Mathematical Programming Computation (2021) 13:509–551
https://doi.org/10.1007/s12532-020-00200-8

FULL LENGTH PAPER

Design and implementation of a modular interior-point solver for linear optimization

Mathieu Tanneau[1] · **Miguel F. Anjos**[2] · **Andrea Lodi**[1]

Received: 25 May 2019 / Accepted: 15 December 2020 / Published online: 8 February 2021

Abstract

This paper introduces the algorithmic design and implementation of Tulip, an open-source interior-point solver for linear optimization. It implements a regularized homogeneous interior-point algorithm with multiple centrality corrections, and therefore handles unbounded and infeasible problems. The solver is written in Julia, thus allowing for a flexible and efficient implementation: Tulip's algorithmic framework is fully disentangled from linear algebra implementations and from a model's arithmetic. In particular, this allows to seamlessly integrate specialized routines for structured problems. Extensive computational results are reported. We find that Tulip is competitive with open-source interior-point solvers on the H. Mittelmann's benchmark of barrier linear programming solvers. Furthermore, we design specialized linear algebra routines for structured master problems in the context of Dantzig–Wolfe decomposition. These routines yield a tenfold speedup on large and dense instances that arise in power systems operation and two-stage stochastic programming, thereby outperforming state-of-the-art commercial interior point method solvers. Finally, we illustrate Tulip's ability to use different levels of arithmetic precision by solving problems in extended precision.

Keywords Linear programming · Interior-point methods · Open-source software

Mathieu Tanneau was supported by an excellence doctoral scholarship from FQRNT.

✉ Andrea Lodi
andrea.lodi@polymtl.ca

Mathieu Tanneau
mathieu.tanneau@polymtl.ca

Miguel F. Anjos
anjos@stanfordalumni.org

[1] CERC, Polytechnique Montréal, Polytechnique de Montréal - Canada Excellence Research Chair, C.P. 6079, Succ. Centre-ville, Montréal, H3C 3A7 Québec, Canada

[2] School of Mathematics, University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD, United Kingdom

Mathematics Subject Classification 90C05 · 90C06 · 90C51

1 Introduction

Linear programming (LP) algorithms have been around for over 70 years, and LP remains a fundamental paradigm in optimization. Indeed, although nowadays most real-life applications involve discrete decisions or non-linearities, the methods employed to solve them often rely on LP as their workhorse. Besides algorithms for mixed-integer linear programming (MILP), these include cutting-plane and outer-approximation algorithms that substitute a non-linear problem with a sequence of iteratively refined LPs [40,47,57]. Furthermore, LP is at the heart of classical decomposition methods such as Dantzig–Wolfe and Benders decompositions [6,13]. Therefore, efficient and robust LP technology is instrumental to our ability to solve more involved optimization problems.

Over the past few decades, interior-point methods (IPMs) have become a standard and efficient tool for solving LPs [26,58]. While IPMs tend to overcome Dantzig's simplex algorithm on large-scale problems, the latter is well-suited for solving sequences of closely related LPs, by taking advantage of an advanced basis. Nevertheless, beyond sheer performance, it is now well recognized that a number of LP-based algorithms can further benefit from IPMs, despite their limited ability to warm start. In cutting plane algorithms, stronger cuts are often obtained by cutting off an interior point rather than an extreme vertex [9,47,48]. Similarly, IPMs have been successfully employed in the context of decomposition methods [5,19,28,50,51], wherein well-centered interior solutions typically provide a stabilization effect [27,31,53], thus reducing tailing-off and improving convergence.

1.1 Exploiting structure in IPMs

The remarkable performance of IPMs stems from both strong algorithmic foundations and efficient linear algebra. Indeed, the main computational effort of IPMs resides in the resolution, at each iteration, of a system of linear equations. Therefore, the efficiency of the underlying linear algebra has a direct impact of the method's overall performance. Remarkably, while most IPM solvers employ general-purpose sparse linear algebra routines, substantial speedups can be obtained by exploiting a problem's specific structure. Nevertheless, successfully doing so requires (i) identifying a problem's structure and associated specialized linear algebra, (ii) integrating these custom routines within an IPM solver, and (iii) having a convenient and flexible way for the user to convey structural information to the solver. The main contribution of our work is to simplify the latter two points.

Numerous works have studied structure-exploiting IPMs, e.g., [8,11,12,29,32,33, 37,39,54]. For instance, block-angular matrices typically arise in stochastic programming when using scenario decomposition. In [8] and later in [39], the authors thus design specialized factorization techniques that outperform generic implementations. Schultz et al. [54] design a specialized IPM for block-angular problems; therein, link-

ing constraints are handled separately, thus allowing to decompose the rest of the problem. Gondzio [33] observed that the master problem in Dantzig–Wolfe decomposition possesses a block-angular structure. Similar approaches have been explored for network flow problems [12], multi-commodity flow problems [32], asset management problems [29], and for solving facility location problems [11,37].

The aforementioned works focus on devising specialized linear algebra for a particular structure or application. On the other hand, a handful of IPM codes that accommodate various linear algebra implementations have been developed. The OOQP software, developed by Gertz and Wright [22], uses object-oriented design so that data structures and linear algebra routines can be tailored to specific applications. Motivated by large-scale stochastic programming, PIPS [43] incorporates a large share of OOQP's codebase, alongside specialized linear solvers for block-angular matrices. Nevertheless, to the best of the authors' knowledge, OOQP is no longer actively maintained, while current development on PIPS focuses on non-linear programming.[1] In a similar fashion, OOPS [29,30,32] implements custom linear algebra that can exploit arbitrary block matrix structures. We also note that both PIPS and OOPS are primarily intended for massive parallelism on high-performance computing infrastructure. Furthermore, the BlockIP software [10] is designed for block-angular convex optimization problems, and solves linear systems with a combination of Cholesky factorization and preconditioned conjugate gradient. Both OOPS and BlockIP can be accessed through SML [34]—which requires AMPL, and are distributed under a closed-source proprietary license.

Finally, while nowadays most optimization solvers are written in C or C++, users are increasingly turning to higher-level programming languages such as Python, Matlab or Julia, alongside a variety of modeling tools, e.g, Pyomo [36], CVXPY [16], YALMIP [42], JuMP [18], to mention a few. Thus, users of high-level languages often have to switch to a low-level language in order to implement performance-critical tasks such as linear algebra. This situation, commonly referred to as the "two-language problem", hinders code development, maintenance, and usability.

1.2 Contributions and outline

In this paper, we describe the design and implementation of a modular interior-point solver, Tulip. The solver is written in Julia [7], which offers several advantages. First, Julia combines both high-level syntax and fast performance, thus addressing the two-language problem. In particular, it offers built-in support for linear algebra, with direct access to dense and sparse linear algebra libraries such as BLAS, LAPACK and SuiteSparse [14]. Second, the Julia ecosystem for optimization comprises a broad range of tools, from solvers' wrappers to modeling languages, alongside a growing and dynamic community of users. Finally, Julia's multiple dispatch feature renders Tulip's design fully flexible, thus allowing to disentangle the IPM algorithmic framework from linear algebra implementations, and to solve problems in arbitrary precision arithmetic.

The remainder of the paper is structured as follows. In Sect. 2, we introduce some notations and relevant definitions.

[1] Personal communication with PIPS developers.

In Sect. 3, we describe the homogeneous self-dual embedding, and Tulip's regularized homogeneous interior-point algorithm. This feature contrasts with most IPM LP codes, namely, those that implement the almost-ubiquitous infeasible primal-dual interior-point algorithm [46]. The main advantage of the homogeneous algorithm is its ability to return certificates of primal or dual infeasibility. It is therefore better suited for use within cutting-plane algorithms or decomposition methods, wherein one may encounter infeasible or unbounded LPs.

In Sect. 4, we highlight the resolution of linear systems within Tulip, which builds on black-box linear solvers. This modular design leverages Julia's multiple dispatch, thereby facilitating the integration of custom linear algebra with no performance loss due to using external routines.

The presolve procedure is described in Sect. 5 and, in Sect. 6, we provide further implementation details of Tulip, such as the treatment of variable bounds, default values of parameters, and default linear solvers. Tulip is publicly available [55] under an open-source license. It can be used as a stand-alone package in Julia, and through the solver-independent interface `MathOptInterface` [41].

In Sect. 7, we report on three sets of computational experiments. First, we compare Tulip to several open-source and commercial IPM solvers on a benchmark set of unstructured LP instances. We observe that, using generic sparse linear algebra, Tulip is competitive with open-source IPM solvers. Second, we demonstrate Tulip's flexible design. We consider block-angular problems with dense linking constraints from two column-generation applications, for which we design specialized linear algebra routines. This implementation yields a tenfold speedup, thereby outperforming commercial solvers on large-scale instances. Third, we show how extended precision can alleviate numerical difficulties, thus illustrating Tulip's ability to work in arbitrary precision arithmetic.

Finally, Sect. 8 concludes the paper and highlights future research directions.

2 Notations

We consider LPs in primal-dual standard form

$$\begin{array}{lll} (P) & \min\limits_{x} & c^T x \\ & s.t. & Ax = b, \\ & & x \geq 0, \end{array} \qquad \begin{array}{lll} (D) & \max\limits_{y,s} & b^T y \\ & s.t. & A^T y + s = c, \\ & & s \geq 0, \end{array} \tag{1}$$

where $c, x, s \in \mathbb{R}^n$, $b, y \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m \times n}$ is assumed to have full row rank. We follow the usual notations from interior-point literature, and write X (resp. S) the diagonal matrix whose diagonal is given by x (resp. s), i.e., $X := Diag(x)$ and $S := Diag(s)$.

We denote I the identity matrix and e the vector with all coordinates equal to one; their respective dimensions are always obvious from context. The norm of a vector is written $\|\cdot\|$ and, unless specified otherwise, it denotes the ℓ_2 norm.

A primal solution x is feasible if $Ax = b$ and $x \geq 0$. A strictly feasible (or interior) solution is a primal feasible solution with $x > 0$. Similarly, a dual solution (y, s) is feasible if $A^T y + s = c$ and $s \geq 0$, and strictly feasible if, additionally, $s > 0$. Finally, a primal-dual solution (x, y, s) is optimal for (1) if x is primal-feasible, (y, s) is dual-feasible, and their objective values are equal, i.e., $c^T x = b^T y$.

A solution (x, y, s) with $x, s \geq 0$ is strictly complementary if

$$\forall i \in \{1, \ldots, n\}, \big(x_i s_i = 0 \text{ and } x_i + s_i > 0\big). \tag{2}$$

The complementarity gap is defined as $x^T s$. When (x, y, s) is primal-dual feasible, the complementarity gap equals the classical optimality gap, i.e., we have $x^T s = c^T x - b^T y$.

For ease of reading, we assume, without loss of generality, that all primal variables are required to be non-negative. The handling of free variables and of variables with finite upper bound will be detailed in Sect. 6.

3 Regularized homogeneous interior-point algorithm

In this section, we describe the homogeneous self-dual formulation and algorithm. Our implementation largely follows the algorithmic framework of [2,59], combined with the primal-dual regularization scheme of [21]. Consequently, we focus on the algorithm's main components, and refer to [2,21,59] for convergence proofs and theoretical results. Specific implementation details will be further discussed in Sect. 6.

3.1 Homogeneous self-dual embedding

The simplified homogeneous self-dual form was introduced in [59]. It consists in reformulating the primal-dual pair (1) as a single, self-dual linear program, which writes

$$\begin{aligned}
(HSD) \quad \min_{x,y,\tau} \quad & 0 && (3)\\
s.t. \quad & -A^T y + c\tau \geq 0, && (4)\\
& Ax - b\tau = 0, && (5)\\
& -c^T x + b^T y \geq 0, && (6)\\
& x, \tau \geq 0, && (7)
\end{aligned}$$

where τ is a scalar variable. Let s, κ be the non-negative slacks associated to (4) and (6), respectively. A solution (x, y, s, τ, κ) is strictly complementary if

$$x_i s_i = 0, x_i + s_i > 0, \text{ and } \tau\kappa = 0, \tau + \kappa > 0.$$

Problem (HSD) is always feasible, has empty interior and, under mild assumptions, possesses a strictly complementary feasible solution [59].

Let $(x^*, y^*, s^*, \tau^*, \kappa^*)$ be a strictly complementary feasible solution for (HSD). If $\tau^* > 0$, then $(\frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*})$ is an optimal solution for the original problem (1). Otherwise, we have $\kappa^* > 0$ and thus $c^T x^* - b^T y^* < 0$. In that case, the original problem (P) is infeasible or unbounded. If $c^T x^* < 0$, then (P) is unbounded and x^* is an unbounded ray. If $-b^T y^* < 0$, then (P) is infeasible and y^* is an unbounded dual ray. The latter is also referred to as a Farkas proof of infeasibility. Finally, if both $c^T x^* < 0$ and $-b^T y^* < 0$, then both (P) and (D) are infeasible.

3.2 Regularized formulation

Friedlander and Orban [21] introduce an exact primal-dual regularization scheme for convex quadratic programs, which we extend to the HSD form. The benefits of regularizations will be further detailed in Sect. 4. Importantly, rather than viewing (HSD) as a generic LP to which the regularization procedure of [21] is applied, we exploit the fact that (HSD) is a self-dual embedding of $(P)-(D)$, and formulate the regularization in the original primal-dual space.

Thus, we consider a *single*, regularized, self-dual problem

$$
\begin{aligned}
(rHSD) \quad \min_{x,y,\tau} \quad & \rho_p(x-\bar{x})^T x + \rho_d(y-\bar{y})^T y + \rho_g(\tau-\bar{\tau})\tau && (8)\\
s.t. \quad & -A^T y + c\tau + \rho_p(x-\bar{x}) \geq 0, && (9)\\
& Ax - b\tau + \rho_d(y-\bar{y}) = 0, && (10)\\
& -c^T x + b^T y + \rho_g(\tau-\bar{\tau}) \geq 0, && (11)\\
& x, \tau, \geq 0, && (12)
\end{aligned}
$$

where ρ_p, ρ_d, ρ_g are positive scalars, and $\bar{x} \in \mathbb{R}^n$, $\bar{y} \in \mathbb{R}^m$, $\bar{\tau} \in \mathbb{R}$ are given estimates of an optimal solution of (HSD). We denote by s, κ the non-negative slack variables of constraints (9) and (11), respectively. The first-order Karush–Kuhn–Tucker (KKT) conditions for $(rHSD)$ can then be expressed in the following form:

$$
\begin{aligned}
\rho_p x - A^T y - s + c\tau &= \rho_p \bar{x}, && && (13)\\
Ax + \rho_d y - b\tau &= \rho_d \bar{y}, && && (14)\\
-c^T x + b^T y + \rho_g \tau - \kappa &= \rho_g \bar{\tau}, && && (15)\\
x_j s_j &= 0, && j = 1, \ldots, n && (16)\\
\tau\kappa &= 0, && && (17)\\
x, s, \tau, \kappa &\geq 0. && && (18)
\end{aligned}
$$

The correspondence between $(rHSD)$ and [21] follows from the fact that, up to a constant term, the objective function (8) equals

$$
\frac{1}{2}\left(\rho_p \|x-\bar{x}\|^2 + \rho_d \|y-\bar{y}\|^2 + \rho_g \|\tau-\bar{\tau}\|^2 + \rho_p \|x\|^2 + \rho_d \|y\|^2 + \rho_g \|\tau\|^2\right).
$$

Note that, for $\rho_p = \rho_d = \rho_g = 0$, the regularized problem $(rHSD)$ reduces to (HSD). Furthermore, Theorem 1 shows that, for positive ρ_p, ρ_d, ρ_g, the regularization is exact.

Theorem 1 *Assume $\rho_p, \rho_d, \rho_g > 0$. Let (x^*, y^*, τ^*) be a complementary optimal solution of (HSD), and let $(\bar{x}, \bar{y}, \bar{\tau}) = (x^*, y^*, \tau^*)$ in the definition of $(rHSD)$. Then, (x^*, y^*, τ^*) is the unique optimal solution of $(rHSD)$.*

Proof The uniqueness of the optimum is a direct consequence of $(rHSD)$ being a convex problem with strictly convex objective.

Next, we show that any feasible solution of $(rHSD)$ has non-negative objective. Let (x, y, s, τ, κ) be a feasible solution of $(rHSD)$. Substituting Eqs. (9)–(11) into the objective (8), one obtains

$$\begin{aligned} Z &= \rho_p(x - \bar{x})^T x + \rho_d(y - \bar{y})^T y + \rho_g(\tau - \bar{\tau})\tau \\ &= (A^T y + s - c\tau)^T x + (b\tau - Ax)^T y + (c^T x - b^T y + \kappa)\tau \\ &= x^T s + \tau\kappa \geq 0. \end{aligned}$$

Then, (x^*, y^*, τ^*) is trivially feasible for $(rHSD)$, and its objective value is $(x^*)^T s^* + \tau^*\kappa^* = 0$. Thus, it is optimal for $(rHSD)$, which concludes the proof. □

3.3 Regularized homogeneous algorithm

We now describe the regularized homogeneous interior-point algorithm. Similar to [21], we apply a single Newton iteration to a sequence of problems of the form $(rHSD)$ where, at each iteration, $\bar{x}$, $\bar{y}$, $\bar{\tau}$ are chosen to be the current primal-dual iterate.

Let (x, y, s, τ, κ) denote the current primal-dual iterate, with $(x, s, \tau, \kappa) > 0$, and define the residuals

$$r_p = b\tau - Ax, \tag{19}$$

$$r_d = c\tau - A^T y - s, \tag{20}$$

$$r_g = c^T x - b^T y + \kappa, \tag{21}$$

and the barrier parameter

$$\mu = \frac{x^T s + \tau\kappa}{n + 1}.$$

For given $\bar{x}$, $\bar{y}$, $\bar{\tau}$, a search direction $(\delta_x, \delta_y, \delta_s, \delta_\tau, \delta_\kappa)$ is computed by solving a Newton system of the form

$$-\rho_p\delta_x + A^T\delta_y + \delta_s - c\delta_\tau = \eta\left(c\tau - A^T y - s + \rho_p(\bar{x} - x)\right), \tag{22}$$

$$A\delta_x + \rho_d\delta_y - b\delta_\tau = \eta\left(b\tau - Ax - \rho_d(y - \bar{y})\right), \tag{23}$$

$$-c^T\delta_x + b^T\delta_y + \rho_g\delta_\tau - \delta_\kappa = \eta\left(c^T x - b^T y + \kappa - \rho_g(\tau - \bar{\tau})\right), \tag{24}$$

$$S\delta_x + X\delta_s = -XSe + \gamma\mu e, \tag{25}$$

$$\kappa\delta_\tau + \tau\delta_\kappa = -\tau\kappa + \gamma\mu, \tag{26}$$

where γ and η are non-negative scalars whose values will be specified in Sect. 3.3.2. We evaluate the Newton system at $(\bar{x}, \bar{y}, \bar{\tau}) = (x, y, \tau)$, which yields

$$\begin{bmatrix} -\rho_p I & A^T & I & -c & 0 \\ A & \rho_d I & 0 & -b & 0 \\ -c^T & b^T & 0 & \rho_g & -1 \\ S & 0 & X & 0 & 0 \\ 0 & 0 & 0 & \kappa & \tau \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_s \\ \delta_\tau \\ \delta_\kappa \end{bmatrix} = \begin{bmatrix} \eta r_d \\ \eta r_p \\ \eta r_g \\ -XSe + \gamma\mu e \\ -\tau\kappa + \gamma\mu \end{bmatrix}. \tag{27}$$

System (27) is identical to the Newton system obtained when solving (HSD) (see, e.g., [2]), except for the regularization terms that appear in the left-hand side. In particular, the right-hand side remains unchanged.

3.3.1 Starting point

We choose the following default starting point

$$(x^0, y^0, s^0, \tau^0, \kappa^0) = (e, 0, e, 1, 1).$$

This initial point was proposed in [59]. Besides its simplicity, it has well-balanced complementarity products, which are all equal to one.

3.3.2 Search direction

At each iteration, a search direction is computed using Mehrotra's predictor-corrector technique [46], combined with Gondzio's multiple centrality corrections [24]. Following [2], we adapt the original formulas of [24,46] to account for the homogeneous embedding.

First, the affine-scaling direction $(\delta_x^{\text{aff}}, \delta_y^{\text{aff}}, \delta_s^{\text{aff}}, \delta_\tau^{\text{aff}}, \delta_\kappa^{\text{aff}})$ is obtained by solving the Newton system

$$-\rho_p\delta_x^{\text{aff}} + A^T\delta_y^{\text{aff}} + \delta_s^{\text{aff}} - c\delta_\tau^{\text{aff}} = r_d, \tag{28}$$

$$A\delta_x^{\text{aff}} + \rho_d\delta_y^{\text{aff}} - b\delta_\tau^{\text{aff}} = r_p, \tag{29}$$

$$-c^T\delta_x^{\text{aff}} + b^T\delta_y^{\text{aff}} + \rho_g\delta_\tau^{\text{aff}} - \delta_\kappa^{\text{aff}} = r_g, \tag{30}$$

$$S\delta_x^{\text{aff}} + X\delta_s^{\text{aff}} = -XSe, \tag{31}$$

$$\kappa\delta_\tau^{\text{aff}} + \tau\delta_\kappa^{\text{aff}} = -\tau\kappa, \tag{32}$$

which corresponds to (27) with $\eta = 1$ and $\gamma = 0$. Taking a full step ($\alpha = 1$) would thus reduce both infeasibility and complementarity gap to zero. However, doing so is generally not possible, due to the non-negativity requirement on (x, s, τ, κ).

Consequently, a corrected search direction is computed, as proposed in [46]. The corrected direction hopefully enables one to make longer steps, thus reducing the total number of IPM iterations. Let $\eta = 1 - \gamma$, where

$$\gamma = (1 - \alpha^{\text{aff}})^2 \min\left(\gamma_{min}, (1 - \alpha^{\text{aff}})\right) \tag{33}$$

for some $\gamma_{min} > 0$, and

$$\alpha^{\text{aff}} = \max\left\{0 \le \alpha \le 1 \mid (x, s, \tau, \kappa) + \alpha(\delta_x^{\text{aff}}, \delta_s^{\text{aff}}, \delta_\tau^{\text{aff}}, \delta_\kappa^{\text{aff}}) \ge 0\right\}. \tag{34}$$

The corrected search direction is then given by

$$-\rho_p \delta_x + A^T \delta_y + \delta_s - c\delta_\tau = \eta r_d, \tag{35}$$

$$A\delta_x + \rho_d \delta_y - b\delta_\tau = \eta r_p, \tag{36}$$

$$-c^T \delta_x + b^T \delta_y + \rho_g \delta_\tau - \delta_\kappa = \eta r_g, \tag{37}$$

$$S\delta_x + X\delta_s = -XSe + \gamma\mu e - \Delta_x^{\text{aff}} \Delta_s^{\text{aff}} e, \tag{38}$$

$$\kappa\delta_\tau + \tau\delta_\kappa = -\tau\kappa + \gamma\mu - \delta_\tau^{\text{aff}} \delta_\kappa^{\text{aff}}, \tag{39}$$

where $\Delta_x^{\text{aff}} = Diag(\delta_x^{\text{aff}})$ and $\Delta_s^{\text{aff}} = Diag(\delta_s^{\text{aff}})$.

3.3.3 Additional centrality corrections

Additional centrality corrections aim at improving the centrality of the new iterate, i.e., to keep the complementary products well balanced. Doing so generally allows to make longer steps, thus reducing the total number of IPM iterations. We implement Gondzio's original technique [24], with some modifications introduced in [2].

Let $\delta = (\delta_x, \delta_y, \delta_s, \delta_\tau, \delta_\kappa)$ be the current search direction, α^{max} the corresponding maximum step size, and define

$$(\bar{x}, \bar{y}, \bar{s}, \bar{\tau}, \bar{\kappa}) := (x, y, s, \tau, \kappa) + \bar{\alpha}(\delta_x, \delta_y, \delta_s, \delta_\tau, \delta_\kappa), \tag{40}$$

where $\bar{\alpha} := \min(1, 2\alpha^{max})$ is a tentative step size.

First, a soft target in the space of complementarity products is computed as

$$t_j = \begin{cases} \mu_l - \bar{x}_j \bar{s}_j & \text{if } \bar{x}_j \bar{s}_j < \mu_l \\ 0 & \text{if } \bar{x}_j \bar{s}_j \in [\mu_l, \mu_u] \\ \mu_u - \bar{x}_j \bar{s}_j & \text{if } \bar{x}_j \bar{s}_j > \mu_u \end{cases}, \quad j = 1, \ldots, n, \tag{41}$$

$$t_0 = \begin{cases} \mu_l - \bar{\tau}\bar{\kappa} & \text{if } \bar{\tau}\bar{\kappa} < \mu_l \\ 0 & \text{if } \bar{\tau}\bar{\kappa} \in [\mu_l, \mu_u] \\ \mu_u - \bar{\tau}\bar{\kappa} & \text{if } \bar{\tau}\bar{\kappa} > \mu_u \end{cases}, \tag{42}$$

where $\mu_l = \gamma\mu\beta$ and $\mu_u = \gamma\mu\beta^{-1}$, for a fixed $0 < \beta \leq 1$. Then, define

$$v = t - \frac{e^T t + t_0}{n+1} e, \tag{43}$$

$$v_0 = t_0 - \frac{e^T t + t_0}{n+1}. \tag{44}$$

A correction is obtained by solving the linear system

$$-\rho_p \delta_x^c + A^T \delta_y^c + \delta_s^c - c\delta_\tau^c = 0, \tag{45}$$

$$A\delta_x^c + \rho_d \delta_y^c - b\delta_\tau^c = 0, \tag{46}$$

$$-c^T \delta_x^c + b^T \delta_y^c + \rho_g \delta_\tau^c - \delta_\kappa^c = 0, \tag{47}$$

$$S\delta_x^c + X\delta_s^c = v, \tag{48}$$

$$\kappa\delta_\tau^c + \tau\delta_\kappa^c = v_0, \tag{49}$$

which yields a corrected search direction

$$(\delta_x, \delta_y, \delta_s, \delta_\tau, \delta_\kappa) + (\delta_x^c, \delta_y^c, \delta_s^c, \delta_\tau^c, \delta_\kappa^c).$$

The corrected direction is accepted if it results in an increased step size.

Finally, additional centrality corrections are computed only if a sufficient increase in the step size is observed. Specifically, as suggested in [2], an additional correction is computed only if the new step size α satisfies

$$\alpha \geq 1.10 \times \alpha^{\max}. \tag{50}$$

3.3.4 Regularizations

Following [21], the regularizations are updated as follows. Let ρ_p^k, ρ_d^k, ρ_g^k denote the regularization terms at iteration k. We set $\rho_p^0 = \rho_d^0 = \rho_g^0 = 1$, and use the update rule

$$\rho_p^{k+1} = \max\left(\sqrt{\epsilon}, \frac{\rho_p^k}{10}\right), \tag{51}$$

$$\rho_d^{k+1} = \max\left(\sqrt{\epsilon}, \frac{\rho_d^k}{10}\right), \tag{52}$$

$$\rho_g^{k+1} = \max\left(\sqrt{\epsilon}, \frac{\rho_g^k}{10}\right), \tag{53}$$

where ϵ denotes the machine precision, e.g., $\epsilon \simeq 10^{-16}$ for double-precision floating point arithmetic.

Further details on the role of regularizations in the resolution of the Newton system are given in Sect. 4. Let us only mention here that ρ_p, ρ_d, ρ_g may become too small to

ensure that the Newton system is properly regularized, e.g., for badly scaled problems. When this is the case, we increase the regularizations by a factor of 100, and terminate the algorithm if three consecutive increases fail to resolve the numerical issues.

3.3.5 Step size

Once the final search direction has been computed, the step size α is given by

$$\alpha = 0.9995 \times \alpha^{max}, \tag{54}$$

where

$$\alpha^{max} = \max \{0 \leq \alpha \leq 1 \mid (x, s, \tau, \kappa) + \alpha(\delta_x, \delta_s, \delta_\tau, \delta_\kappa) \geq 0\}.$$

3.3.6 Stopping criteria

The algorithm stops when, up to numerical tolerances, one of the following three cases holds: the current iterate is optimal, the primal problem is proven infeasible, the dual problem is proven infeasible (unbounded primal).

The problem is declared solved to optimality if

$$\frac{\|r_p\|_\infty}{\tau(1 + \|b\|_\infty)} < \varepsilon_p, \tag{55}$$

$$\frac{\|r_d\|_\infty}{\tau(1 + \|c\|_\infty)} < \varepsilon_d, \tag{56}$$

$$\frac{|c^T x - b^T y|}{\tau + |b^T y|} < \varepsilon_g, \tag{57}$$

where $\varepsilon_p, \varepsilon_d, \varepsilon_g$ are positive parameters. The above criteria are independent of the magnitude of τ, and correspond to primal feasibility, dual feasibility and optimality, respectively.

Primal or dual infeasibility is detected if

$$\mu < \varepsilon_i, \tag{58}$$

$$\frac{\tau}{\kappa} < \varepsilon_i, \tag{59}$$

where ε_i is a positive parameter. When this is the case, a complementary solution with small τ has been found. If $c^T x < -\varepsilon_i$, the problem is declared dual infeasible (primal unbounded), and x is an unbounded ray. If $-b^T y < -\varepsilon_i$, the problem is declared primal infeasible (dual unbounded), and y is a Farkas dual ray.

Finally, premature termination criteria such as numerical instability, time limit or iteration limit are discussed in Sect. 6.

4 Solving linear systems

Search directions and centrality corrections are obtained by solving several Newton systems such as (28)–(32), all with identical left-hand side matrix but different right-hand side. Specifically, each Newton system has the form

$$\begin{bmatrix} -\rho_p I & A^T & I & -c & \\ A & \rho_d I & & -b & \\ -c^T & b^T & & \rho_g & -1 \\ S & & X & & \\ & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_s \\ \delta_\tau \\ \delta_\kappa \end{bmatrix} = \begin{bmatrix} \xi_d \\ \xi_p \\ \xi_g \\ \xi_{xs} \\ \xi_{\tau\kappa} \end{bmatrix}, \tag{60}$$

where $\xi_p, \xi_d, \xi_g, \xi_{xs}, \xi_{\tau\kappa}$ are appropriate right-hand side vectors. The purpose of this section is to provide further details on the techniques used for the resolution of (60), and their implementation in Tulip.

4.1 Augmented system

First, we eliminate δ_s and δ_κ as follows:

$$\delta_s = X^{-1}(\xi_{xs} - S\delta_x), \tag{61}$$

$$\delta_\kappa = \tau^{-1}(\xi_{\tau\kappa} - \kappa\delta_\tau), \tag{62}$$

which yields

$$\begin{bmatrix} -(\Theta^{-1} + \rho_p I) & A^T & -c \\ A & \rho_d I & -b \\ -c^T & b^T & \tau^{-1}\kappa + \rho_g \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_\tau \end{bmatrix} = \begin{bmatrix} \xi_d - X^{-1}\xi_{xs} \\ \xi_p \\ \xi_g + \tau^{-1}\xi_{\tau\kappa} \end{bmatrix}, \tag{63}$$

where $\Theta = XS^{-1}$.

As outlined in [2,58], a solution to (63) is obtained by first solving

$$\begin{bmatrix} -(\Theta^{-1} + \rho_p I) & A^T \\ A & \rho_d I \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} c \\ b \end{bmatrix}, \tag{64}$$

and

$$\begin{bmatrix} -(\Theta^{-1} + \rho_p I) & A^T \\ A & \rho_d I \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \xi_d - X^{-1}\xi_{xs} \\ \xi_p \end{bmatrix}. \tag{65}$$

Linear systems of the form (64) and (65) are referred to as *augmented systems*. Then, $\delta_x, \delta_y, \delta_\tau$ are computed as follows:

$$\delta_\tau = \frac{\xi_g + \tau^{-1}\xi_{\tau\kappa} + c^T u + b^T v}{\tau^{-1}\kappa + \rho_g} - c^T p + b^T q, \tag{66}$$

$$\delta_x = u + \delta_\tau p, \tag{67}$$
$$\delta_y = v + \delta_\tau q. \tag{68}$$

Note that (64) does not depend on the right-hand side ξ. Thus, it is only solved once per IPM iteration, and its solution is reused when solving subsequent Newton systems.

Finally, as pointed in [21], the augmented system's structure motivates the following observations. First, the use of primal-dual regularizations controls the effective condition number of the augmented system, which, in turn, improves the algorithm's numerical behavior. Second, the augmented system's matrix is symmetric quasi-definite. This allows the use of efficient symmetric indefinite factorization techniques, which only require one symbolic analysis at the beginning of the optimization. In particular, dual regularizations ensure that this quasi-definite property is retained even when A does not have full rank. Third, directly solving the augmented system implicitly handles dense columns in A, which make the system of normal equations dense [58]. We have also found this approach to be more numerically stable than a normal equations system-based approach.

4.2 Black-box linear solvers

The augmented system may be solved using a number of techniques, with direct methods—namely, symmetric factorization techniques—being the most popular choice. Importantly, the algorithm itself is unaffected by *how* the augmented system is solved, provided that it is solved accurately. Our implementation leverages Julia's multiple dispatch feature and built-in support for linear algebra, thus allowing to disentangle the algorithmic framework from the linear algebra implementation.

First, the interior-point algorithm is defined over abstract linear algebra structures. Namely, the constraint matrix A is treated as an `AbstractMatrix`, whose concrete type is only known once the model is instantiated. Julia's standard library includes extensive support for linear algebra, thus removing the need for a custom abstract linear algebra layer.

Second, while the reduction from the Newton system to the augmented system is performed explicitly, the latter is solved by a black-box linear solver. Specifically, we design an `AbstractKKTSolver` type, from which concrete linear solver implementations inherit. The `AbstractKKTSolver` interface is deliberately minimal, and consists of three functions:[2] `setup`, `update!`, and `solve!`.

A linear solver is instantiated at the beginning of the optimization using the `setup` function. Custom options can be passed to `setup` so that the user can select a linear solver of their choice. At the beginning of each IPM iteration, the linear solver's state is updated by calling the `update!` function. For instance, if a direct method is used, this step corresponds to updating the factorization. Following the call to `update!`, augmented systems can be solved through the `solve!` function. Default, generic, linear solvers are described in Sect. 6.3, and an example of specialized linear solver is given in Sect. 7.2. Specific details are provided in Tulip's online documentation.[3]

[2] In Julia, a `!` is appended to functions that mutate their arguments.

[3] https://ds4dm.github.io/Tulip.jl/dev/.

Finally, specialized methods are automatically dispatched based on the (dynamic) type of A. These include matrix-vector and matrix-matrix product, as well as matrix factorization routines. We emphasize that the dispatch feature is a core component of the Julia programming language, and is therefore entirely transparent to the user. Consequently, one can easily define custom routines that exploit certain properties of A, so as to speed-up computation or reduce memory overheads. Furthermore, this customization is entirely independent of the interior-point algorithm, thus allowing to properly assess the impact of different linear algebra implementations.

5 Presolve

Tulip's presolve module performs elementary reductions, all of which are described in [1,25]. Therefore, in this section, we only outline the presolve procedure; further implementation details are given in Sect. 6.

5.1 Presolve

We only perform reductions that do not introduce any additional non-zero coefficients, i.e., fill-in, to the problem. The presolve procedure is outlined in Algorithm 1, and proceeds as follows.

First, we ensure all bounds are consistent, remove all empty rows and columns, and identify all row singletons, i.e., rows that contain a single non-zero coefficient. Then, a series of passes is performed until no further reduction is possible. At each pass, the following reductions are applied: empty rows and columns, fixed variables, row singletons, free and implied free column singletons, forcing and dominated rows, and dominated columns. The presolve terminates if infeasibility or unboundedness is detected, in which case an appropriate primal or dual ray is constructed. If all rows and columns are eliminated, the problem is declared solved, and a primal-dual optimal solution is constructed.

Finally, to improve the numerical properties of the problem, rows and columns are re-scaled as follows:

$$\tilde{A} = D^{(r)} \times A \times D^{(c)}, \tag{69}$$

where $\tilde{A}$ is the scaled matrix, A is the constraint matrix of the reduced problem, and $D^{(r)}$, $D^{(c)}$ are diagonal matrices with coefficients

$$D_i^{(r)} = \frac{1}{\sqrt{\|A_{i,\cdot}\|}}, \quad \forall i, \tag{70}$$

$$D_j^{(c)} = \frac{1}{\sqrt{\|A_{\cdot,j}\|}}, \quad \forall j. \tag{71}$$

Column and row bounds, as well as the objective, are scaled appropriately.

Algorithm 1 Presolve procedure

Input: Initial LP

Remove empty rows
Remove empty columns

repeat

Check for bounds inconsistencies
Remove empty columns

Remove row singletons
Remove fixed variables

Remove row singletons
Remove forcing/dominated rows

Remove row singletons
Remove free columns singletons

Remove row singletons
Remove dominated columns

until No reduction is found

Scale rows and columns

5.2 Postsolve

A primal-dual solution to the presolved problem is computed using the interior-point algorithm described in Sect. 3. A solution to the original problem is then constructed in a postsolve phase, which is described in [1,25]. Note that, in general, the postsolve solution is *not* an interior point with respect to the original problem, e.g., some variables may be at their upper or lower bound.

6 Implementation details

Tulip is an officially registered Julia package, and is publicly available[4] under an open-source license. The entire source code comprises just over 4, 000 lines of Julia code, which makes it easy to read and to modify. The code is single-threaded, however external linear algebra libraries may exploit multiple threads.

We provide an interface to `MathOptInterface` [41], a solver-agnostic abstraction layer for optimization. Thus, Tulip is readily available through both `JuMP` [18], an open-source algebraic modeling language embedded in Julia, and the convex optimization modeling framework `Convex` [56].

Finally, Tulip supports arbitrary precision arithmetic, thus allowing, for instance, to solve problems in quadruple (128 bits) precision. This functionality is available from Tulip's direct API and through the `MathOptInterface` API; it is illustrated in Sect. 7.3.

[4] Source code is available at https://github.com/ds4dm/Tulip.jl, and online documentation at https://ds4dm.github.io/Tulip.jl/dev/.

6.1 Bounds on variables

Tulip stores LP problems in the form

$$\begin{array}{lll} (LP) & \min\limits_{x} & c^T x \quad + c_0 \\ & s.t. & l_i^b \leq \sum_j a_{i,j} x_j \leq u_i^b, \quad \forall i = 1, \dots, m, \\ & & l_j^x \leq \quad x_j \quad \leq u_j^x, \quad \forall j = 1, \dots, n, \end{array} \tag{72}$$

where $l_{i,j}^{b,x}, u_{i,j}^{b,x} \in \mathbb{R} \cup \{-\infty, +\infty\}$, i.e., some bounds may be infinite. Before being passed to the interior-point optimizer, the problem is transformed into standard form. This transformation occurs after the presolve phase, and is transparent to the user. In particular, primal-dual solutions are returned with respect to formulation (72).

Free variables are an outstanding issue for interior-point methods, see, e.g. [3,58], and are not supported explicitly in Tulip. Instead, free variables are automatically split into the difference of two non-negative variables, with the knowledge that this reformulation may introduce some numerical instability.

Although finite upper bounds may be treated as arbitrary constraints, it is more efficient to handle them separately. Let $\mathcal{I}$ denote the set of indices of upper-bounded variables. Upper-bound constraints then write

$$x_i \leq u_i, \quad \forall i \in \mathcal{I}, \tag{73}$$

which we write in compact form $Ux \leq u$, where $U \in \mathbb{R}^{|\mathcal{I}| \times n}$ and

$$U_{i,j} = \begin{cases} 1 & \text{if } i = j \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases}.$$

Therefore, internally, Tulip solves linear programs of the form

$$\begin{array}{lll} (P) & \min\limits_{x,w} & c^T x \\ & s.t. & Ax = b, \\ & & Ux + w = u \\ & & x, w \geq 0, \end{array} \qquad \begin{array}{lll} (D) & \max\limits_{y,s,z} & b^T y - u^T z \\ & s.t. & A^T y + s - U^T z = c, \\ & & s, z \geq 0. \end{array} \tag{74}$$

Let us emphasize that handling upper bounds separately only affects the underlying linear algebra operations, not the interior-point algorithm.

The Newton system (60) then writes

$$\begin{bmatrix} -\rho_p I & A^T & I & -U^T & -c & \\ A & \rho_d I & & & -b & \\ U & I & & & -u & \\ -c^T & b^T & & -u^T & \rho_g & -1 \\ S & & X & & & \\ & Z & & W & & \\ & & & & \kappa & \tau \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_w \\ \delta_y \\ \delta_s \\ \delta_z \\ \delta_\tau \\ \delta_\kappa \end{bmatrix} = \begin{bmatrix} \xi_d \\ \xi_p \\ \xi_u \\ \xi_g \\ \xi_{xs} \\ \xi_{wz} \\ \xi_{\tau\kappa} \end{bmatrix}, \tag{75}$$

and it reduces, after performing diagonal substitutions, to solving two augmented systems of the form

$$\begin{bmatrix} -(\tilde{\Theta}^{-1} + \rho_p I) & A^T \\ A & \rho_d I \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} \tilde{\xi}_d \\ \tilde{\xi}_p \end{bmatrix}, \tag{76}$$

where $\tilde{\Theta} = \left(X^{-1}S + U^T(W^{-1}Z)U\right)^{-1}$. Note that $\tilde{\Theta}$ is a diagonal matrix with positive diagonal. Therefore, system (76) has the same size and structure as (64). Furthermore, $\tilde{\Theta}$ can be computed efficiently using only vector operations, i.e., without any matrix-matrix nor matrix-vector product.

6.2 Solver parameters

The default values for numerical tolerances of Sect. 3.3.6 are

$$\begin{aligned} \varepsilon_p &= \sqrt{\epsilon}, \\ \varepsilon_d &= \sqrt{\epsilon}, \\ \varepsilon_g &= \sqrt{\epsilon}, \\ \varepsilon_i &= \sqrt{\epsilon}, \end{aligned}$$

where ϵ is the machine precision, which depends on the arithmetic. For instance, double precision (64 bits) floating point arithmetic corresponds to $\epsilon_{64} \simeq 10^{-16}$, while quadruple precision (128 bits) corresponds to $\epsilon_{128} \simeq 10^{-34}$.

When computing additional centrality corrections, we use the following default values:

$$\begin{aligned} \gamma_{min} &= 10^{-1}, \\ \beta &= 10^{-1}. \end{aligned}$$

The default maximum number of centrality corrections is set to 5.

Finally, the maximum number of IPM iterations is set to a default of 100. A time limit may be imposed by the user, in which case it is checked at the beginning of each IPM iteration.

6.3 Default linear solvers

Several generic linear algebra implementations are readily available in Tulip, and can be selected without requiring any additional implementation.

The default settings are as follows. First, A is stored in a `SparseMatrixCSC` struct, i.e., in compressed sparse column format. Elementary linear algebra operations, e.g., matrix-vector products, employ Julia's standard library. Augmented systems are then solved by a direct method, namely, an LDL^T factorization of the quasi-definite augmented system. Sparse factorizations use either the CHOLMOD module

of SuiteSparse [14], or the `LDLFactorizations` package [52], a Julia translation of SuiteSparse's LDL^T factorization code that supports arbitrary arithmetic. Tulip uses the former for double precision floating point arithmetic, and the latter otherwise. Finally, the solver's log indicates: the model's arithmetic, the linear solver's backend, e.g., CHOLMOD, and the linear system being solved, i.e., either the augmented system of the normal equations system.

As mentioned in Sect. 4, custom options for linear algebra can be passed to the solver. Specifically, the `MatrixOptions` parameter lets the user select a matrix implementation of their choice, and the `KKTOptions` parameter is used to specify a choice of linear solver. Their usage is depicted in Fig. 1.

In Fig. 1a, the default settings are used. The model is instantiated at line 3; the `Model{Float64}` syntax indicates that `Float64` arithmetic is used. Then, the problem is read from the `problem.mps` file at line 4, and the model is solved at line 6. Figure 1b–d are identical, but select different linear algebra implementations by setting the appropriate `MatrixOptions` and `KKTOptions` parameters.

Figure 1b illustrates the use of dense linear algebra. Line 7 indicates that A should be stored as a dense matrix. Then, at line 8, a dense linear solver is selected through the `SolverOptions(DenseSPD)` setting. In this case, the augmented system is reduced to the (dense) normal equations systems, and a dense Cholesky factorization is applied; BLAS/LAPACK routines are automatically called when using single and double precision floating point arithmetic, otherwise Julia's generic routines are called.

In the example of Fig. 1c, linear systems are reduced to the normal equations system, and CHOLMOD's sparse Cholesky factorization is applied. Note that a single dense column in A results in a fully dense normal equations systems. Thus, in the absence of a mechanism for handling dense columns, this approach may be impractical for some large problems. Finally, in Fig. 1d, the augmented system is solved using an LDL^T factorization, computed by `LDLFactorizations`.

7 Computational results

In this section, we compare Tulip to several open-source and commercial solvers, focusing on those that are available to Julia users. Let us emphasize that our goal is *not* to perform a comprehensive benchmark of interior-point LP solvers.

We evaluate Tulip's performance and robustness in the following three settings. First, in Sect. 7.1, we consider general LP instances from H. Mittelmann's benchmark,[5] which are solved using generic sparse linear algebra. Then, in Sect. 7.2, we consider structured instances that arise in decomposition methods, for which we develop specialized linear algebra. Finally, in Sect. 7.3, we illustrate Tulip's ability to use different levels of arithmetic precision by solving problems in higher precision.

[5] http://plato.asu.edu/ftp/lpbar.html.

```
import Tulip

model = Tulip.Model{Float64}()              # Instantiate model
Tulip.load_problem!(model, "problem.mps")   # Read problem

Tulip.optimize!(model)                      # Solve the problem
```

(a) Sample code using default linear algebra settings

```
import Tulip

model = Tulip.Model{Float64}()              # Instantiate model
Tulip.load_problem!(model, "problem.mps")   # Read problem

# Select dense linear algebra
model.params.MatrixOptions = MatrixOptions(Matrix)
model.params.KKTOptions = SolverOptions(DenseSPD)

Tulip.optimize!(model)                      # Solve the problem
```

(b) Sample code using dense linear algebra

```
import Tulip

model = Tulip.Model{Float64}()              # Instantiate model
Tulip.load_problem!(model, "problem.mps")   # Read problem

# Solve the normal equations with CHOLMOD
model.params.KKTOptions = SolverOptions(CholmodSPD)

Tulip.optimize!(model)                      # Solve the problem
```

(c) Sample code using CHOLMOD to solve the normal equations system

```
import Tulip

model = Tulip.Model{Float64}()              # Instantiate model
Tulip.load_problem!(model, "problem.mps")   # Read problem

# Solve the augmented system with LDLFactorizations
model.params.KKTOptions = SolverOptions(LDLFactSQD)

Tulip.optimize!(model)                      # Solve the problem
```

(d) Sample code using `LDLFactorizations`

Fig. 1 Code examples for reading and solving a problem with various linear algebra implementations

7.1 Results on general LP instances

We select all instances from H. Mittelmann's benchmark of barrier LP solvers, except `qap15` and `L1_sixm1000obs`. The former is identical to `nug15`, and the latter could not be solved by any solvers in the prescribed time limit. This yields a testset

of 43 medium to large-scale instances. We compare the following open-source and commercial solvers: Clp 1.17 [20], GLPK 4.64 [45], ECOS 2.0[17], Tulip 0.5.0 [55], CPLEX 12.10 [38], Gurobi 9.0 [35] and Mosek 9.2 [49]. All are accessed through their respective Julia interface. We run the interior-point algorithm of each solver with a single thread, no crossover, and a 10, 000 s time limit. For Tulip, the maximum number of IPM iterations is increased from the default 100 to 500. All other parameters are left to their default values.

Experiments are carried out on a cluster of machines equipped with dual Intel Xeon 6148–2.4 GHz CPUs, and varying amounts of RAM. Each job is run with a single thread and 16GB of memory. Scripts for running these experiments are available online,[6] together with the logfiles of each solver.

Computational results are displayed in Table 1. For each solver, we report the total number of instances solved, the mean runtime, and individual runtimes for each instance. Segmentation faults are indicated by `seg`, timeouts by `t`, other failures by `f`, and reduced accuracy solutions by `r`. The time to read in the data is not included. Mean runtimes are shifted geometric means

$$\mu_\delta(t_1, \ldots, t_N) = \left(\prod_{i=1}^{N}(t_i + \delta)\right)^{\frac{1}{N}} - \delta = \exp\left[\frac{1}{N}\sum_{i=1}^{N}\log(t_i + \delta)\right] - \delta,$$

with $\delta = 10\,\text{s}$.

First, the three commercial solvers CPLEX, Gurobi and Mosek display similar performance and robustness, and outperform open-source alternatives by one to two orders of magnitude. While CPLEX and Gurobi encountered numerical issues on a few instances, we found that these were resolved by activating crossover.

Second, Clp displays a worse performance than expected, solving only 25 problems with an average runtime about two times larger than Tulip's. In fact, out of 43 instances, we recorded 5 segmentation faults, 8 unidentified errors, with the 10, 000 s time limit being reached on the remaining 10 unsolved instances. A more detailed analysis of the log suggests that segmentation faults and some unknown errors are caused by memory-related issues, i.e., large Cholesky factors that do not fit in memory. We note that those errors do not occur when running Clp through its command-line executable: the executable performs additional checks to decide whether the model should be dualized; this can yield smaller linear systems and thus avoid memory issues. Nevertheless, given that the `dualize` option is not available in Clp's C interface[7], on which Clp's Julia wrapper is built, the present results best represent the behavior that Julia users would encounter.

Third, among open-source solvers, Tulip is the top performer with 33 instances solved and a mean runtime of 604.6s, while GLPK has the worst performance with only 6 instances reportedly solved. Tulip's 5 failures include 3 instances that ran out of memory; for the remaining 2, i.e., `ns1688926` and `watson_2`, Tulip fails to reach the prescribed accuracy due to numerical issues. A possible remedy to the latter will

[6] https://github.com/mtanneau/LPBenchmarks.

[7] See discussion in https://github.com/coin-or/Clp/issues/151.

Table 1 Results on the Mittelmann test set

Problem	Clp	CPLEX	ECOS	GLPK	Gurobi	Mosek	Tulip
Solved	25	39	26	6	41	43	33
Average	1607.5	52.4	2344.7	7092.8	42.5	32.0	604.6
L1_sixm250obs	seg	23.7	f	f	f	148.8	f
Linf_520c	seg	25.9	f	seg	30.4	22.7	t
brazil3	2.3	0.2	f	f	0.6	0.6	2.3
Buildingenergy	f	18.3	362.0	f	19.9	16.6	39.1
chrom1024-7	seg	0.3	96.8	f	1.5	3.2	3.6
cont1	2322.4	5.2	138.0	f	5.9	12.6	26.9
cont11	626.8	f	174.7	f	14.8	12.7	51.4
dbic1	118.5	9.6	332.9	f	9.2	9.4	30.2
Degme	t	235.5	f	f	308.3	253.0	t
ds-big	237.1	29.6	331.6r	f	31.1	16.5	95.2
ex10	t	6.3	f	f	46.8	21.6	5427.5
fome13	413.3	19.0	1284.0	f	17.8	16.9	538.2
Irish-e	363.6	21.1	f	f	16.5	20.3	35.4
Karted	6509.1	89.1	5801.8	9334.9	115.7	44.4	3786.9
neos	433.4	26.7	1201.0r	seg	39.5	33.1	419.5
neos1	f	4.9	167.2	f	6.3	4.1	130.4
neos2	f	4.0	129.6	f	4.7	3.4	462.1
neos3	seg	26.5	1282.9	f	33.7	17.1	1358.2
neos5052403	2067.9	43.6	3489.4r	f	28.1	13.1	475.4
ns1644855	t	334.2	f	f	437.0	468.0	t
ns1687037	t	f	f	f	19.6	12.3	330.4
ns1688926	t	25.7	f	f	f	2.0	f
nug08-3rd	t	3.4	f	f	2.8	55.0	f
nug15	352.2	12.7	1871.4	f	0.9	17.8	998.6
pds-100	t	168.2	f	f	106.2	152.7	f
pds-40	1303.7	25.6	f	f	22.1	32.9	5000.6
psched3-3	t	86.6	f	f	147.5r	148.5	t
rail02	t	208.9	f	f	134.3	195.8	t
rail4284	5407.8	72.2	8578.3r	f	133.5	81.0	1049.5
s100	1587.5	29.7	f	f	38.1	30.2	894.2
s250r10	263.5	17.8	f	f	25.4	30.8	257.2
savsched1	183.9	27.1	2355.4	f	25.9	55.5	138.8
Self	21.5	3.2	162.3	45.1	4.0	3.1	13.3

Table 1 continued

Problem	Clp	CPLEX	ECOS	GLPK	Gurobi	Mosek	Tulip
shs1023	286.2	f	f	f	48.2	74.6	371.6
square41	202.8	3.4	1703.3r	f	4.9	32.7	134.0
stat96v1	164.9	f	163.6r	f	32.2r	6.3	41.3
stat96v4	1.4	1.0	31.4	261.5	1.0	2.2	1.8
stormG2_1000	seg	64.2	9593.0	f	122.6	131.9	216.3
stp3d	1864.1	31.6	1363.3	f	31.9	39.2	529.7
support10	f	17.1	6210.2	f	19.3	27.2	3553.1
tp-6	7872.1	150.9	f	f	203.8	312.3	5543.0
ts-palko	1757.1	35.9	1109.3	2315.7	50.7	31.7	841.1
watson_2	65.0	24.4	f	111.0	26.6	30.1	f

seg segmentation fault, r reduced accuracy solution, t time limit, f other failure
All times in seconds

be discussed in Sect. 7.3. Finally, out of the 26 instances reported as solved by ECOS, 6 were solved to reduced accuracy. This situation typically corresponds to ECOS encountering numerical issues close to optimality, but a feasible or close-to-feasible solution is still available.

7.2 Results on structured LP instances

We now compare Tulip to state-of-the-art commercial solvers on a collection of structured problems, for which we design specialized linear algebra routines. Specifically, we consider the context of Dantzig–Wolfe (DW) decomposition [13] in conjunction with a column-generation (CG) algorithm; we refer to [15] for a thorough overview of DW decomposition and CG algorithms. Here, we focus on the resolution of the master problem, i.e., we consider problems of the form

$$(MP) \quad \min_{\lambda} \quad \sum_{r=1}^{R} \sum_{j=1}^{n_r} c_{r,j} \lambda_{r,j} + c_0^T \lambda_0 \tag{77}$$

$$s.t. \quad \sum_{j=1}^{n_r} \lambda_{r,j} = 1, \quad r = 1, \dots, R, \tag{78}$$

$$\sum_{r=1}^{R} \sum_{j=1}^{n_r} a_{r,j} \lambda_{r,j} + A_0 \lambda_0 = b_0, \tag{79}$$

$$\lambda \geq 0, \tag{80}$$

where R is the number of sub-problems, m_0 is the number of linking constraints, n_r is the number of columns from sub-problem r, $A_0 \in \mathbb{R}^{m_0 \times n_0}$, and $\forall (r, j)$, $a_{r,j} \in \mathbb{R}^{m_0}$. Let $M = R + m_0$ and $N = n_0 + n_1 + \cdots + n_R$ be the number of constraints and

variables in (MP), respectively. In what follows, we focus on the case where (i) R is large, typically in the thousands or tens of thousands, (ii) m_0 is not too large, typically in the hundreds, and (iii) the vectors $a_{r,j} \in \mathbb{R}^{m_0}$ and A_0 are dense.

7.2.1 Instance collection

We build a collection of master problems from two sources. First, we generate instances of Distributed Energy Resources (DER) coordination from [4]. We select a renewable penetration rate $\xi = 0.33$, a time horizon $T = \{24, 48, 96\}$, and a number of resources $R = \{1024, 2048, 4096, 8192, 16{,}384, 32{,}768\}$. Second, we select all two-stage stochastic programming (TSSP) problems from [28] that have at least 1000 scenarios. This yields 18 DER instances, and 27 TSSP instances.

Then, each instance is solved by column generation; master problems are solved with Gurobi's barrier (with crossover) and sub-problems are solved with Gurobi's default settings. In the case of DER instances, which contain mixed-integer variables, only the root node of a branch-and-price tree is solved. Finally, at every tenth CG iteration and the last, the current master problem is saved. Thus, we obtain a dataset of 153 master problems of varying sizes.

CG algorithms benefit from sub-optimal, well-centered interior solutions from the master problem [31], which are typically obtained by simply relaxing an IPM solver's optimality tolerance. These provide the double benefit of stabilizing the CG procedure, thus reducing the number of CG iterations, and speeding-up the resolution of the master problem by stopping the IPM early. Importantly, this approach requires *feasible*, but sub-optimal, dual solutions from the master problem. While in classical primal-dual IPMs, feasibility is generally reached earlier than optimality, in the homogeneous algorithm, infeasibilities and complementarity are reduced at the same rate [2]. As a consequence, for IPM solvers that implement the homogeneous algorithm, such as Mosek, ECOS and Tulip, relaxing optimality tolerances yields no computational gain. Nevertheless, let us formally restate that our present goal is *not* to implement a state-of-the-art column-generation solver, but to quantify the benefits of specialized linear algebra in that context; in particular, specialized linear algebra would equally benefit classical primal-dual IPMs, since the approach of [31] does not affect the master problem's structure. Therefore, we only implement a vanilla CG procedure, which is described in "Appendix A". In particular, we do not make use of any acceleration technique beyond the use of partial pricing.

Tables 2 and 3 display some statistics for DER and TSSP instances, respectively. For each instance, we report: the number of sub-problems R, the number of CG iterations (Iter), total time spent solving the master problem (Master) and pricing sub-problems (Pricing) during the CG procedure and, for the final (MP): the number of linking constraints (m_0), the number of variables (N), and the proportion of non-zero coefficients in the linking constraints (%nz). From the two tables, we see that `DER`, `4node` and `4node-base` instances display relatively dense linking rows, with 35 to 90% coefficients being non-zeros, and a modest number of linking constraints. Other instances are either sparser, e.g., the `env` and `env-diss` instances whose linking rows are only 13% dense, or have few linking constraints, e..g, `phone`. Therefore,

Table 2 Column-generation statistics—DER instances

Instance	R	CG statistics			MP statistics		
		Iter	Master(s)	Pricing(s)	m_0	N	%nz
DER-24	1024	43	4.5	16.6	24	6493	89.7
	2048	40	9.7	40.8	24	12,152	89.0
	4096	41	24.0	86.5	24	24,559	89.4
	8192	40	74.9	155.9	24	48,668	89.7
	16,384	42	195.8	419.9	24	95,845	90.1
	32,768	40	585.7	826.3	24	192,039	89.6
DER-48	1024	49	10.8	25.7	48	7440	87.0
	2048	49	24.6	50.7	48	14,736	88.0
	4096	49	60.8	103.2	48	29,328	88.3
	8192	50	148.1	212.3	48	59,536	88.5
	16,384	48	355.4	418.3	48	114,832	88.5
	32,768	47	853.8	870.2	48	225,424	88.4
DER-96	1024	64	49.0	67.9	96	9504	86.7
	2048	56	90.8	117.0	96	16,672	87.8
	4096	53	191.7	220.1	96	31,520	88.2
	8192	60	603.4	529.9	96	69,920	88.5
	16,384	57	1248.7	993.0	96	133,408	89.0
	32,768	54	3657.2	2163.7	96	254,240	88.7

we expect that our specialized implementation will yield larger gains for the former instances.

7.2.2 Specialized linear algebra

We now describe a specialized Cholesky factorization that exploits the block structure of the master problem. First, the constraint matrix of (MP) is unit block-angular, i.e., it has the form

$$A = \begin{bmatrix} e^T & & 0 \\ & \ddots & \vdots \\ & e^T & 0 \\ A_1 & \cdots & A_R \; A_0 \end{bmatrix}, \tag{81}$$

where

$$A_r = \begin{pmatrix} | & & | \\ a_{r,1} & \dots & a_{r,n_r} \\ | & & | \end{pmatrix} \in \mathbb{R}^{m_0 \times n_r}. \tag{82}$$

Table 3 Column-generation statistics—TSSP instances

Instance	R	CG statistics			MP statistics		
		Iter	Master(s)	Pricing(s)	m_0	N	%nz
4node	1024	24	3.4	2.9	60	5997	41.5
	2048	24	7.9	7.2	60	11,614	38.8
	4096	22	14.4	14.0	60	22,034	38.0
	8192	23	43.8	28.1	60	44,691	37.3
	16,384	23	114.8	58.0	60	87,569	37.9
	32,768	21	248.7	95.8	60	158,895	36.5
4node-base	1024	26	4.5	2.8	60	6197	59.4
	2048	27	11.9	5.5	60	12,968	60.2
	4096	25	35.8	10.5	60	24,153	60.3
	8192	22	46.6	17.4	60	43,399	59.4
	16,384	25	143.8	45.0	60	95,792	60.4
	32,768	23	321.6	79.8	60	179,472	60.2
Assets	37,500	6	2.1	6.2	13	77,928	38.5
env	1200	6	0.1	0.5	85	2860	12.5
	1875	6	0.1	0.7	85	4283	12.9
	3780	6	0.2	1.5	85	8357	13.3
	5292	6	0.2	2.1	85	11,541	13.5
	8232	6	0.4	3.3	85	17,664	13.6
	32,928	6	2.5	13.2	85	69,783	13.8
env-diss	1200	13	0.2	0.7	85	4439	12.3
	1875	15	0.4	1.3	85	7435	12.7
	3780	15	1.0	2.6	85	15,168	12.9
	5292	15	1.5	3.6	85	20,745	13.0
	8232	15	2.5	5.7	85	31,752	13.0
	32,928	14	14.8	21.8	85	123,892	13.3
Phone	32,768	5	1.4	7.6	9	65,553	83.3
StormG2	1000	21	7.9	10.9	306	6075	23.9

Let us recall that the normal equations system writes

$$\left(A(\Theta^{-1} + \rho_p I)^{-1} A^T + \rho_d I\right) \delta_y = \xi, \tag{83}$$

where $\delta_y \in \mathbb{R}^M$, and $\Theta \in \mathbb{R}^{N \times N}$ is a diagonal matrix with positive diagonal. Let S denote the left-hand matrix of (83), and define

$$\tilde{\Theta} = (\Theta^{-1} + \rho_p I)^{-1} = \begin{pmatrix} \tilde{\Theta}_1 & & & \\ & \ddots & & \\ & & \tilde{\Theta}_R & \\ & & & \tilde{\Theta}_0 \end{pmatrix}, \tag{84}$$

and $\tilde{\theta}_r = \tilde{\Theta}_r e \in \mathbb{R}^{n_r}$, for $r = 0, \ldots, R$. Consequently, the normal equations system has the form

$$\begin{bmatrix} d_1 & & & (A_1\tilde{\theta}_1)^T \\ & \ddots & & \vdots \\ & & d_R & (A_R\tilde{\theta}_R)^T \\ A_1\tilde{\theta}_1 & \cdots & A_R\tilde{\theta}_R & \Phi \end{bmatrix} \begin{bmatrix} (\delta_y)_1 \\ \vdots \\ (\delta_y)_R \\ (\delta_y)_0 \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_R \\ \xi_0 \end{bmatrix}, \tag{85}$$

where

$$d_r = e^T\tilde{\theta}_r + \rho_d, \quad r = 1, \ldots, R, \tag{86}$$

$$\Phi = \sum_{r=0}^{R} A_r\tilde{\Theta}_r A_r^T + \rho_d I. \tag{87}$$

Then, define

$$l_r = \frac{1}{d_r} A_r\tilde{\theta}_r \in \mathbb{R}^{m_0}, \quad r = 1, \ldots, R, \tag{88}$$

$$C = \Phi - \sum_{r=1}^{R} \frac{1}{d_r}(A_r\tilde{\theta}_r)(A_r\tilde{\theta}_r)^T \in \mathbb{R}^{m_0 \times m_0}. \tag{89}$$

Given that both S and its upper-left block are positive definite, so is the Schur complement C. Therefore, its Cholesky factorization exists, which we denote $C = L_C D_C L_C^T$. It then follows that a Cholesky factorization of S is given by

$$S = \underbrace{\begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ l_1 & \cdots & l_R & L_C \end{bmatrix}}_{L} \times \underbrace{\begin{bmatrix} d_1 & & & \\ & \ddots & & \\ & & d_R & \\ & & & D_C \end{bmatrix}}_{D} \times \underbrace{\begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ l_1 & \cdots & l_R & L_C \end{bmatrix}^T}_{L^T}. \tag{90}$$

Finally, once the Cholesky factors L and D are computed, the normal equations (85) are solved as follows:

$$(\delta_y)_0 = (L_C D_C L_C^T)^{-1} \left(\xi_0 - \sum_{r=1}^{R} \xi_r l_r \right), \tag{91}$$

$$(\delta_y)_r = \frac{1}{d_r} \xi_r - l_r^T (\delta_y)_0, \quad r = 1, \ldots, R. \tag{92}$$

Exploiting the structure of A yields several computational advantages. First, the factors L and D can be computed directly from A and Θ, i.e., the matrix S does not need to be explicitly formed nor stored, thus saving both time and memory. Second, the sparsity structure of L is known beforehand. Specifically, the lower blocks $l_1, \ldots, l_R$ are all dense column vectors, and the Schur complement C is a dense $m_0 \times m_0$ matrix. Therefore, one does not need a preprocessing phase wherein a sparsity-preserving ordering is computed, thus saving time and making memory allocation fully known in advance. Third, since most heavy operations are performed on dense matrices, efficient cache-exploiting kernels for dense linear algebra can be used, further speeding-up the computations. Finally, note that most operations such as forming the Cholesky factors and performing the backward substitutions, are amenable to parallelization.

7.2.3 Experimental setup

We implement the specialized routines described above in Julia.[8] Specifically, we define a `UnitBlockAngularMatrix` type, together with specialized matrix-vector product methods, and a `UnitBlockAngularFactor` type for computing factorizations and solving linear systems. The lower blocks in A and L are stored as dense matrices, and dense linear algebra operations are performed using BLAS/LAPACK routines directly. This approach is most efficient for problems with few and dense linking constraints, like the ones we consider here, but is not suited to problems that involve a large number of sparse linking constraints. The entire implementation is less than 250 lines of code.

This specialized implementation is passed to the solver by setting the `MatrixOptions` and `KKTOptions` parameters accordingly, as illustrated in Fig. 2. A `Model` object is first created at line 4, and the problem data is imported at line 5. At line 11, we set the `MatrixOptions` parameter to specify that the constraint matrix is of the `UnitBlockAngularMatrix` type with $m_0 = 24$ linking constraints, $n_0 = 72$ linking variables, $n = 6421$ non-linking variables, and $R = 1024$ unit blocks. Then, at line 16, we select the `UnitBlockAngularFactor` type as a linear solver. Finally, the correct matrix and linear solver are instantiated within the `optimize!` call at line 20. Importantly, let us emphasize that no modification was made to Tulip's source code: the correct methods are automatically selected by Julia's multiple dispatch feature, with no performance loss for calling an external function.

Experiments are carried out on an Intel Xeon E5-2637@3.50 GHz CPU, 128 GB RAM machine running Linux; scripts and data for running these experiments are

[8] https://github.com/mtanneau/UnitBlockAngular.jl.

```
import Tulip
using UnitBlockAngular

model = Tulip.Model{Float64}()
Tulip.load_problem!(model, "DER_24_1024_43.mps")  # read file

# Deactivate presolve
model.params.Presolve = 0

# Select matrix options
model.params.MatrixOptions = Tulip.TLA.MatrixOptions(
    UnitBlockAngularMatrix,
    m0=24, n0=72, n=6421, R=1024
)
# Select custom linear solver
model.params.KKTOptions = Tulip.KKT.SolverOptions(
    UnitBlockAngularFactor
)

Tulip.optimize!(model)  # solve the problem
```

Fig. 2 Sample Julia code illustrating the use of a custom `UnitBlockAngularMatrix` type and specialized factorization

available online.[9] We compare the following IPM solvers: CPLEX 12.10 [38], Gurobi 9.0 [35], Mosek 9.2.5 [49], Tulip 0.5.0 with generic linear algebra, and Tulip 0.5.0 with specialized linear algebra; the latter is denoted Tulip*. According to each solver's documentation,[10] both CPLEX and Gurobi implement Mehrotra's predictor-corrector algorithm [46] with Gondzio's multiple corrections [24], while Mosek uses a homogeneous algorithm.

We run each solver on a single thread, and no crossover. Presolve may alter the structure of A in several ways by, e.g., reducing the number of linking constraints, eliminating variables—possibly some entire blocks—or modifying the unit blocks during scaling. Therefore, since we are interested in comparing the per-iteration cost among solvers, we also deactivate presolve. Finally, none of the selected IPM solvers have any warm-start capability, i.e., in a CG algorithm, master problems would effectively be solved from scratch at each CG iteration. Thus, solving master problems independently of one another, as is done here, does not invalidate our analysis.

7.2.4 Results

Results are reported in Table 4; for conciseness, only the final master problem of each CG instance is included here. Results for the entire collection can be found in Table 6, "Appendix B". For each instance and solver, we report total CPU time (T), in seconds, and the number of IPM iterations (Iter). In Table 6, the number of CG iterations (at which the instance was obtained) is also displayed.

[9] Code for generating DER instances is available at https://github.com/mtanneau/DER_experiments and for TSPP instances at https://github.com/mtanneau/TSSP.

[10] Mosek always uses its homogeneous algorithm. With default settings, CPLEX and Gurobi only do so when solving node relaxations of a MIP model.

Table 4 Performance comparison of IPM solvers on structured instances

Problem	R	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
		T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
DER-24	1024	0.2	33	0.2	27	**0.2**	21	1.1	33	0.5	33
	2048	0.4	48	**0.4**	36	0.4	27	2.5	47	0.6	47
	4096	1.0	40	1.0	32	**0.8**	26	4.7	38	1.0	38
	8192	4.3	79	2.7	46	**2.4**	38	19.0	67	2.6	68
	16,384	10.8	93	5.3	48	**5.3**	42	49.8	86	5.3	83
	32,768	33.9	148	19.4	85	12.3	43	103.6	91	**11.4**	86
DER-48	1024	0.5	37	0.4	19	**0.3**	21	1.8	28	0.4	28
	2048	1.6	40	1.0	21	**0.8**	25	5.7	37	0.9	37
	4096	4.1	44	2.0	25	2.0	27	14.1	39	**1.7**	39
	8192	9.7	51	4.2	20	4.5	24	37.0	46	**3.4**	47
	16,384	22.3	64	9.9	29	9.3	28	89.7	60	**7.4**	57
	32,768	57.1	85	21.6	32	21.1	33	178.8	59	**14.2**	54
DER-96	1024	3.3	38	1.2	19	0.9	22	6.6	31	**0.9**	31
	2048	7.9	45	2.4	20	1.7	21	18.2	38	**1.7**	37
	4096	16.3	51	5.5	24	5.2	28	42.6	40	**3.2**	40
	8192	51.7	75	15.5	29	11.1	31	137.6	60	**8.8**	57
	16,384	107.8	86	31.9	31	24.4	39	260.0	55	**17.3**	59
	32,768	291.9	119	102.9	54	**55.5**	47	753.7	89	65.4	86
4node	1024	15.7	21	0.4	43	**0.2**	25	1.3	30	0.5	32
	2048	0.7	38	0.6	27	**0.6**	25	2.1	36	0.9	36
	4096	1.1	27	1.7	37	**0.7**	17	4.5	28	1.2	28
	8192	2.7	30	2.3	29	**1.8**	24	12.3	35	2.7	33
	16,384	5.8	29	10.7	53	**4.0**	22	26.4	33	4.6	33
	32,768	17.0	57	18.7	55	14.6	41	74.8	56	**14.2**	59
4node-base	1024	17.0	17	1.0	60	**0.3**	27	1.4	28	0.6	27
	2048	1.0	35	2.6	72	**0.8**	33	3.7	32	0.9	33
	4096	2.3	38	5.3	72	**1.5**	34	9.1	34	1.8	34
	8192	3.8	29	3.7	27	**2.6**	25	19.7	36	2.8	36
	16,384	13.5	53	26.2	74	8.0	37	63.4	53	**7.0**	47
	32,768	20.3	37	29.0	43	14.9	30	107.7	48	**12.9**	50
Assets	37,500	1.6	21	**0.6**	12	1.1	20	2.0	13	1.0	13
env	1200	0.0	21	**0.0**	12	0.1	16	0.3	16	0.3	16
	1875	0.1	22	**0.0**	12	0.1	13	0.4	16	0.4	16
	3780	0.1	25	**0.1**	12	0.1	14	0.7	17	0.5	17
	5292	0.2	27	**0.1**	13	0.1	13	0.7	17	0.7	17
	8232	0.3	26	**0.2**	13	0.3	14	1.1	18	1.2	18
	32,928	1.7	26	**0.9**	13	1.3	17	5.1	21	4.4	21

Table 4 continued

Problem	R	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
		T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
env-diss	1200	0.1	15	**0.0**	15	0.1	17	0.4	23	0.4	23
	1875	0.1	17	**0.1**	18	0.1	18	0.6	22	0.5	22
	3780	0.2	20	**0.1**	18	0.2	18	1.0	22	1.0	22
	5292	0.3	22	**0.3**	23	0.3	22	1.3	25	1.5	25
	8232	1.0	31	0.6	29	**0.5**	23	3.2	35	2.6	35
	32,928	4.8	28	**2.1**	22	2.5	19	10.0	27	7.7	27
Phone	32,768	0.5	15	**0.4**	8	0.6	8	1.9	10	0.7	10
stormG2	1000	1.6	37	0.8	18	**0.5**	22	4.0	29	1.7	28

Results obtained without presolve
Shortest computing times are in bold

We begin by comparing Tulip with and without specialized linear algebra. First, the number of IPM iterations is almost identical between the two, with differences never exceeding 6 IPM iterations. The differences are caused by small numerical discrepancies between the linear algebra implementations, which remain negligible until close to the optimum. Second, using specialized linear algebra results in a significant speedup, especially on larger and denser instances. Indeed, on large `DER` and `4node` instances, we typically observe a tenfold speedup. For smaller and sparser instances, e.g., the `env` instances, or with very few linking constraints such as `phone`, using specialized linear algebra still brings a moderate performance improvement.

Next, we compare Tulip with specialized linear algebra, Tulip*, against state-of-the-art commercial solvers. Given CPLEX's poorer relative performance on this test set, in the following we mainly discuss the results of Tulip* in comparison with Mosek and Gurobi. First, our specialized implementation is able to outperform commercial codes on the larger and denser instances, while remaining within a reasonable factor on smaller and sparse instances. The largest performance improvement is observed on the `DER-48` instance with $R = 32,768$, for which Tulip* achieves a 30% speedup over the fastest commercial alternative. This demonstrates that, when exploiting structure, open-source solvers can compete with state-of-the-art commercial codes. Second, Tulip's iteration count is typically 50 to 100% larger than that of Mosek and Gurobi. When comparing average per-iteration times on the denser instances, we observe that Tulip is generally 1.5 to 3 times faster than Gurobi and Mosek.

Recall that the cost of an individual IPM iteration depends not only on problem size and the efficiency of the underlying linear algebra, but also on algorithmic features such as the number of corrections, which we cannot measure directly. Nevertheless, the performance difference is significant enough to suggest that algorithmic improvements aimed at reducing the number of IPM iterations would substantially improve Tulip's performance.

7.3 Solving problems in extended precision

Almost all optimization solvers perform computations in double precision (64 bits) floating-point arithmetic, denoted by `double` and `Float64` in C and Julia, respectively. Julia's parametric type system and multiple dispatch allow to write generic code: in the present case, this results in Tulip's code can be used with *arbitrary* arithmetic. We now illustrate this functionality for solving problems in higher precision.

The ability to use extended precision is useful is various contexts. First, while typical numerical tolerances for most LP solvers range from 10^{-6} to 10^{-8}, one may *require* levels of precision that exceed what double-precision arithmetic can achieve. For instance, in [44], the authors consider problems where variations of order 10^{-6} to 10^{-10} are meaningful. One remedy to this issue is to use, e.g., quadruple-precision arithmetic. Second, even with "standard" tolerances, solvers may encounter numerical issues for badly scaled problems, sometimes resulting in the optimization being aborted. These issues may be alleviated by using higher precision, thereby allowing to solve a given challenging instance, albeit at a performance cost. Finally, in the course of developing a new optimization software or algorithmic technique, identifying whether inconsistencies are due to numerical issues, mathematical errors, or software bugs, can be a daunting and time-consuming task. In that context, the ability to easily switch between different arithmetics enables one to factor out rounding errors and related issues, thereby identifying –or ruling out– other sources of errors.

Let us note that a handful of simplex-based solvers have the capability to compute extended-precision or exact solutions to LP problems, either by performing computations in exact arithmetic, solving a sequence of LPs with increasing precision, or using iterative refinement techniques; the reader is referred to [23] for an overview of such approaches and available software. We are not aware of any existing interior-point solver with this capability. As pointed out in [23], performing all computations in the prescribed arithmetic, as is the case in Tulip, is intractable for large problems. Consequently, Tulip should not be viewed as a competitive tool for solving LPs in extended precision. Rather, the main advantage of our implementation is its simplicity and flexibility: it required no modification of the source code, runs the same algorithm regardless of the arithmetic, and its use is straightforward. Indeed, as Fig. 3 illustrates, besides loading the appropriate packages, the user only needs to specify the arithmetic when creating a model; the rest of the code is identical. Therefore, using Tulip with higher-precision arithmetic is best envisioned as a prototyping tool, or to occasionally solve a numerically challenging problem.

As an example of this use case, we consider the 6 instances from Sect. 7.1 that required more than 100 IPM iterations; this generally indicates numerical issues. Each instance is solved with Tulip in quadruple-precision arithmetic. We use the `Double64` type from the `DoubleFloats` Julia package, which implements the so-called "double–double" arithmetic, wherein a pair of double-precision numbers is used to approximate one quadruple-precision number. This implementation allows to exploit fast, hardware-implemented, double-precision arithmetic, while achieving similar precision as 128 bits floating point arithmetic. Experiments were carried on the same cluster of machines as in Sect. 7.1. Besides the different arithmetic, we increase

```
import Tulip

model = Tulip.Model{Float64}()  # Float64 arithmetic
Tulip.load_problem!(tlp, "neos2.mps")  # read file

Tulip.optimize!(model)  # solve the problem
```

(a) Using `Float64` arithmetic.

```
import Tulip
using DoubleFloats

model = Tulip.Model{Double64}()  # Double64 arithmetic
Tulip.load_problem!(tlp, "neos2.mps")  # read file

Tulip.optimize!(model)  # solve the problem
```

(b) Using `Double64` arithmetic.

Fig. 3 Sample Julia code illustrating the use of different arithmetics

the time limit to 40, 000s and set tolerances to 10^{-8}, that is, the problems are solved up to usual double-precision tolerances. All other settings are left identical to those of Sect. 7.1.

Results are displayed in Table 5. For each instance and arithmetic, we report the total solution time (CPU) in seconds, the number of IPM iterations (Iter), and the solver's result status (Status). We first note that, when using `Double64` arithmetic, all instances are solved to optimality. This validates the earlier finding that instances `ns1688926` and `watson_2` did encounter numerical issues. Second, we observe a drastic reduction in the number of IPM iterations from `Float64` to `Double64`, with decreases in iteration counts ranging from 40% to over 90% in the case of `neos2` and `ns1688926`. Third, while the per-iteration cost of `Double64` is typically 8× larger than that of `Float64`, overall computing times do not increase as much due to the reduction in IPM iterations. In fact, in the extreme cases of `ns1688926`, solving the problem in `Double64` is significantly faster than solving it in `Float64`. Finally, the results of Table 5 suggest that Tulip would most benefit from greater numerical stability on instances such as `neos2`, `ns1688926`, `stat96v1` and `watson_2`. This may include, for instance, the use of iterative refinement when solving Newton systems. On the other hand, similar iterations counts for both arithmetics, would have suggested algorithmic issues, e.g., short steps being taken due to the iterates being far from the central path.

8 Conclusion

In this paper, we have described a regularized homogeneous interior-point algorithm and its implementation in Tulip, an open-source linear optimization solver. Our solver is written in Julia, and leverages some of the language's features to propose a flexible and easily-customized implementation. Most notably, Tulip's algorithmic framework is fully disentangled from linear algebra implementations and the choice of arithmetic.

Table 5 Problematic instances from the Mittelmann benchmark

Instance	Float64			Double64		
	CPU (s)	Iter	Status	CPU(s)	Iter	Status
neos2	462.1	460	Optimal	265.1	37	Optimal
ns1688926	1007.7	500	Iterations	142.8	18	Optimal
s250r10	257.2	169	Optimal	1385.0	93	Optimal
shs1023	371.6	266	Optimal	968.8	105	Optimal
stat96v1	41.3	275	Optimal	30.4	42	Optimal
watson_2	295.7	500	Iterations	243.4	67	Optimal

The performance of the code has been evaluated on generic instances from H. Mittelmann's benchmark testset, on two sets of structured instances for which we developed specialized linear algebra routines, and on numerically problematic instances using higher-precision arithmetic. The computational evaluation has shown three main results. First, when solving generic LP instances, Tulip is competitive with open-source IPM solvers that have a Julia interface. Second, when solving structured problems, the use of custom linear algebra routines yields a tenfold speedup over generic ones, thereby outperforming state-of-the-art commercial IPM solvers on larger and denser instances. These results demonstrate the benefits of being able to seamlessly integrate specialized linear algebra within an interior-point algorithm. Third, in a development context, Tulip can be conveniently used in conjunction with higher-precision arithmetic, so as to alleviate numerical issues.

Finally, future developments will consider the use of iterative methods for solving linear systems, the development of more general structured linear algebra routines and their multi-threaded implementation, and more efficient algorithmic techniques for solving problems in extended precision. Because of the way in which Tulip has been designed, all those developments do not require any significant rework of the code structure.

Acknowledgements We thank Dominique Orban for helpful discussions on the regularization scheme and its implementation. We are also indebted to three anonymous referees for their careful reading and constructive suggestions that helped us improving the quality and readability of the paper.

A Dantzig–Wolfe decomposition and column generation

In this section, we present the Dantzig–Wolfe decomposition principle [13] and the basic column-generation framework. We refer to [15] for a thorough overview of column generation, and the relation between Dantzig–Wolfe decomposition and Lagrangian decomposition.

A.1 Dantzig–Wolfe decomposition

Consider the problem

$$
\begin{aligned}
(P) \quad \min_{x} \quad & \sum_{r=0}^{R} c_r^T x_r \\
s.t. \quad & \sum_{r=0}^{R} A_r x_r = b_0, \\
& x_0 \geq 0, \\
& x_r \in \mathcal{X}_r, \quad r = 1, \ldots, R,
\end{aligned}
$$

where, for each $r = 1, \ldots, R$, $\mathcal{X}_r$ is defined by a finite number of linear inequalities, plus integrality restrictions on some of the coordinates of x_r. Therefore, the convex hull of $\mathcal{X}_r$, denoted by $conv(\mathcal{X}_r)$, is a polyhedron whose set of extreme points (resp. extreme rays) is denoted by Ω_r (resp. Γ_r). Any element of $conv(\mathcal{X}_r)$ can thus be written as a convex combination of extreme points $\{\omega\}_{\omega \in \Omega_r}$, plus a non-negative combination of extreme rays $\{\rho\}_{\rho \in \Gamma_r}$ i.e.,

$$
conv(\mathcal{X}_r) = \left\{ \sum_{\omega \in \Omega_r} \lambda_\omega \omega + \sum_{\rho \in \Gamma_r} \lambda_\rho \rho \;\middle|\; \lambda \geq 0, \sum_{\omega} \lambda_\omega = 1 \right\}. \tag{93}
$$

The Dantzig–Wolfe decomposition principle [13] then consists in substituting x_r with such a combination of extreme points and extreme rays. This change of variable yields the so-called *Master Problem*

$$
\begin{aligned}
(MP) \quad \min_{x,\lambda} \quad & c_0^T x_0 + \sum_{r=1}^{R} \sum_{\omega \in \Omega_r} c_{r,\omega} \lambda_{r,\omega} + \sum_{r=1}^{R} \sum_{\rho \in \Gamma_r} c_{r,\rho} \lambda_{r,\rho} && (94) \\
s.t. \quad & \sum_{\omega \in \Omega_r} \lambda_{r,\omega} = 1, \quad r = 1, \ldots, R && (95) \\
& A_0 x_0 + \sum_{r=1}^{R} \sum_{\omega \in \Omega_r} a_{r,\omega} \lambda_{r,\omega} + \sum_{r=1}^{R} \sum_{\rho \in \Gamma_r} a_{r,\rho} \lambda_{r,\rho} = b_0, && (96) \\
& x_0, \lambda \geq 0, && (97) \\
& \sum_{\omega \in \Omega_r \lambda_{r,\omega} \omega + \sum_{\rho \in \Gamma_r} \lambda_{r,\rho} \rho \in \mathcal{X}_r, \quad r=1,\ldots,R} && (98)
\end{aligned}
$$

where $c_{r,\omega} = c_r^T \omega$, $c_{r,\rho} = c_r^T \rho$, and $a_{r,\omega} = A_r \omega$, $a_{r,\rho} = A_r \rho$. Constraints (95) and (96) are referred to as *convexity* and *linking* constraints, respectively.

The linear relaxation of (MP) is given by (94)–(97); its objective value is greater or equal to that of the linear relaxation of (P) [15]. Note that if (P) in a linear program,

i.e., all variables are continuous, then constraints (98) are redundant, and (94)–(97) is equivalent to (P). In the mixed-integer case, problem (94)–(97) is the root node in a branch-and-price tree. In this work, we focus on solving this linear relaxation. Thus, in what follows, we make a slight abuse of notation and use the term "Master Problem" to refer to (94)–(97) instead.

A.2 Column generation

The Master Problem has exponentially many variables. Therefore, it is typically solved by column generation, wherein only a small subset of the variables are considered. Additional variables are generated iteratively by solving an auxiliary sub-problem.

Let $\bar{\Omega}_r$ (resp. $\bar{\Gamma}_r$) be a small subset of Ω_r (resp. of Γ_r), and define the *Restricted Master Problem* (RMP)

$$(RMP) \quad \min_{\lambda} \quad c_0^T x_0 + \sum_{r=1}^{R} \sum_{\omega \in \bar{\Omega}_r} c_{r,\omega} \lambda_{r,\omega} + \sum_{r=1}^{R} \sum_{\rho \in \bar{\Gamma}_r} c_{r,\rho} \lambda_{r,\rho} \tag{99}$$

$$s.t. \quad \sum_{\omega \in \bar{\Omega}_r} \lambda_{r,\omega} = 1, \quad r = 1, \ldots, R \tag{100}$$

$$\sum_{r=1}^{R} \sum_{\omega \in \bar{\Omega}_r} a_{r,\omega} \lambda_{r,\omega} + \sum_{r=1}^{R} \sum_{\rho \in \bar{\Gamma}_r} a_{r,\rho} \lambda_{r,\rho} = b_0, \tag{101}$$

$$x_0, \lambda \geq 0. \tag{102}$$

In all that follows, we assume that (RMP) is feasible and bounded. Note that feasibility can be obtained by adding artificial slacks and surplus variables with sufficiently large cost, effectively implementing an l_1 penalty. If the RMP is unbounded, then so is the MP.

Let $\sigma \in \mathbb{R}^R$ and $\pi \in \mathbb{R}^{m_0}$ denote the vector of dual variables associated to convexity constraints (100) and linking constraints constraints (101), respectively. Here, we assume that (σ, π) is dual-optimal for (RMP); the use of interior, sub-optimal dual solutions is explored in [31]. Then, for given r, $\omega \in \Omega_r$ and $\rho \in \Gamma_r$, the reduced cost of variable $\lambda_{r,\omega}$ is

$$\bar{c}_{r,\omega} = c_{r,\omega} - \pi^T a_{r,\omega} - \sigma_r = (c_r^T - \pi^T A_r)\omega - \sigma_r,$$

while the reduced cost of variable $\lambda_{r,\rho}$ is

$$\bar{c}_{r,\rho} = c_{r,\rho} - \pi^T a_{r,\rho} = (c_r^T - \pi^T A_r)\rho.$$

If $\bar{c}_{r,\omega} \geq 0$ for all r, $\omega \in \Omega_r$ and $\bar{c}_{r,\rho} \geq 0$ for all r, $\rho \in \Gamma_r$, then the current solution is optimal for the MP. Otherwise, a variable with negative reduced cost is added to the RMP. Finding such a variable, or proving that none exists, is called the *pricing step*.

Explicitly iterating through the exponentially large sets Ω_r and Γ_r is prohibitively expensive. Nevertheless, the pricing step can be written as the following MILP:

$$(SP_r) \quad \min_{x_r} \quad (c_r^T - \pi^T A)x_r - \sigma_r \tag{103}$$

$$s.t. \quad x_r \in \mathcal{X}_r, \tag{104}$$

which we refer to as the r^{th} *sub-problem*. If SP_r is infeasible, then $\mathcal{X}_r$ is empty, and the original problem P is infeasible. This case is ruled out in all that follows. Then, since the objective of SP_r is linear, any optimal solution is either an extreme point $\omega \in \Omega_r$ (bounded case), or an extreme ray $\rho \in \Gamma_r$ (unbounded case). The corresponding variable $\lambda_{r,\omega}$ or $\lambda_{r,\rho}$ is identified by retrieving an optimal point or unbounded ray. Finally, note that all R sub-problems $SP_1, \ldots, SP_R$ can be solved independently from one another. Optimality in the Master Problem is attained when no variable with negative reduced cost can be identified from all R sub-problems.

We now describe a basic column-generation procedure, which is formally stated in Algorithm 2. The algorithm starts with an initial RMP that contains a small subset of columns, some of which may be artificial to ensure feasibility. At the beginning of each iteration, the RMP is solved to optimality, and a dual solution (π, σ) is obtained which is used to perform the pricing step. Each sub-problem is solved to identify a variable with most negative reduced cost. If a variable with negative reduced cost is found, it is added to the RMP; if not, the column-generation procedure stops.

Algorithm 2 Column-generation procedure

Input: Initial RMP
while stopping criterion not met **do**
  Solve RMP and obtain optimal dual variables (π, σ)
  // Pricing step
  for all $r \in \mathcal{R}$ **do**
    Solve SP_r with the query point (π, σ_r); obtain ω^* or ρ^*
    if $\bar{c}_{r,\omega^*} < 0$ or $\bar{c}_{r,\rho^*} < 0$ **then**
      Add corresponding column to the RMP
    end if
  end for
  // Stopping criterion
  if no column added to RMP **then**
    STOP
  end if
end while

For large instances with numerous subproblems, full pricing, wherein all subproblems are solved at each iteration, is often not the most efficient approach. Therefore, we implemented a partial pricing strategy, in which subproblems are solved in a random order until either all subproblems have been solved, or a user-specified number of columns with negative reduced cost have been generated.

B Detailed results on structured LP instances

See Table 6.

Table 6 Structured instances: performance comparison of IPM solvers

Instance	R	CG	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
			T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
DER-24	1024	10	**0.0**	19	0.0	15	0.1	19	0.4	19	0.3	19
DER-24	1024	20	0.1	27	**0.1**	23	0.1	21	0.5	23	0.3	23
DER-24	1024	30	0.1	28	**0.1**	22	0.1	24	0.8	26	0.3	26
DER-24	1024	40	0.2	44	**0.2**	27	0.2	28	1.2	37	0.4	39
DER-24	1024	43	0.2	33	0.2	27	**0.2**	21	1.1	33	0.5	33
DER-24	2048	10	0.2	31	**0.1**	21	0.1	20	0.8	23	0.3	23
DER-24	2048	20	0.3	30	**0.1**	19	0.2	18	1.0	22	0.3	22
DER-24	2048	30	**0.3**	29	0.3	28	0.3	20	1.4	30	0.5	30
DER-24	2048	40	0.4	48	**0.4**	36	0.4	27	2.5	47	0.6	47
DER-24	4096	10	0.4	35	**0.2**	20	0.3	19	1.3	28	0.5	28
DER-24	4096	20	0.8	39	0.4	23	**0.4**	22	2.1	29	0.5	29
DER-24	4096	30	1.3	65	1.0	42	**0.7**	29	5.4	58	0.9	56
DER-24	4096	40	1.1	38	1.1	32	**0.9**	26	5.0	38	0.9	38
DER-24	4096	41	1.0	40	1.0	32	**0.8**	26	4.7	38	1.0	38
DER-24	8192	10	0.9	32	**0.5**	18	0.6	21	2.6	25	0.7	25
DER-24	8192	20	2.0	39	1.1	26	**1.1**	21	5.9	34	1.1	34
DER-24	8192	30	2.9	62	**1.8**	36	2.1	40	12.5	55	1.9	55
DER-24	8192	40	4.3	79	2.7	46	**2.4**	38	19.0	67	2.6	68
DER-24	16, 384	10	2.5	47	1.3	26	1.7	26	9.0	39	**1.2**	36
DER-24	16, 384	20	4.1	42	2.1	29	2.5	22	13.8	37	**2.0**	37
DER-24	16, 384	30	5.4	55	3.4	36	3.6	26	23.1	48	**3.0**	48
DER-24	16, 384	40	12.4	110	10.2	88	**6.0**	51	57.6	100	6.7	100
DER-24	16, 384	42	10.8	93	5.3	48	**5.3**	42	49.8	86	5.3	83
DER-24	32, 768	10	4.6	39	3.3	34	3.5	23	17.9	36	**2.2**	34
DER-24	32, 768	20	11.0	53	8.8	52	8.0	39	47.4	66	**5.5**	65
DER-24	32, 768	30	14.5	68	12.3	56	**8.2**	31	96.1	100	11.2	100
DER-24	32, 768	40	33.9	148	19.4	85	12.3	43	103.6	91	**11.4**	86
DER-48	1024	10	0.1	24	**0.1**	13	0.1	21	0.8	24	0.3	24
DER-48	1024	20	0.2	26	**0.2**	20	0.2	22	1.1	26	0.3	26
DER-48	1024	30	0.3	31	**0.2**	16	0.2	22	1.5	32	0.5	32
DER-48	1024	40	0.4	32	0.4	21	**0.3**	22	1.6	30	0.4	30
DER-48	1024	49	0.5	37	0.4	19	**0.3**	21	1.8	28	0.4	28
DER-48	2048	10	0.3	26	0.3	19	**0.3**	20	1.3	24	0.4	25
DER-48	2048	20	0.7	37	0.5	21	0.5	22	2.2	31	**0.4**	31

Table 6 continued

Instance	R	CG	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
			T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
DER-48	2048	30	0.8	37	0.6	19	0.5	21	2.6	27	**0.5**	27
DER-48	2048	40	1.2	38	0.9	24	0.7	21	4.1	33	**0.7**	33
DER-48	2048	49	1.6	40	1.0	21	**0.8**	25	5.7	37	0.9	37
DER-48	4096	10	0.8	34	0.6	19	0.6	21	3.2	28	**0.4**	28
DER-48	4096	20	1.5	41	1.1	24	1.0	24	5.8	34	**0.8**	34
DER-48	4096	30	1.7	38	1.4	23	1.4	23	8.2	35	**1.0**	35
DER-48	4096	40	3.0	40	2.2	30	1.9	25	9.9	33	**1.4**	33
DER-48	4096	49	4.1	44	2.0	25	2.0	27	14.1	39	**1.7**	39
DER-48	8192	10	2.1	39	1.5	26	1.5	25	8.1	32	**1.1**	32
DER-48	8192	20	3.9	44	1.9	18	2.0	23	12.7	31	**1.5**	31
DER-48	8192	30	7.2	55	2.9	26	2.9	26	20.4	39	**2.2**	39
DER-48	8192	40	7.3	45	4.0	24	3.8	22	25.4	38	**2.4**	38
DER-48	8192	50	9.7	51	4.2	20	4.5	24	37.0	46	**3.4**	47
DER-48	16,384	10	5.0	49	2.9	25	3.8	35	22.4	41	**2.3**	41
DER-48	16,384	20	7.8	45	5.5	28	5.2	26	31.5	37	**2.8**	37
DER-48	16,384	30	14.6	59	7.3	29	6.3	25	53.6	50	**5.0**	48
DER-48	16,384	40	16.3	53	9.3	27	8.5	27	64.5	50	**5.2**	45
DER-48	16,384	48	22.3	64	9.9	29	9.3	28	89.7	60	**7.4**	57
DER-48	32,768	10	10.8	49	7.4	27	8.1	29	46.9	41	**4.1**	41
DER-48	32,768	20	16.8	47	8.6	24	11.2	32	69.5	42	**5.7**	41
DER-48	32,768	30	30.5	61	14.9	26	13.4	26	107.5	51	**10.0**	51
DER-48	32,768	40	36.2	57	21.1	31	16.9	28	133.8	51	**10.4**	46
DER-48	32,768	47	57.1	85	21.6	32	21.1	33	178.8	59	**14.2**	54
DER-96	1024	10	0.5	27	0.3	18	**0.3**	20	1.4	23	0.5	23
DER-96	1024	20	0.8	29	0.5	18	0.5	25	2.4	27	**0.4**	27
DER-96	1024	30	1.2	32	0.6	17	0.6	23	3.5	30	**0.5**	30
DER-96	1024	40	1.6	34	1.0	19	0.7	22	4.1	31	**0.6**	32
DER-96	1024	50	2.2	34	1.2	19	0.8	22	5.8	30	**0.7**	30
DER-96	1024	60	2.6	34	1.4	19	1.0	23	6.9	31	**0.8**	31
DER-96	1024	64	3.3	38	1.2	19	0.9	22	6.6	31	**0.9**	31
DER-96	2048	10	1.2	37	0.8	21	0.7	29	4.0	29	**0.5**	29
DER-96	2048	20	2.2	33	1.1	19	0.9	23	5.9	26	**0.8**	26
DER-96	2048	30	2.5	44	1.6	22	1.3	25	9.9	35	**1.1**	35
DER-96	2048	40	4.8	38	2.3	26	1.5	23	12.4	33	**1.2**	33

Table 6 continued

Instance	R	CG	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
			T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
DER-96	2048	50	6.7	41	2.4	23	1.8	25	15.2	36	**1.6**	36
DER-96	2048	56	7.9	45	2.4	20	1.7	21	18.2	38	**1.7**	37
DER-96	4096	10	3.0	41	1.7	24	1.5	28	9.5	32	**1.0**	32
DER-96	4096	20	4.4	51	2.9	27	1.9	26	18.2	39	**1.6**	40
DER-96	4096	30	6.0	53	3.6	24	2.7	28	21.1	35	**2.0**	36
DER-96	4096	40	13.6	53	4.4	24	3.3	27	31.2	39	**2.4**	39
DER-96	4096	50	14.2	45	5.7	25	4.2	25	36.1	37	**2.7**	37
DER-96	4096	53	16.3	51	5.5	24	5.2	28	42.6	40	**3.2**	40
DER-96	8192	10	5.6	53	4.3	27	3.0	28	23.0	35	**2.6**	35
DER-96	8192	20	11.1	62	7.2	33	4.9	32	43.4	40	**3.4**	40
DER-96	8192	30	13.5	59	8.8	31	5.9	26	54.4	40	**3.8**	40
DER-96	8192	40	32.7	63	12.6	35	7.4	25	77.6	45	**5.0**	45
DER-96	8192	50	39.3	65	11.5	25	10.1	33	89.1	44	**6.6**	45
DER-96	8192	60	51.7	75	15.5	29	11.1	31	137.6	60	**8.8**	57
DER-96	16,384	10	12.6	61	11.0	37	6.9	34	55.0	41	**4.4**	41
DER-96	16,384	20	21.2	62	14.5	31	10.9	31	92.3	42	**6.1**	42
DER-96	16,384	30	30.1	68	18.5	32	14.2	34	147.9	50	**10.1**	52
DER-96	16,384	40	70.0	69	21.8	28	16.1	30	196.5	54	**11.5**	52
DER-96	16,384	50	85.5	73	27.7	29	18.6	32	231.8	57	**14.5**	54
DER-96	16,384	57	107.8	86	31.9	31	24.4	39	260.0	55	**17.3**	59
DER-96	32,768	10	28.1	70	25.4	45	18.0	39	152.5	52	**11.8**	49
DER-96	32,768	20	39.9	57	33.8	36	18.9	28	180.4	37	**10.7**	39
DER-96	32,768	30	61.9	72	46.6	34	27.9	31	337.6	58	**21.3**	58
DER-96	32,768	40	174.6	88	70.8	42	40.4	39	483.2	69	**30.0**	66
DER-96	32,768	50	233.6	102	58.8	32	46.8	36	609.0	74	**43.1**	72
DER-96	32,768	54	291.9	119	102.9	54	**55.5**	47	753.7	89	65.4	86
4node	1024	10	0.1	28	0.3	53	**0.1**	28	0.9	31	0.5	30
4node	1024	20	0.2	27	0.2	22	**0.2**	26	1.0	27	0.4	27
4node	1024	24	15.7	21	0.4	43	**0.2**	25	1.3	30	0.5	32
4node	2048	10	19.6	24	0.7	51	**0.4**	32	1.9	44	0.9	37
4node	2048	20	0.7	38	0.8	42	**0.5**	37	1.9	33	0.9	32
4node	2048	24	0.7	38	0.6	27	**0.6**	25	2.1	36	0.9	36
4node	4096	10	0.9	36	2.1	63	**0.7**	28	3.6	40	1.3	40

Table 6 continued

Instance	R	CG	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
			T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
4node	4096	20	0.9	23	1.0	27	**0.6**	19	3.7	26	1.3	26
4node	4096	22	1.1	27	1.7	37	**0.7**	17	4.5	28	1.2	28
4node	8192	10	**1.8**	33	3.4	62	1.8	33	10.2	44	2.1	43
4node	8192	20	3.2	42	4.3	51	**2.3**	36	14.2	43	3.2	44
4node	8192	23	2.7	30	2.3	29	**1.8**	24	12.3	35	2.7	33
4node	16,384	10	6.8	61	11.4	85	**5.7**	53	34.4	62	5.7	67
4node	16,384	20	7.0	42	20.8	108	6.4	44	31.6	45	**5.5**	44
4node	16,384	23	5.8	29	10.7	53	**4.0**	22	26.4	33	4.6	33
4node	32,768	10	9.8	42	11.8	42	9.1	40	56.4	52	**8.3**	53
4node	32,768	20	17.0	58	35.0	95	**13.5**	45	81.9	60	15.7	65
4node	32,768	21	17.0	57	18.7	55	14.6	41	74.8	56	**14.2**	59
4node-base	1024	10	**0.1**	24	0.2	18	0.3	21	0.8	24	0.3	24
4node-base	1024	20	0.3	25	0.3	23	**0.2**	28	1.3	28	0.5	28
4node-base	1024	26	17.0	17	1.0	60	**0.3**	27	1.4	28	0.6	27
4node-base	2048	10	15.0	15	0.8	36	**0.3**	23	1.4	29	0.5	30
4node-base	2048	20	0.8	37	1.0	31	**0.7**	35	3.1	36	0.8	36
4node-base	2048	27	1.0	35	2.6	72	**0.8**	33	3.7	32	0.9	33
4node-base	4096	10	0.9	35	3.3	92	**0.7**	24	4.4	38	0.9	42
4node-base	4096	20	1.8	38	4.4	76	**1.1**	30	8.8	42	1.6	42
4node-base	4096	25	2.3	38	5.3	72	**1.5**	34	9.1	34	1.8	34
4node-base	8192	10	1.8	33	2.0	21	1.5	26	7.6	29	**1.5**	29
4node-base	8192	20	4.4	39	16.3	133	3.9	40	18.1	39	**2.7**	39
4node-base	8192	22	3.8	29	3.7	27	**2.6**	25	19.7	36	2.8	36
4node-base	16,384	10	4.4	38	10.1	57	3.6	30	19.3	39	**3.4**	39
4node-base	16,384	20	10.6	49	17.1	51	6.9	36	46.2	46	**5.6**	45
4node-base	16,384	25	13.5	53	26.2	74	8.0	37	63.4	53	**7.0**	47
4node-base	32,768	10	10.9	45	76.1	214	10.3	40	44.3	36	**6.0**	36
4node-base	32,768	20	27.8	68	80.1	125	25.9	72	119.3	59	**15.6**	63
4node-base	32,768	23	20.3	37	29.0	43	14.9	30	107.7	48	**12.9**	50
assets	37,500	6	1.6	21	**0.6**	12	1.1	20	2.0	13	1.0	13
env	1200	6	0.0	21	**0.0**	12	0.1	16	0.3	16	0.3	16
env	1875	6	0.1	22	**0.0**	12	0.1	13	0.4	16	0.4	16
env	3780	6	0.1	25	**0.1**	12	0.1	14	0.7	17	0.5	17
env	5292	6	0.2	27	**0.1**	13	0.1	13	0.7	17	0.7	17
env	8232	6	0.3	26	**0.2**	13	0.3	14	1.1	18	1.2	18
env	32,928	6	1.7	26	**0.9**	13	1.3	17	5.1	21	4.4	21

Table 6 continued

Instance	R	CG	CPLEX		Gurobi		Mosek		Tulip		Tulip*	
			T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter	T(s)	Iter
env-diss	1200	10	0.0	17	**0.0**	19	0.0	16	0.4	22	0.4	22
env-diss	1200	13	0.1	15	**0.0**	15	0.1	17	0.4	23	0.4	23
env-diss	1875	10	0.1	27	**0.1**	17	0.1	20	0.6	22	0.5	22
env-diss	1875	15	0.1	17	**0.1**	18	0.1	18	0.6	22	0.5	22
env-diss	3780	10	0.2	23	**0.1**	16	0.1	17	0.8	21	0.7	21
env-diss	3780	15	0.2	20	**0.1**	18	0.2	18	1.0	22	1.0	22
env-diss	5292	10	0.4	31	**0.2**	25	0.2	21	1.0	25	1.3	26
env-diss	5292	15	0.3	22	**0.3**	23	0.3	22	1.3	25	1.5	25
env-diss	8232	10	0.6	26	0.4	22	**0.4**	18	1.7	22	1.8	22
env-diss	8232	15	1.0	31	0.6	29	**0.5**	23	3.2	35	2.6	35
env-diss	32,928	10	4.6	37	2.8	36	**1.9**	17	8.0	27	7.2	27
env-diss	32,928	14	4.8	28	**2.1**	22	2.5	19	10.0	27	7.7	27
phone	32,768	5	0.5	15	**0.4**	8	0.6	8	1.9	10	0.7	10
stormG2	1000	10	0.7	35	0.5	21	**0.3**	21	2.0	32	1.5	31
stormG2	1000	20	1.4	33	0.8	18	**0.5**	19	4.5	29	1.7	29
stormG2	1000	21	1.6	37	0.8	18	**0.5**	22	4.0	29	1.7	28

Shortest computing times are in bold

References

1. Andersen, E.D., Andersen, K.D.: Presolving in linear programming. Math. Program. **71**(2), 221–245 (1995). https://doi.org/10.1007/BF01586000
2. Andersen, E.D., Andersen, K.D.: The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm, pp. 197–232. Springer, Boston (2000). https://doi.org/10.1007/978-1-4757-3216-0_8
3. Anjos, M.F., Burer, S.: On handling free variables in interior-point methods for conic linear optimization. SIAM J. Optim. **18**(4), 1310–1325 (2008). https://doi.org/10.1137/06066847X
4. Anjos, M.F., Lodi, A., Tanneau, M.: A decentralized framework for the optimal coordination of distributed energy resources. IEEE Trans. Power Syst. **34**(1), 349–359 (2019). https://doi.org/10.1109/TPWRS.2018.2867476
5. Babonneau, F., Vial, J.P.: Accpm with a nonlinear constraint and an active set strategy to solve nonlinear multicommodity flow problems. Math. program. **120**(1), 179–210 (2009)
6. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numer. Math. **4**(1), 238–252 (1962). https://doi.org/10.1007/BF01386316
7. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. SIAM Rev. **59**(1), 65–98 (2017). https://doi.org/10.1137/141000671
8. Birge, J.R., Qi, L.: Computing block-angular karmarkar projections with applications to stochastic programming. Manag. Sci. **34**(12), 1472–1479 (1988)
9. Bixby, R.E., Gregory, J.W., Lustig, I.J., Marsten, R.E., Shanno, D.F.: Very large-scale linear programming: a case study in combining interior point and simplex methods. Oper. Res. **40**(5), 885–897 (1992). https://doi.org/10.1287/opre.40.5.885
10. Castro, J.: Interior-point solver for convex separable block-angular problems. Optim. Methods Softw. **31**(1), 88–109 (2016). https://doi.org/10.1080/10556788.2015.1050014

11. Castro, J., Nasini, S., Saldanha-da Gama, F.: A cutting-plane approach for large-scale capacitated multi-period facility location using a specialized interior-point method. Math. Program. **163**(1), 411–444 (2017). https://doi.org/10.1007/s10107-016-1067-6
12. Choi, I.C., Goldfarb, D.: Exploiting special structure in a primal–dual path-following algorithm. Math. Program. **58**(1), 33–52 (1993). https://doi.org/10.1007/BF01581258
13. Dantzig, G.B., Wolfe, P.: Decomposition principle for linear programs. Oper. Res., pp. 101–111. (1960). https://doi.org/10.1287/opre.8.1.101
14. Davis, T.A.: SuiteSparse: a suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html
15. Desaulniers, G., Desrosiers, J., Solomon, M.M.: Column generation, GERAD 25th anniversary, vol. 5, 1 edn. Springer Science & Business Media (2006)
16. Diamond, S., Boyd, S.: CVXPY: a python-embedded modeling language for convex optimization. J. Mach. Learn. Res. **17**(83), 1–5 (2016)
17. Domahidi, A., Chu, E., Boyd, S.: ECOS: An SOCP solver for embedded systems. In: European Control Conference (ECC), pp. 3071–3076 (2013)
18. Dunning, I., Huchette, J., Lubin, M.: Jump: a modeling language for mathematical optimization. SIAM Rev. **59**(2), 295–320 (2017)
19. Elhedhli, S., Goffin, J.L.: The integration of an interior-point cutting plane method within a branch-and-price algorithm. Math. Program. **100**(2), 267–294 (2004)
20. Forrest, J., Vigerske, S., Ralph, T., Hafer, L., jpfasano, Santos, H.G., Saltzman, M., Gassmann, H., Kristjansson, B., King, A.: coin-or/Clp: version 1.17.6 (2020). https://doi.org/10.5281/zenodo.3748677
21. Friedlander, M.P., Orban, D.: A primal-dual regularized interior-point method for convex quadratic programs. Math. Program. Comput. **4**(1), 71–107 (2012). https://doi.org/10.1007/s12532-012-0035-2
22. Gertz, E.M., Wright, S.J.: Object-oriented software for quadratic programming. ACM Trans. Math. Softw. **29**(1), 58–81 (2003). https://doi.org/10.1145/641876.641880
23. Gleixner, A.M., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. INFORMS J. Comput. **28**(3), 449–464 (2016). https://doi.org/10.1287/ijoc.2016.0692
24. Gondzio, J.: Multiple centrality corrections in a primal-dual method for linear programming. Comput. Optim. Appl. **6**(2), 137–156 (1996). https://doi.org/10.1007/BF00249643
25. Gondzio, J.: Presolve analysis of linear programs prior to applying an interior point method. INFORMS J. Comput. **9**(1), 73–91 (1997). https://doi.org/10.1287/ijoc.9.1.73
26. Gondzio, J.: Interior point methods 25 years later. Eur. J. Oper. Res. **218**(3), 587–601 (2012). https://doi.org/10.1016/j.ejor.2011.09.017
27. Gondzio, J., Gonzalez-Brevis, P., Munari, P.: New developments in the primal-dual column generation technique. Eur. J. Oper. Res. **224**(1), 41–51 (2013). https://doi.org/10.1016/j.ejor.2012.07.024
28. Gondzio, J., González-Brevis, P., Munari, P.: Large-scale optimization with the primal-dual column generation method. Math. Program. Computation **8**(1), 47–82 (2016). https://doi.org/10.1007/s12532-015-0090-6
29. Gondzio, J., Grothey, A.: Parallel interior-point solver for structured quadratic programs: application to financial planning problems. Ann. Oper. Res. **152**(1), 319–339 (2007). https://doi.org/10.1007/s10479-006-0139-z
30. Gondzio, J., Grothey, A.: Exploiting structure in parallel implementation of interior point methods for optimization. Comput. Manag. Sci. **6**(2), 135–160 (2009). https://doi.org/10.1007/s10287-008-0090-3
31. Gondzio, J., Sarkissian, R.: Column generation with a primal-dual method. Technical report 96.6, Logilab (1996). https://www.maths.ed.ac.uk/~gondzio/reports/pdcgm.pdf
32. Gondzio, J., Sarkissian, R.: Parallel interior-point solver for structured linear programs. Math. Program. **96**(3), 561–584 (2003). https://doi.org/10.1007/s10107-003-0379-5
33. Gondzio, J., Sarkissian, R., Vial, J.P.: Using an interior point method for the master problem in a decomposition approach. Eur. J. Oper. Res. **101**(3), 577–587 (1997). https://doi.org/10.1016/S0377-2217(96)00182-8
34. Grothey, A., Hogg, J., Colombo, M., Gondzio, J.: A Structure Conveying Parallelizable Modeling Language for Mathematical Programming, pp. 145–156. Springer, New York (2009). https://doi.org/10.1007/978-0-387-09707-7_13
35. Gurobi Optimization, L.: Gurobi optimizer reference manual (2018). https://www.gurobi.com

36. Hart, W.E., Watson, J.P., Woodruff, D.L.: Pyomo: modeling and solving mathematical programs in python. Math. Program. Comput. **3**(3), 219–260 (2011)
37. Hurd, J.K., Murphy, F.H.: Exploiting special structure in primal dual interior point methods. ORSA J. Comput. **4**(1), 38–44 (1992). https://doi.org/10.1287/ijoc.4.1.38
38. IBM: IBM ILOG CPLEX Optimization Studio. https://www.ibm.com/products/ilog-cplex-optimization-studio
39. Jessup, E.R., Yang, D., Zenios, S.A.: Parallel factorization of structured matrices arising in stochastic programming. SIAM J. Optim. **4**(4), 833–846 (1994). https://doi.org/10.1137/0804048
40. Kelley Jr., J.: The cutting-plane method for solving convex programs. J. Soc. Ind. Appl. Math. **8**(4), 703–712 (1960). https://doi.org/10.1137/0108053
41. Legat, B., Dowson, O., Garcia, J.D., Lubin, M.: MathOptInterface: a data structure for mathematical optimization problems. (2020). arXiv:2002.03447 [math]
42. Löfberg, J.: Yalmip : A toolbox for modeling and optimization in matlab. In: Proceedings of the CACSD Conference. Taipei, Taiwan (2004)
43. Lubin, M., Petra, C.G., Anitescu, M., Zavala, V.: Scalable stochastic optimization of complex energy systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 64:1–64:64. ACM, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063470
44. Ma, D., Saunders, M.A.: Solving multiscale linear programs using the simplex method in quadruple precision. In: Al-Baali, M., Grandinetti, L., Purnama, A. (eds.) Numerical Analysis and Optimization, pp. 223–235. Springer International Publishing, Cham (2015)
45. Makhorin, A.: GNU Linear Programming Kit, version 4.64 (2017). https://www.gnu.org/software/glpk/glpk.html
46. Mehrotra, S.: On the implementation of a primal-dual interior point method. SIAM J. Optim. **2**(4), 575–601 (1992). https://doi.org/10.1137/0802028
47. Mitchell, J.E.: Cutting plane methods and subgradient methods. In: Decision Technologies and Applications, chap. 2, pp. 34–61. INFORMS (2009). https://doi.org/10.1287/educ.1090.0064
48. Mitchell, J.E., Borchers, B.: Solving Linear Ordering Problems with a Combined Interior Point/Simplex Cutting Plane Algorithm, pp. 349–366. Springer, Boston (2000). https://doi.org/10.1007/978-1-4757-3216-0_14
49. MOSEK ApS: The MOSEK Optimization Suite. https://www.mosek.com/
50. Munari, P., Gondzio, J.: Using the primal-dual interior point algorithm within the branch-price-and-cut method. Comput. Oper. Res. **40**(8), 2026–2036 (2013). https://doi.org/10.1016/j.cor.2013.02.028
51. Naoum-Sawaya, J., Elhedhli, S.: An interior-point benders based branch-and-cut algorithm for mixed integer programs. Ann. Oper. Res. **210**(1), 33–55 (2013)
52. Orban, D., contributors: LDLFactorizations.jl: Factorization of symmetric matrices. https://github.com/JuliaSmoothOptimizers/LDLFactorizations.jl (2020). https://doi.org/10.5281/zenodo.3900668
53. Rousseau, L.M., Gendreau, M., Feillet, D.: Interior point stabilization for column generation. Oper. Res. Lett. **35**(5), 660–668 (2007). https://doi.org/10.1016/j.orl.2006.11.004
54. Schultz, G.L., Meyer, R.R.: An interior point method for block angular optimization. SIAM J. Optim. **1**(4), 583–602 (1991). https://doi.org/10.1137/0801035
55. Tanneau, M.: Tulip.jl (2020). https://doi.org/10.5281/zenodo.3787950. https://github.com/ds4dm/Tulip.jl
56. Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in Julia. In: Proceedings of the 1st Workshop for High Performance Technical Computing in Dynamic Languages, pp. 18–28. IEEE Press (2014)
57. Westerlund, T., Pettersson, F.: An extended cutting plane method for solving convex minlp problems. Comput. Chem. Eng. **19**, 131–136 (1995). https://doi.org/10.1016/0098-1354(95)87027-X
58. Wright, S.: Primal–Dual Interior-Point Methods. Society for Industrial and Applied Mathematics (1997). https://doi.org/10.1137/1.9781611971453
59. Xu, X., Hung, P.F., Ye, Y.: A simplified homogeneous and self-dual linear programming algorithm and its implementation. Ann. Oper. Res. **62**(1), 151–171 (1996). https://doi.org/10.1007/BF02206815