2003. *Modern Computer Algebra*. Cambridge University Press, Cambridge, UK.

WAN, Z. 2006. An algorithm to solve integer linear systems exactly using numerical methods. *Journal of Symbolic Computation 41*, 621–632.

WANG, P. S. 1981. A p-adic algorithm for univariate partial fractions. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA, 212–217.

WIEDEMANN, D. H. 1986. Solving sparse linear equations over finite fields. *IEEE Trans. on Inf. Theory 32*, 54–62.

Table VIII. Details for specific instances

	Problem Characteristics			Solve Times in Seconds			
Problem Name	Dim	Nz/Row	Sol. Bitsize	Dixon	Iter. Refine	QSLU_rational	Wiedemann
brd14051	16360	11.05	1604	4.34	3.80	29.08	3245.80
cont11_l	58936	3.04	1263	7811.22	[failed]	1.15	1254009.51
fome13	24884	2.84	149	0.40	0.39	1.48	541.25
gen1	329	33.48	439931	1511.03	1395.16	77387.00	118338.65
gen2	328	27.11	20095	6.30	4.39	1706.02	41.39
gen4	375	23.78	22468	10.64	9.18	2673.08	150.09
jendrec1	1779	19.22	23452	8850.06	16908.90	14.27	1134895.15
maros-r7	1350	23.64	46915	42.45	30.17	21.80	1112.93
mod2	4435	2.92	42404	86.93	61.16	1.57	4894.21
momentum3	3254	4.65	159597	652.54	407.21	2978.85	17784.33
nemswrld	2205	6.04	16327	9.73	6.54	16.05	487.04
nug30	14681	3.10	1453	2.66	1.85	191.67	1397.15
pilot	1132	14.68	34247	24.29	[failed]	272.44	459.32
pilot4	289	9.70	79932	25.18	19.52	13.34	582.12
pilot87	1625	19.32	118607	395.61	316.16	13154.93	7235.02
pla33810	18940	6.51	270	0.51	0.48	2.63	822.83
pla85900.nov21	40304	5.72	2182	8.97	7.92	171.19	22234.55
progas	1167	5.56	102225	92.04	63.90	15.37	1798.70
rat5	902	13.33	28969	17.71	12.27	3337.09	211.51
self	924	170.35	46997	266.45	253.80	20970.46	3854.43
slptsk	2315	14.87	77394	213.14	215.33	421.01	7965.67
stat96v3	13485	3.70	3037	4.71	3.77	18.05	2569.22
stat96v4	3139	7.56	150292	618.96	449.77	2763.37	12924.12
stormg2_1000	14075	2.31	34	0.06	0.09	0.18	94.33
watson_1	5729	2.53	744	83.04	74.44	0.09	7165.69