NASOQ: Numerically Accurate Sparsity-Oriented QP Solver

KAZEM CHESHMI, University of Toronto, Canada
DANNY M. KAUFMAN, Adobe Research, USA
SHOAIB KAMIL, Adobe Research, USA
MARYAM MEHRI DEHNAVI, University of Toronto, Canada

Quadratic programs (QP), minimizations of quadratic objectives subject to linear inequality and equality constraints, are at the heart of algorithms across scientific domains. Applications include fundamental tasks in geometry processing, simulation, engineering, animation and finance where the accurate, reliable, efficient, and scalable solution of QP problems is critical. However, available QP algorithms generally provide either accuracy or scalability – but not both. Some algorithms reliably solve QP problems to high accuracy but work only for smaller-scale QP problems due to their reliance on dense matrix methods. Alternately, many other QP solvers scale well via sparse, efficient algorithms but cannot reliably deliver solutions at requested accuracies. Towards addressing the need for accurate *and* efficient QP solvers at scale, we develop NASOQ, a new, full-space QP algorithm that provides accurate, efficient, and scalable solutions for QP problems. To enable NASOQ we construct a new row modification method and fast implementation of LDL factorization for indefinite systems. Together they enable efficient updates and accurate solutions of the iteratively modified KKT systems required for accurate QP solves. While QP methods have been previously tested on large synthetic benchmarks, to test and compare NASOQ's suitability for real-world applications we collect here a new benchmark set comprising a wide range of graphics-related QPs across physical simulation, animation, and geometry processing tasks. We combine these problems with numerous pre-existing stress-test QP benchmarks to form, to our knowledge, the largest-scale test set of application-based QP problems currently available. Building off of our base NASOQ solver we then develop and test two NASOQ variants against best, state-of-the-art available QP libraries – both commercial and open-source. Our two NASOQ-based methods each solve respectively 98.8% and 99.5% of problems across a range of requested accuracies from 10^{-3} to 10^{-9} with average speedups ranging from 1.7× to 24.8× over fastest competing methods.

CCS Concepts: • **Mathematics of computing** → **Quadratic programming**; **Solvers**; **Computations on matrices**; • **Computing methodologies** → *Physical simulation.*

Additional Key Words and Phrases: Sparse Linear Algebra, Sparse Row Modification, Quadratic Programming, Contact Simulation, Mesh Deformation, Optimization

ACM Reference Format:
Kazem Cheshmi, Danny M. Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. 2020. NASOQ: Numerically Accurate Sparsity-Oriented QP Solver. 1, 1 (May 2020), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Solving a quadratic program (QP) is a core numerical task critical in domains spanning geometry processing [Dvorožňák et al. 2018; Sýkora et al. 2014; Zhu et al. 2018], animation [Jacobson et al. 2011; Righetti and Schaal 2012], physical simulation [Barbic 2007; Erleben 2013], robotics [Pandala et al. 2019], machine learning [Agrawal et al. 2019; Amos and Kolter 2017], engineering, and design [Fesanghary et al. 2008]. Unfortunately, available QP solvers are often neither accurate nor robust enough for many applications [Kaufman et al. 2008; Smith et al. 2012; Yao et al. 2017; Zheng and James 2011; Zhu et al. 2018], necessitating heuristics, approximations and/or multiple failsafe backups to succeed.

A long-standing challenge then has been to provide a single, unified QP solver that is 1) accurate, 2) efficient, and 3) scalable. By accurate we mean that the QP solver converges to all reasonable requested accuracies; by efficient we mean that it converges rapidly in wall-clock time; and by scalable we mean that it efficiently converges across both large- and small-sized QP instances. As we show in Section 6, available QP solver libraries generally succeed for some subsets of QPs, while often failing or becoming impractically slow to achieve success for others. To make matters worse, in many cases, given the algorithms employed, it is not possible to predict in advance when a QP method will succeed or fail per QP problem instance [Zheng and James 2011].

The key challenge for solving a QP is in identifying an *active set* [Fletcher 2013]. An active set is a subset of a QP's linear inequality constraints that are treated as equalities at optimality. All other inequalities can then effectively be safely ignored. If an active set is found, a QP problem instance then reduces to solving a much easier QP subject to just its active constraints set as equalities.

Algorithms for solving large-scale QPs generally treat the entire constraint set as approximately "active" with barrier terms penalizing all constraint violations simultaneously. This allows the application of large-scale, general-purpose sparse linear solvers, but generally comes at the cost of uncertainty in the active set and degraded solution accuracy. On the other hand, to address accuracy, many other QP algorithms employ *active-set methods.* These are a range of methods that iteratively explore and test active-set proposals. Details vary across methods but in all cases each iteration requires solving large numbers of reduced QPs. Each reduced QP is solved subject to a different set of proposed active constraints treated as equalities. In turn, solving these many reduced QPs accurately and efficiently is the computational crux of active-set methods. This amounts to solving at each instance an indefinite linear system for equality constrained optimality conditions – a Karush-Kuhn-Tucker (KKT) system [Boyd and Vandenberghe 2004; Fletcher 2013]. Current solutions employed rely either on accurate linear solvers that work well for small systems but are too expensive for repeated solves of new large, sparse problems, or else rely on less expensive

Authors' addresses: Kazem Cheshmi, kazem@cs.toronto.edu, University of Toronto, Toronto, Ontario, Canada; Danny M. Kaufman, kaufman@adobe.com, Adobe Research, Seattle, Washington, USA; Shoaib Kamil, kamil@adobe.com, Adobe Research, New York, New York, USA; Maryam Mehri Dehnavi, mmehride@cs.toronto.edu, University of Toronto, Toronto, Ontario, Canada.

2020. XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

but also less accurate methods for solving linear systems that once again unacceptably reduce accuracy [Stellato et al. 2020].

To address these issues we construct the Numerically Accurate Sparsity-Oriented QP Solver (NASOQ), a new, general-purpose, active-set algorithm for the accurate, efficient, and scalable solution of QPs. NASOQ is built upon three core contributions:

- LBL: a new LDL factorization algorithm for the fast, accurate factorization and update of sparse symmetric indefinite systems including those that arise in KKT problems;
- SoMod: a new sparsity-oriented row modification method that enables fast factorization for KKT matrix changes via efficient updates of previously computed factors; and
- Two new QP solvers that extend the Goldfarb-Idnani (GI) active-set strategy [Goldfarb and Idnani 1983] by application of LBL and SoMod to enable user-exposed trade-offs between speed and accuracy for large and sparse QP problems.

SoMod is a new algorithm designed to enable the rapid and accurate solutions of the many successively-updated KKT systems encountered during active-set QP solves. At start of QP solves, SoMod performs an initial symbolic analysis of a KKT system containing all constraints, then utilizes this information for both the initial factorization (which includes only the equality constraints) as well as subsequent factorizations (which include proposed active sets). By precomputing symbolic information, SoMod enables efficiently updating the factorization when constraints are added or removed from the proposed active set.

To compute each initial indefinite factorization for SoMod, we construct LBL, a novel implementation of the LDL factorization using Load-Balanced Level Coarsening [Cheshmi et al. 2018] for parallelization. LBL provides state-of-the-art performance for solving indefinite KKT systems while enabling precomputation of the required symbolic analysis for subsequent factorization updates.

With these building blocks in place we construct and analyze NASOQ via a pair of new active-set QP algorithms. To consistently evaluate NASOQ with both prior and future QP methods we also introduce a new benchmark set composed of both practical, real-world stress-test QPs taken from a wide range of geometry, simulation, and design applications as well as prior QP benchmarks [Jacobson et al. 2011, 2018; Levin 2019; Maros and Mészáros 1999; Segata 2019; Weidner et al. 2018]. As we demonstrate in Section 6, across a range of requested accuracies in this benchmark NASOQ obtains consistent accuracy by converging for 99.5% of benchmark problems while the best convergence across competing state-of-the-art solvers is 94%. At the same time NASOQ remains most efficient by providing an average speedup of 1.7×–24.8× across requested accuracy ranges compared to the fastest competing times across compared QP solvers. Please see Section 6 for details of our analysis.

2 PROBLEM STATEMENT AND PRELIMINARIES

We focus on the solution of convex quadratic programming problems to find the linearly constrained minimizers of quadratic energies. In full generality our problem then is

$$\min_x \ \frac{1}{2}\, x^T H x + q^T x \ \text{ s.t. } \ Ax = b,\ Cx \le d \tag{1}$$

where the unknown minimizer $x \in \mathbb{R}^n$ is constrained by linear equality constraints $Ax = b$ and inequality constraints $Cx \le d$. Note that in many cases we may have only inequality or equality constraints. However, in the following, without loss of generality, we consider the full mixed case. Here the symmetric matrix H is, either by construction or standard user regularization [Golub and Van Loan 2012; Schenk and Gärtner 2006], a positive-definite matrix. The QP in (1) is then strictly convex. In applications, the matrices H, A, and C are often large and sparse. By sparse we mean that the majority of matrix entries are zero, e.g., we have an average of 98% zero entries in our benchmark examples.

Unlike the solution of symmetric linear systems (or equivalently, unconstrained quadratic energies) the optimality conditions, and thus the accuracy of a QP solution, are much more complex to evaluate. Optimality of (1) is given by the specialized Karush-Kuhn-Tucker (KKT) conditions[1] [Fletcher 2013; Wong 2011]

$$\begin{aligned} Hx + q + A^T y + C^T z &= 0 \\ Ax - b &= 0 \\ 0 \le z \perp\ d - Cx &\ge 0. \end{aligned} \tag{2}$$

where y and z are the QP problem's Lagrange multipliers [Wong 2011].

2.1 Accuracy

Applications require controllable quality and thus controllable accuracy for solutions to the QP problem. The accuracy of a QP solution is evaluated by reduction of four corresponding measures[2]

$$\text{Primal-feasibility: } \left\| \left((Ax-b)^T, (\max(\mathbf{0}, Cx-d))^T \right)^T \right\| < \epsilon_f, \tag{3}$$

$$\text{Stationarity: } \|Hx + q + A^T y + C^T z\| < \epsilon_s, \tag{4}$$

$$\text{Complementarity: } \|z \odot (Cx - d)\| < \epsilon_c, \tag{5}$$

$$\text{Non-negativity: } \|\min(\mathbf{0}, z)\| < \epsilon_n. \tag{6}$$

In the following we design NASOQ and analyze QP methods on their ability to drive all four of these measures (∞-norm) below a common, maximum error threshold accuracy: $\epsilon \ge \max(\epsilon_f, \epsilon_s, \epsilon_c, \epsilon_n)$. While necessary accuracies for each of the four measures certainly change per application, a desirable goal for a general-purpose QP algorithm is to solve every reasonable problem to any requested accuracy. Here we design for general-purpose QP problems and so do not predict a priori what measures are most important. Thus we evaluate fitness by asking each solve to drive all measures below ϵ.

Primal-feasibility measures constraint satisfaction. Applying the ∞-norm gives the worst violation of the enforced constraints by a given solution. In many applications constraints are invariants that need to be satisfied such as positive volume, non-penetration, or structural feasibility. Errors in constraint satisfaction lead to unacceptable failures and constraint drift in applications that depend on constraint resolution at each call of a QP solve.

Stationarity measures the balance between energy and constraint gradients. This is critical for stable and accurate solutions. For example, in structural engineering applications stationarity measures

[1] Here $x \perp y$ is the *complementarity condition* $x_i y_i = 0$, $\forall$ corresponding entries i in vectors x and y.
[2] Here $\odot$ is the *Hadamard* (element-wise) product.

Algorithm 1: Dual-feasible active-set QP solver.

```
Data: H, q, A, b, C, d
Result: x, y, z
/* Feasibility phase */
1  Initialize z_0 = 0; k = 0; active-set=∅;
2  Solve Equation 7 to initialize x_0, y_0;
/* Optimality phase */
3  while x_k is not primal-feasible do
4      Solve Equation 8 to compute descent Δx,Δy,Δz;
5      Compute the step length t;
6      if t = ∞ then
7          Problem is unbounded.;
8      else
9          Update x_{k+1}, y_{k+1}, z_{k+1} with Δx, Δy, Δz;
10         Update the active set;
11         Update the KKT system with the updated active set;
12     end
13     k = k + 1 ;
14 end
15 x_k, y_k, z_k are optimal;
```

how well force balance is modeled, while in dynamic simulations stationarity measures how well the equations of motion are satisfied. In many applications even small residuals of stationarity with respect to measured dimensions of the systems can lead to simulation instabilities and blow-ups and/or unacceptable modeling errors for engineering applications.

Complementarity measures the pairwise products of dual variables and their corresponding inequality constraints and is critical for correctly capturing active sets. For example, in multi-body simulations [Erleben 2013; Heyn et al. 2013] dual variables often represent contact forces while constraints model intersections. Complementarity then encodes the property that contact forces cannot be applied unless objects are touching. Large violations of complementarity can create instabilities and visual artifacts of floating bodies with contact forces artificially applying action at a distance.

Non-negativity then ensures that dual variables are positive. Negative dual variables likewise have serious consequences for stability and quality in applications. Consider, for example, bounded biharmonic weights for deformation skinning are computed via QP solves [Jacobson et al. 2011] with resultant weights requiring non-negativity; while similarly for contact problems negative dual variables indicate an unacceptable violation of the "no-velcro" condition – that contact forces should push but not pull [Smith et al. 2012].

2.2 Active-Set KKT System Solutions

We focus on enabling scalable, efficient, and accurate solutions for QPs at all scales. For a given input QP we seek an as-efficient-as-possible solver that will obtain a user-requested accuracy. While state-of-the-art barrier and first-order QP methods are promising for scaling to large QP problems, their solutions suffer from degraded accuracy [Stellato et al. 2020] and no general method exists for determining a priori when they will succeed or fail in reaching the requested accuracy [Boyd et al. 2011]. On the other end of the spectrum, active-set QP methods provide high-accuracy QP solutions. However, in order for active-set QP algorithms to reach a targeted accuracy they must also accurately solve a large number of successive indefinite linear systems visited by the algorithm at each inner iteration, which can be computationally expensive.

Active-set methods start with a feasible solution and keep a running set of proposed *active* inequality constraints $\mathcal{W}$ to reach the optimal solution while maintaining feasibility conditions. Active-set methods are then either primal-feasible, preserving the primal-feasibility condition or else are dual-feasible, preserving the non-negativity condition. Here we focus on the Goldfarb-Idnani (GI) [Goldfarb and Idnani 1983] strategy. GI is a dual-feasible active-set approach and so enables direct and inexpensive initialization [Wong 2011].

The high-level pseudocode for the GI algorithm is shown in Algorithm 1. The GI algorithm begins (lines 1–2) by initializing an empty active-set proposal, $\mathcal{W} = \emptyset$ with zero dual variables, $z_0 = 0$. The resulting initial KKT system to solve is then the indefinite linear system,

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} -q \\ b \end{bmatrix} \tag{7}$$

which corresponds to solving a *feasible* QP with just equality constraints applied.

Then, each successive iteration of the GI method (lines 3–14, Algorithm 1) improves the last iterate's solution by updating the active-set proposal $\mathcal{W}$ and so the corresponding active-set constraint matrix $C_{\mathcal{W}}$ and the right-hand side constraint vector c_w. The GI method updates the active set by only adding one or removing one constraint in each successive iteration. Here w is the activated constraint.

The next descent direction for the QP is then determined by solving the *updated* KKT system

$$\begin{bmatrix} H & A^T & C_{\mathcal{W}}^T \\ A & 0 & 0 \\ C_{\mathcal{W}} & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} c_w \\ 0 \\ 0 \end{bmatrix} \tag{8}$$

The dual and primal variables of the next iteration are then updated by finding step lengths along the computed descent directions. The step lengths ensure that the activated constraint becomes primal-feasible and all dual variables remain dual-feasible. Thus, in each iteration, both the dual and primal variables corresponding to the constraints in the active set are both non-negative and primal-feasible. Each iteration's linear solve of the updated indefinite KKT system in (8) becomes increasingly expensive as QP system sizes and constraint numbers grow. However, at the same time, the GI algorithm requires accurate solutions for each of these successive KKT systems for algorithmic stability and in order to consistently obtain accurate solutions for the overall QP problem [Powell 1985].

A key observation then is that each update to $\mathcal{W}$ and correspondingly to the matrix in (8) is small and specifically requires the update of just a single row in $C_{\mathcal{W}}$. Currently, active-set algorithms leverage this observation with indirect methods [Gould 2006] that solve the KKT system by adaptively updating it with respect to $\mathcal{W}$ via the application of the QR decomposition and the Schur complement.

While indirect methods provide accurate and efficient KKT solutions at small scales, they are unable to take advantage of sparsity. These methods suffer from extensive fill-in[3] due to the QR factorization and the Schur complement form and so do not scale due to slow compute times and memory overhead for QP problems with large numbers of variables and/or large numbers of constraints.

Alternately direct active-set QP methods form the KKT system explicitly and solve it via direct or iterative linear solvers. Direct methods solve each iteration's KKT system via indefinite factorization methods [Maes 2011] followed by a solve stage. This leads to accurate and scalable solutions but is inefficient due to the repeated cost of recomputing factorizations. Application of iterative methods, e.g., Krylov subspace methods [Gould et al. 2001], in place of direct solves are not typically performed as it remains challenging to find effective, general-purpose preconditioners for KKT matrices with generic active sets [Maes 2011]. Re-purposing prior factorizations to compute the solution of a modified KKT system is thus a highly attractive direction for combined efficiency and accuracy. However, to our knowledge, no previous solution for indefinite matrix factorization updates exists. The closest possible option we find is CHOLMOD row modification [Davis and Hager 2005], which is an efficient and effective solution designed for symmetric positive definite (SPD) matrices but does not provide accurate and stable solution updates for indefinite KKT systems.

In this work, we focus on enabling accurate, scalable, and efficient QP solutions by taking advantage of sparsity and by efficiently updating factorizations of the active-set method's indefinite KKT system. In doing so we address the gap between direct and indirect methods. We develop NASOQ to combine the advantages of leveraging direct, accurate solutions of KKT systems with the small and localized updates of subsequent KKT systems. NASOQ leverages our new SoMod method which enables efficient, sparsity preserving updates of existing factorizations after each new constraint update to $\mathcal{W}$ and, as we show in Section 6, enables the application of accurate direct factorization methods across a wide range of large- and small-scale QP problems not previously possible.

3 RELATED WORK

QP solvers. QP algorithms can be categorized into three broad classes of methods: barrier (primarily interior-point), first-order, and active-set.

Barrier methods. Barrier methods [Domahidi et al. 2013; Gertz and Wright 2003; Gondzio and Grothey 2006; Mattingley and Boyd 2012; Mosek 2015; Optimization 2014; Pandala et al. 2019; Waltz and Nocedal 2004] apply weighted barrier functions in the objective to enforce inequality constraints, converting inequality constrained QPs into equality-constrained nonlinear problems that can be solved by Newton or quasi-Newton methods [Boyd and Vandenberghe 2004; El-Bakry et al. 1996]. Performing a series (homotopy) of progressively tighter and thus more challenging barrier solves leads to interior-point and related methods [Boyd and Vandenberghe 2004]. As unconstrained optimization methods can then be directly applied, barrier methods can leverage sparse linear methods and so scale to large systems. Thus popular, a wide range of commercial [Mosek 2015; Optimization 2014] and open-source [Gertz and Wright 2003; Wächter and Biegler 2006] interior-point solvers are available. However, accurate solutions are challenging to obtain for barrier methods – especially as system and constraint sizes grow. As accuracy is tightened, barrier solvers generally require increasingly large numbers of iterations. In turn, each iteration necessitates the expensive solution of a new, large-scale linear system.

First-order method: OSQP. Barrier methods leverage second-order expansions of constraint information via Newton-type methods which can be expensive for computation per iterate. On the other hand, per-iteration efficient methods can be constructed by leveraging first-order strategies. In particular, operator splitting via the alternating direction method of multipliers (ADMM) has been recently applied to design OSQP [Stellato et al. 2020], an efficient, highly scalable, first-order QP algorithm. OSQP forms all constraints into a single, large saddle-point-like system and then re-applies the solution of this system in each successive ADMM iteration to update primal and dual terms. OSQP thus can take advantage of lightweight computations per iteration and scales well to large, sparse QP problems. However, consistent with first-order strategies it can be slow or even unable to reach accurate solutions for larger and more challenging QP problems.

Active-set methods. Active-set methods [Ferreau et al. 2014; Gill et al. 2005, 1991; Maes 2011; Schittkowski 2003] start with an initial feasible solution and then iterate to obtain the optimal solution while maintaining feasibility. After finding the initial feasible solution, which is either primal- or dual-feasible depending on the method, active-set methods look for the optimal active-set by solving successive KKT systems that include all constraints in the current active-set. Solving these KKT systems is the most expensive part in these methods. Active-set methods are divided into direct and indirect based on how they solve KKT systems [Benzi et al. 2005]. Indirect methods, known as range-space [Goldfarb and Idnani 1983] and null-space [Gill et al. 2005] methods, solve the KKT system using a Cholesky factorization along with a QR or Schur complement. Although these techniques provide an accurate solution, they do not preserve sparsity and thus do not scale for large QP problems due to high memory usage and extensive computations in the QR and Schur complement factorization. *Full-space methods* [Gould et al. 2003; Huynh 2008], on the other hand, are direct active-set QP methods that work directly with the KKT system. Solving the KKT system using an iterative algorithm such as a Krylov subspace method [Gould et al. 2001] for active-set methods requires finding an efficient preconditioner for any arbitrary active-set which is often difficult [Maes 2011]. Factorizing the KKT system using a direct method is very expensive and thus existing full-space techniques build an augmented system, along with an initial KKT, to compute the solution of the KKT system via the Schur complement [Gould et al. 2003] or Block-LU [Huynh 2008]. Both of these methods require large amounts of storage thus limiting their scalability and efficiency. Nevertheless, full-space methods have a promising property – they preserve the sparsity pattern of the system. In this work we leverage this sparsity to efficiently re-use factors across iterations. We introduce a new, full-space active-set

[3] *Fill-ins* are additional nonzeros created in the factor during a matrix factorization.

algorithm, based on the Goldfarb-Idnani active-set strategy, that directly factorizes the successive KKT systems with SoMod and LBL to enable sparsity-oriented row modification and indefinite factorization of the successive KKT systems across our full-space QP solver's iterations.

LDL factorization. Using direct factorization methods for solving a linear system of equations is common in many computer graphics applications [Herholz and Alexa 2018; Herholz et al. 2017; Yeung et al. 2016] and is a subroutine in full-space QP solvers. A number of existing factorization methods are designed for solving a sparse SPD system of equations [Chen et al. 2008; Cheshmi et al. 2017, 2018; Herholz and Alexa 2018]. These methods use a square-root based Cholesky factorization [Davis 2006; Golub and Van Loan 2012] that will fail with symmetric indefinite systems from negative values under the square root when factorizing diagonals. Applying these solvers with regularization [Herholz and Alexa 2018] prevents these types of failures but can introduce significant inaccuracies to problem solutions. Some existing indefinite factorization methods are square-root free [Golub and Van Loan 2012] but are slow, e.g., Suitesparse's LDL [Davis 2019] which is a single-thread implementation. Parallel indefinite solvers such as MKL Pardiso [Wang et al. 2014], the standalone Pardiso solver [Schenk and Gärtner 2002, 2006] and MA57 [Duff 2004; Hogg and Scott 2013] provide fast factorizations but do not support factor modifications for when a row/column is changed. We introduce LBL, a new, parallel, indefinite, square-root free solver with pivoting, that additionally enables modifying already-computed factors efficiently. LBL extends the parallelism strategy from Cheshmi et al. [2018] from SPD to indefinite factorizations where the now-required pivoting introduces new dependencies.

L-factor modification. Modifying the L-factor to avoid re-factoring a symmetric matrix after small changes [Davis and Hager 1999, 2009] is a critical task in computer graphics [Hecht et al. 2012; Herholz and Alexa 2018], circuit simulation [Davis and Hager 2005; Hager 1989], and optimization [Davis and Hager 2005]. In all such applications, the linear system of equations changes either by a rank update/-downdate (adding or subtracting the outer product of a row by itself) or a row modification. Existing modification methods [Davis and Hager 1999; Herholz and Alexa 2018] are generally designed to perform rank update/downdate ($A + ww^T$) on SPD matrices. These modification methods are then not applicable to our active-set QP solver where a row/column is modified for a symmetric indefinite system in each iteration. In turn, to our knowledge the only existing system with sparse row modification is CHOLMOD [Davis and Hager 2005] which is an efficient solution designed for SPD matrices. Thus CHOLMOD row modification does not provide accurate and stable solution updates for indefinite KKT systems; see Section 6.5. We propose SoMod row modification to leverage the sparsity pattern of constraint row updates and so accurately and efficiently modify the L-factor of indefinite factorization.

4 SOMOD: SPARSITY-ORIENTED ROW MODIFICATION

A scalable solution to a dual-feasible active-set QP requires an efficient solution to the successive KKT systems in Equations 7 and 8. This section discusses SoMod, a novel method for efficiently solving these KKT systems using the combination of a novel sparsity-oriented row modification method, a novel implementation of LDL factorization, and an efficient triangular solve. SoMod consists of two phases: an initialization phase associated with Equation 7 and a factor modification phase associated with Equation 8. In both phases, SoMod solves $Kx = s$ for x where s is a dense vector and K is a sparse symmetric indefinite KKT matrix. At the start of each QP solve we initialize the KKT matrix with the subsystem corresponding to applying just the equality constraints, so that:

$$K = \begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \tag{9}$$

where H and A are respectively the matrices for the quadratic objective and equality constraints. In order to solve the system $Kx = s$, SoMod applies LDL factorization to decompose the matrix K into

$$K = P_{fill} P_S (LDL^T + E) P_S^T P_{fill}^T \tag{10}$$

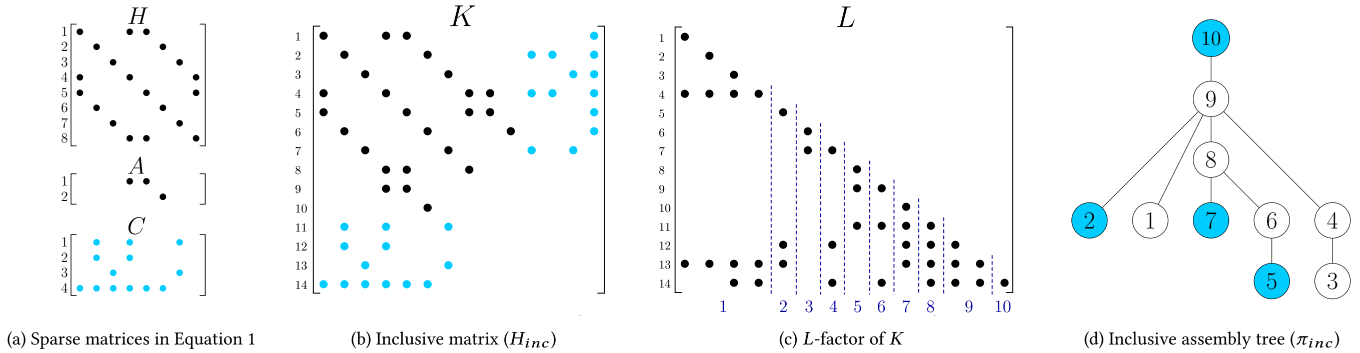
where D is a blocked diagonal symmetric matrix (due to our use of Bunch-Kaufman pivoting [Schenk and Gärtner 2006]), L is a sparse lower triangular matrix, E is a diagonal perturbation matrix (necessary to avoid zero diagonals, which can cause instabilities [Hogg and Scott 2013; Schenk and Gärtner 2006]), P_{fill} is a fill-reducing ordering (such as METIS [Karypis 1997]), and P_S is reordering due to pivoting. Given this factorization of the matrix, SoMod then uses L and D along with s (the right-hand side) to quickly compute the solution x via triangular solve.

The overall process of the factorization in this initialization phase of SoMod closely follows that of standard sparse linear system solvers. For efficient factorization, the sparsity pattern of K is analyzed during *symbolic analysis*. Symbolic analysis uses the sparsity pattern of K to construct *symbolic information*, which consists of the fill-reducing ordering P_{fill} and the sparsity pattern of L. Symbolic information guides the *numeric factorization*, which operates on the actual numeric values of K to compute the nonzero values of L and D. Unlike prior work, SoMod applies symbolic analysis in a way that allows the results to be reused during the modification phase. The initialization phase also includes *permutation* with P_{fill}, constraint-aware super-node creation, *perturbation* with E, and a restricted *pivoting* strategy with P_S; all of these steps are described in Section 4.1.

The modification phase in SoMod, described in Section 4.2, iteratively solves each new, updated KKT system which contains additional active constraints. During this phase, SoMod solves $Kx = s$ for each updated K:

$$K = \begin{bmatrix} H & A^T & C_{\mathcal{W}}^T \\ A & 0 & 0 \\ C_{\mathcal{W}} & 0 & 0. \end{bmatrix} \tag{11}$$

Here, for each update, C_w contains rows from the full constraint matrix corresponding to the current proposed active constraint set. Rather than solving each of these systems from scratch, SoMod updates the starting solution in our initialization phase using factor modification. Specifically, SoMod updates the symbolic information



(a) Sparse matrices in Equation 1 (b) Inclusive matrix (H_{inc}) (c) L-factor of K (d) Inclusive assembly tree (π_{inc})

Fig. 1. The symbolic initialization phase of SoMod starts with creating an inclusive matrix, shown in Figure 1b from the matrices in Figure 1a which are inputs to the QP problem in Equation 1. The inclusive matrix is then permuted with a fill-reducing permutation to compute the sparsity pattern of the L-factor with minimum number of fill-ins. The sparsity pattern of the L-factor of the inclusive matrix in Figure 1b is computed and shown in Figure 1c. Boundaries of Supernodes are shown with dotted lines and supernode numbers are illustrated below the L-factor. The corresponding inclusive (assembly) tree of the L-factor in Figure 1c is shown in Figure 1d. The colored nodes correspond to the inequality constraint rows (matrix C in Figure 1a). The constraint-aware supernode creation strategy ensures that supernodes corresponding to the inequality constraint nodes contain only a single column. The colored nodes of the inclusive tree are removed to create the pruned inclusive tree passed to numerical factorization along with the L-factor in Figure 1c.

from the initialization phase, followed by updating the numeric L-factor to account for the added/removed constraints in each iteration. Then, a triangular solve is once again used to find x given the updated L and D matrices.

4.1 Initialization Phase

In SoMod, the initialization phase produces symbolic information that can be reused by subsequent factorizations in the factor modification phase. After producing symbolic information, this phase then proceeds with numeric factorization, followed by triangular solve to return the solution to the KKT system.

4.1.1 Symbolic Analysis with the Inclusive Matrix. The initialization phase first builds an *inclusive matrix*. Then, we permute the inclusive matrix and generate symbolic information, including the sparsity pattern of the L-factor of the inclusive matrix, the assembly tree of the inclusive matrix (the *inclusive tree*), a pruned inclusive tree, and P_{fill} in Equation 10. This symbolic information collectively facilitates an efficient numeric factorization in the initialization phase and also provides symbolic information leveraged by the factor modification stage.

The inclusive matrix, the sparsity of L, and the assembly tree. An inclusive matrix is first assembled using the objective and equality constraint matrices (H and A) and the sparsity pattern of the inequality constraint matrix. That is, the inclusive matrix includes all entries of C but with values set to zero. The numerical values of the inequality constraint matrix will be added to the inclusive matrix during the modification stage of SoMod when a constraint is added. Figure 1b shows an example of an inclusive matrix created from the matrices in Figure 1a.

SoMod then builds an *elimination tree* [Davis 2006; Liu 1990] for the inclusive matrix, which enables obtaining the sparsity pattern of the L-factor and creating the inclusive assembly tree. The elimination tree of the inclusive matrix is a tree that expresses dependencies between operations on columns of the L-factor, dictating the order of factorization. Because of fill-ins, the sparsity pattern of the L-factor is different from that of the inclusive matrix. The number of fill-ins correlates with the number of operations in the factorization process. Thus, after creating the inclusive matrix, it will be permuted with P_{fill}, a fill-reducing ordering, which improves the speed of numeric factorization by reducing the number of operations in the factorization process.

Finally, the inclusive elimination tree and the sparsity pattern of its L-factor are used to create constraint-aware supernodes and the inclusive assembly tree, which is the supernodal version of the inclusive elimination tree. Supernodes [Davis 2006; Schenk and Gärtner 2006] are created by grouping columns with similar sparsity patterns if they form a chain in the elimination tree; that is, two consecutive columns are grouped together if one column is the only child of its next column. Thus each node in the assembly tree represents a group of columns together in a supernode; a node in the elimination tree represents just a single column. For example, the tree in Figure 1d is the assembly tree of the supernodal L-factor in Figure 1c; each node of the tree corresponds to a supernode and the numbers shown below the L-factor correspond to the supernode's number in the assembly tree. The parent of each node in the assembly tree is obtained using the row index of the first off-diagonal nonzero of its corresponding supernode in the L-factor. For example in Figure 1c, the row index of the first off-diagonal nonzero of supernode 3 is 4 and thus node 4 is the parent of node 3 in Figure 1d.

Sparse factorization can be more efficient when operating on supernodes instead of individual columns [Chen et al. 2008]. Supernode creation in SoMod is constraint-aware, so rows/columns of C are not grouped with each other or with other columns; this

makes it possible to add or remove constraints separately from one another while still allowing the rest of the assembly tree to benefit from the increased efficiency of the supernodal approach. For example, in the inclusive tree in Figure 1d, column 14 can form a supernode with columns 12 and 13; however, since column 14 is the fourth constraint in C, it is excluded from the supernode.

The pruned inclusive assembly tree. The inclusive assembly tree contains dummy entries for all inequality constraints. During this phase, and during row modification, we instead use a *pruned* inclusive assembly tree that contains only the entries corresponding to active inequality constraints; this is represented by an array of parents, denoted with π. Thus, before performing the initial factorization, we remove all dummy entries corresponding to inequality constraints. As an additional optimization, SoMod also creates a visibility vector v that shows whether a column of the L-factor should be visited during the initial numerical factorization phase; this information is derived from the pruned assembly tree, but in practice using the visibility vector can be faster than finding paths in the assembly tree. Because the initial KKT matrix only includes equality constraints, the visibility vector is initialized by setting all rows of the inclusive matrix that correspond to rows of C to invisible.

Constraint-aware supernode creation facilitates creating the pruned inclusive tree by ensuring every row of matrix C corresponds to one node in the inclusive tree. This allows removing rows by only changing the pruned inclusive tree as described in Section 4.2.1.

4.1.2 Numeric Factorization with LBL. Numeric factorization computes the nonzero values of the L-factor in solving Equation 7 (line 2 of Algorithm 1) using LBL, a modified LDL factorization algorithm that uses Load-Balanced Level Coarsening [Cheshmi et al. 2018], a scheduling technique that improves the performance of numeric factorization on parallel architectures. While prior work applied Load-Balanced Level Coarsening to Cholesky factorization for symmetric positive definite matrices, LBL extends the technique to symmetric indefinite matrices that arise from KKT problems.

Numeric factorization takes as input the sparsity pattern of the L-factor and the visibility vector, and first computes the *perturbation matrix* (E in Equation 10), using information from the inclusive matrix to enable a stable factorization. Perturbation ensures no zeros exist in the diagonal entries of the matrix, since these lead to division-by-zero during factorization. Numeric factorization then uses the pruned inclusive assembly tree to determine an efficient and correct order of computation, and then uses this schedule to compute the nonzero values of L, D, and P_S in Equation 10.

Perturbation. We add a small value to zero diagonals of the inclusive matrix that correspond to rows of the equality constraints. Since the location of the equality constraints are known in the inclusive matrix, SoMod computes the perturbation matrix E:

$$E_{i,i} = diag_pert; \quad n \leq i \leq n + m \tag{12}$$

where n and m are the number of variables and constraints respectively. Matrix E will be added to the inclusive matrix as shown in Equation 10.

Load-Balanced Level Coarsened scheduling. Before performing the factorization, we use Load-Balanced Level Coarsening to compute the order of factorization using the pruned inclusive assembly tree. This scheduling algorithm provides a partitioning of the tree that groups supernodes into partitions that can execute efficiently on a parallel processor while preserving ordering dependencies. For example, in the pruned tree of Figure 1d, nodes 1, 3, and 6 can run in parallel, since none of them depend on lower non-colored nodes in the tree.

Algorithm 2: Blocked-diagonal LDL factorization. Matrices K, L, D, P_S correspond to Equation 10. *super* is a vector that shows the boundary of supernodes in L. $\mathcal{M}$ is a set that shows the order of computation. Matrices L and D only have the sparsity pattern for LBL and include the previous factor for row modification.

```
   Data: K, L, D, super, M
   Result: L, D, P_S
1  for j ∈ M do
2      b = super_j
3      u = super_{j+1}
4      L_{:,b:u} = 0
5      T(:,:) = 0
       /* Applying contributions from factorized supernodes in r */
6      for r ∈ L_{0:b,:} do
7          T = T + L_{b:n,r} × D_{r,r} × L^T_{b:u,r}
8      end
9      [L_{b:u,b:u}, D_{b:u,b:u}, P_{S b:u,b:u}] = LDL(K_{b:u,b:u} - T_{b:u,:})
       /* Applying column permutation */
10     L_{:,b:u} = L_{:,b:u} × P^T_{S b:u,b:u}
11     L_{u:n,b:u} = (L_{b:u,b:u} × D_{b:u,b:u})^{-1} × (K_{u:n,b:u} − T_{u:n,:})
12 end
   /* Applying row permutation */
13 L = P_S × L
```

LBL: parallel blocked-diagonal LDL factorization. LBL is a parallel LDL factorization method that takes the computed schedule from the Load-Balanced Level Coarsening algorithm, the visibility vector, and the sparsity pattern of the L-factor and computes the nonzero values of the L and D factors of the perturbed KKT matrix of the system in Equation 7 (line 2 in Algorithm 1). Pseudocode for LBL is shown in Algorithm 2.

LBL uses a supernodal left-looking approach [Davis 2006] (one of several ways to compute LDL factorization), computing the supernodes of the L-factor using already factorized supernodes to the left of the current supernode. The list of supernodes and the order of computation are specified in *super* and $\mathcal{M}$ respectively. Each iteration of LBL first accumulates contributions of supernodes to the left and stores them in temporary matrix T. After deducting T from the current column (line 9), the algorithm first factorizes the diagonal part of the current supernode using a dense LDL factorization and then uses the computed factors to factorize the off-diagonal part of the current column (line 11). The dense LDL factorization uses the Bunch-Kaufman algorithm, which only reorders rows within a supernode of the L-factor. Thus, LBL pivoting is restricted to rows

Algorithm 3: Symbolic row removal algorithm. k is the node to remove. π is the pruned inclusive assembly tree. v is the visibility vector. r is the list of root nodes. $\pi^{-1}(k)$ returns the children list of node k.

```
Data: π, v, k, r
Result: π, v, r
/* Find the parent of deleting node k */
1  f = π(k)
/* Update all children of node k with its parent */
2  for j ∈ π^{-1}(k) do
3      if v(j) then
4          π(j) = f
5          if k is a root node then
6              r = r ⋃ {j}
7          end
8      end
9  end
10 r = r − {k}
/* Update the visibility vector */
11 v(k)=false
```

Algorithm 4: Symbolic row addition algorithm. k is the node added. π is the pruned inclusive assembly tree. π_{inc} is the inclusive assembly tree. v is the visibility vector. r is the list of root nodes. $\rho_{inc}(j)$ returns the list of ancestors of node j. $\pi^{-1}(f)$ returns the children list of node f.

```
Data: π, π_inc, v, r, k
Result: π, v, r
/* Find the first visible ancestor of k */
1  f = min {j | j ∈ ρ_inc(k) ∧ v(j)}
2  if f is a node then
       /* Find all nodes that k is their least ancestor */
3      for j ∈ π^{-1}(f) do
4          if k ∈ ρ_inc(j) then
5              π(j) = k
6          end
7      end
8  else
       /* Look for any missing child in root nodes */
9      for j ∈ r do
10         if k ∈ ρ_inc(j) then
11             π(j) = k
12             r = r - {j}
13         end
14     end
15     r = r ⋃ k
16 end
/* Update the pruned inclusive assembly tree */
17 π(k) = f
18 v(k)=true
```

within a supernode [Schenk and Gärtner 2006], which preserves the sparsity pattern of the L-factor during factorization.

After pivoting, rows of the L-factor in supernodes to the left of the current supernode must be permuted as well; were this done within the parallel region (lines 2–11), it would introduce dependencies that would prevent efficient computation, rendering the Load-Balanced Level Coarsening schedule useless. Thus, unlike typical LDL factorization methods, LBL separates row and column permutations, applying row permutations after the factorization (line 13). After obtaining the factorization, SoMod then uses triangular solve to efficiently obtain a solution to the initial KKT problem, as described in Section 4.3.

LBL thus works with the same base SBK algorithm as in MKL Pardiso [Schenk and Gärtner 2006]. However, LBL enables additional important features necessary for updates. The first and most key feature is that LBL enables factor modification: when adding or removing a constraint from K, LBL modifies the factor as opposed to MKL Pardiso which requires computing the factor from scratch. The second feature is LBL's application of a static scheduler [Cheshmi et al. 2018] to schedule the computation to ensure load-balanced parallelism while preserving locality. In contrast, MKL Pardiso utilizes dynamic scheduling which optimizes solely for load-balanced execution, which results in suboptimal locality. To prevent dependencies due to pivoting in SBK that would limit parallelism, LBL postpones row permutation to after numerical factorization.

4.2 Factor Modification

After the initialization phase, finding a solution to the QP problem requires solving a large number of successive symmetric indefinite KKT systems. The factor modification phase in SoMod efficiently solves these successive systems by reusing the computed factors from the initialization phase and modifying them based on whether a new inequality constraint is added or removed. In contrast, the usual approach would solve these systems from scratch, performing symbolic analysis and factorization without reusing any previously-computed information.

Successive KKT matrices are created by adding or removing rows of matrix C to/from the existing KKT system. To obtain the solution to the linear system in Equation 8 (line 4 of Algorithm 1), SoMod first updates the symbolic information and then the numeric factorization previously obtained from the initialization phase or obtained from the previous iteration of the QP algorithm. It then uses the updated L-factor and D to obtain a solution (Section 4.3). In this subsection we explain how factor modification efficiently modifies the previously obtained symbolic information and then uses the new symbolic information to update the existing numeric factors.

4.2.1 Symbolic Modification. The symbolic modification phase modifies the pruned inclusive assembly tree using the full inclusive tree when row k of the inclusive matrix is modified. Depending on whether the modification adds or removes a row, SoMod uses symbolic row removal or row addition algorithms to update the tree.

Row removal. When a constraint is removed from the KKT matrix, SoMod updates the pruned inclusive tree using the symbolic row removal algorithm shown in Algorithm 3. To remove node k, the removal algorithm first finds its parent and then assigns all children

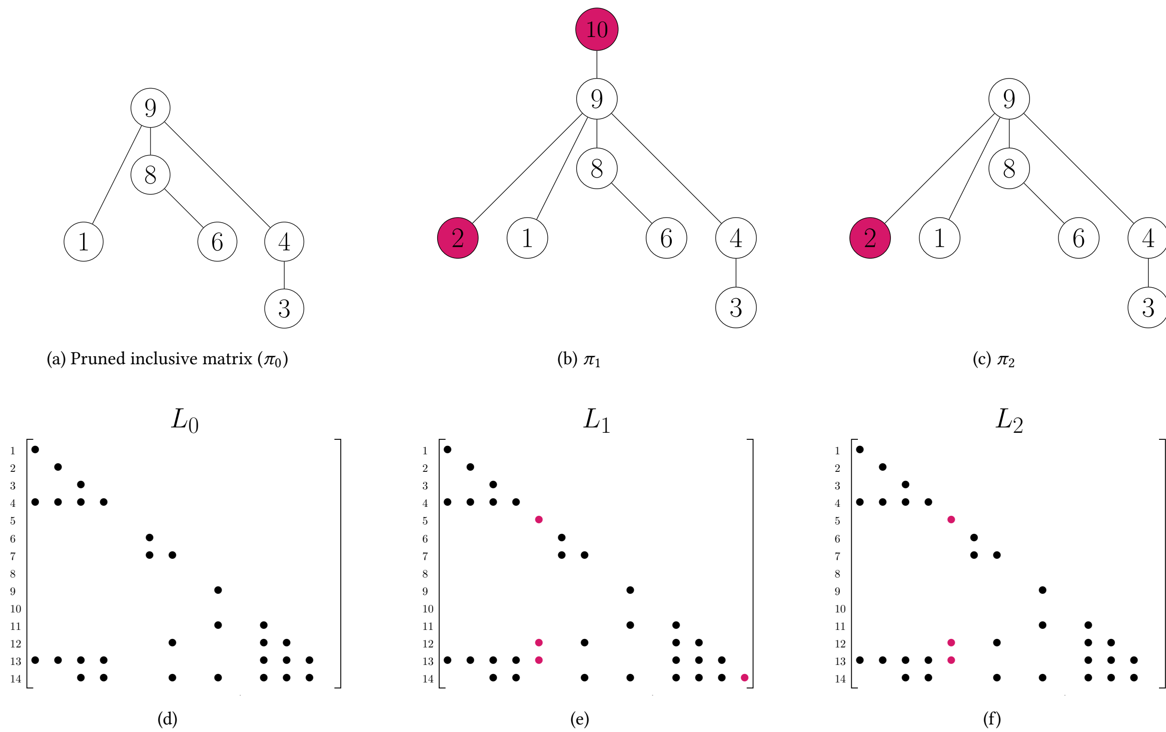


Fig. 2. Factor modification example starting with the pruned inclusive tree (Figure 2a) and the L-factor (Figure 2d) that are computed in the initialization phase, in order, by removing all nodes corresponding to the inequality matrix from the inclusive tree in Figure 1d and by hiding all rows of the inequality matrix in the L-factor in Figure 1c. SoMod symbolically adds rows that correspond to nodes 2 and 10 (rows 5 and 14, respectively) to the inclusive matrix using the row addition algorithm, resulting in a new pruned inclusive tree shown in Figure 2b. The corresponding supernodes in the L-factor in Figure 2e, shown in red, are also visible and will be updated using the numerical modification algorithm. Figure 2c is the result of removing node 10 from Figure 2b by using the symbolic row removal algorithm. Column 14 of the L-factor (which corresponds to node 10 in the tree) in Figure 2f becomes invisible after row removal.

of node k to its parent. If node k is a root node and therefore has no parent, we add its children to a list r which contains all root nodes. This list is used in the row addition algorithm to facilitate the process of adding constraints corresponding to such nodes. For example, Figure 2b shows a pruned inclusive assembly tree with two already-added constraint rows 2 and 10 and Figure 2c shows the pruned inclusive assembly tree after node 10 is removed. Node 9 then becomes a root node, so Algorithm 3 adds it to r and removes 10 from r.

Row addition. When row k is added to the KKT matrix, SoMod updates its symbolic information, finding where to insert node k into the pruned inclusive assembly tree. The algorithm first visits the tree to find the first visible ancestor of k. If there is a first visible ancestor, the algorithm then finds the children of this closest ancestor f in the pruned inclusive tree; the algorithm looks through these children and updates any for which k is the parent. If k does not have a first visible ancestor, the algorithm cannot use its ancestor's information to update the pruned inclusive tree and thus would need to search for its children by considering all nodes of the inclusive tree. Instead, the algorithm uses the list of root nodes from the node removal algorithm and only searches in r. Algorithm 4 shows the process of row addition. Figure 2a shows the pruned inclusive tree with no constraints and Figure 2b shows the pruned inclusive assembly tree after adding constraint rows 2 and 10. When adding node 2, Algorithm 4 first finds its visible ancestor, node 9, and then updates the parent node with 9 since none of children of node 9 belong to node 2. Node 10 is a root node in the inclusive tree of Figure 1d, thus the algorithm goes over the list of root nodes, which has 9 in it as explained in the example of the row removal section. Since 9 is a child of 10 in the inclusive tree, its parent will be updated with 10.

4.2.2 *Numeric Modification.*

SoMod uses the updated pruned inclusive tree to update the numeric factorization of the newly-modified KKT system by visiting only the columns that are dependent on the modified row in the pruned inclusive assembly tree. Given the pruned inclusive assembly tree of a linear system, solving the factorization after adding or removing a row is similar to Algorithm 2. The only difference is how the input $\mathcal{M}$ is computed. Numeric modification only updates supernodes that are in $\rho(k)$, which are the nodes in an *up-traversal* starting from k. An up-traversal from a node visits all ancestors of that node. For example, the up-traversal of node 2 in the tree of Figure 1d is {2, 9, 10}. For the node removal case, the

symbolic row removal algorithm is called after the numeric modification. This allows the modification algorithm to apply the effect of removing the node before removing the necessary information from the symbolic information; instead, the row is replaced with all zeros in order to update the numeric factorization. For the node addition case, the symbolic row addition algorithm is called before numeric modification, and the appropriate nonzeros are added to the row. Calling Algorithm 4 before numeric modification allows the numeric modification algorithm to utilize values of the added row during the L-factor update.

4.3 Triangular Solve and Accuracy Refinement

In both the initialization phase as well as the modification phase, once SoMod obtains the newly-computed or updated numeric factors L and D, it uses them to solve the linear system $Kx = s$. SoMod finds the solution vector x by doing a forward triangular solve, a block-diagonal system solve, and a backward triangular solve as shown in Equation 13. SoMod uses an efficient parallel triangular solve from [Cheshmi et al. 2018] and a simple single-threaded hand-written block-diagonal system solver for these steps.

$$\begin{aligned} b &= (P_{fill}P_S)s \\ Lx_1 &= b \\ Dx_2 &= x_1 \\ L^T x_3 &= x_2 \\ x &= (P_{fill}P_S)^{-1}x_3 \end{aligned} \tag{13}$$

As shown in Equation 10 and discussed in Section 4.1.2, we add perturbation matrix E to the KKT matrix prior to solving. As a result, the solution x contains inaccuracies, necessitating an accuracy refinement strategy to obtain an acceptable solution.

Accuracy refinement. It is standard practice to use an iterative method after a direct solve to improve the accuracy of the solution [Arioli et al. 2007; Hénon et al. 2002; Saad 2003; Schenk and Gärtner 2002]. SoMod uses an iterative method, right-preconditioned GMRES [Saad 2003], to refine the obtained solution from the solve phase. The GMRES algorithm is preconditioned with the output of LDL factorization and performs up to *max_iter* iterations to achieve the requested residual norm *res_tol*, but will terminate early if the required tolerance is achieved.

5 NASOQ: NUMERICALLY ACCURATE SPARSITY-ORIENTED QP SOLVER

With our key innovation SoMod in place, we now can define our two closely-related QP solution algorithms, NASOQ-Fixed and NASOQ-Tuned. Both methods integrate SoMod row modification and LBL within the GI dual-feasible active-set framework and so provide efficient, accurate, and sparsity-preserving full-space QP algorithms. NASOQ-Fixed and NASOQ-Tuned both, as we show in Section 6, consistently improve over state-of-the-art QP methods across our benchmark, while the two methods each individually offer a different balance in the trade-off between efficiency and accuracy for larger scale problems.

In this section we first highlight the changes applied by SoMod row modification and LBL to the GI framework with NASOQ-Fixed and then, building off of this baseline, discuss the NASOQ-Tuned method as a direct and natural extension of NASOQ-Fixed.

Algorithm 5: NASOQ: A dual-feasible full-space QP solver.

```
   Data: H, q, A, b, C, d
   Result: x, y, z
   /* Feasibility phase                                    */
 1 LinearSolve SoMod(H, A, C);
 2 z_0 = 0; k = 0; active-set=∅; modify=ADD;
 3 SoMod.symbolic_initialization();
 4 SoMod.LBL(q, b);
 5 [x_0, y_0] = SoMod.solve();
   /* Optimality phase                                     */
 6 while x_k is not primal-feasible do
 7     w = most_violated(C, d, x_k);
 8     if modify == ADD then
 9         SoMod.symbolic_row_addition(w);
10         SoMod.numerical_modification(w);
11     else
12         SoMod.numerical_modification(g);
13         SoMod.symbolic_row_removal(g);
14     end
15     [Δx, Δy, Δz]=SoMod.solve();
16     Compute the step length t;
17     if t = ∞ then
18         Problem is unbounded.;
19     else
20         Update x_{k+1}, y_{k+1}, z_{k+1} with Δx, Δy, Δz;
21         Update the active set;
22         Add or remove a row to/from KKT;
23         Set g to removed constraint;
24         Set modify to either ADD or REMOVE;
25     end
26     k = k + 1 ;
27 end
28 x_k, y_k, z_k are optimal;
```

Algorithm 5 summarizes our full NASOQ-Fixed algorithm in pseudocode. At each update and solution of the new active-set KKT (previously lines 2 and 4 in Algorithm 1) NASOQ-Fixed now applies the SoMod solve phase via LBL and row modification. This allows NASOQ-Fixed to replace the Cholesky and QR solves in the standard GI method; this is the key difference between NASOQ and standard GI methods. In addition, standard GI implementations require one additional iteration for each equality constraint while NASOQ, due to its full-space approach, applies equality constraints by solving the initial KKT system, see Equation (7).

Numerical optimization methods generally apply a wide diversity of empirically tuned parameters [Nocedal and Wright 2006; Stellato et al. 2020]. A key feature of NASOQ is that in our construction of SoMod's LBL and row modification we expose three parameters with direct and intuitive interpretations that enable us to balance

efficiency against accuracy for different applications and problem scales. With NASOQ-Fixed we demonstrate that without tuning a default setting works well across the board. With NASOQ-Tuned we similarly demonstrate that if a range of reasonable settings for these parameters are a priori known, NASOQ's active set approach enables a rapid sweep for improved accuracy. These parameters are

- *max_iter*: the maximum number of refinement iterations for incrementally improving the solution of a KKT system after the solve phase (see Section 4.3);
- *stop_tol*: the threshold defining the upper bound for the residual accuracy of the KKT system during the refinement phase (see Section 4.3);
- *diag_perturb*: value added to zero-entry diagonals of the KKT matrix (see Section 4.1.2) to stabilize LBL and row modification in SoMod.

Increasing *max_iter* and decreasing *stop_tol* generally improve accuracy at the cost of more computation. Setting of *diag_perturb* then relates to problem conditioning and machine accuracy; a typical value is near square root of machine precision [Arioli et al. 2007].

5.1 NASOQ-Fixed

NASOQ-Fixed is a direct, one-shot application of our SoMod-enhanced GI approach. Here we presume that, per-application, time is sufficient for a single QP solve, but no further, and so seek the most effective values for given input QP characteristics.

NASOQ-Fixed sets *diag_perturb* and *stop_tol* to fixed values *for all* input QP problems to 10^{-9} and 10^{-15} respectively. Here we find that adapting *max_iter* per problem based on the requested accuracy ϵ is most effective (with the assumption that we will only have a single attempt at the solve). When lower accuracy ϵ's are requested, NASOQ-Fixed applies fewer refinement iterations (lowering *max_iter*) and then correspondingly increases *max_iter* to obtain more accurate linear solves when higher accuracies are specified. Concretely, NASOQ-Fixed sets

$$max_iter = \begin{cases} 1, & \epsilon > 10^{-4} \\ 2, & 10^{-8} \leq \epsilon \leq 10^{-4} \\ 3, & \epsilon < 10^{-8}. \end{cases} \tag{14}$$

Finally, for very small QP problems, i.e., fewer than 100 nonzeros, NASOQ-Fixed keeps *max_iter* = 3 as this does not impose an appreciable cost.

5.2 NASOQ-Tuned

NASOQ-Tuned leverages the underlying active-set framework. Active-set methods terminate in bounded time with respect to number of constraints, and, in practice generally much more rapidly than barrier and first-order approaches [Maes 2011; Wright 1997]. Thus the cost of running multiple passes of NASOQ to determine whether a chosen setting for our three parameters successfully matches a requested accuracy is generally acceptable when accuracy is critical and some efficiency can be sacrificed.

NASOQ-Tuned therefore sweeps through a set of empirically-determined parameter combinations, as found from testing solely against the subset (0.5%) of the QP problem instances in the testing set (see Section 6.3.1) for which NASOQ-Fixed does not converge. NASOQ-Tuned begins with an initial pass of NASOQ-Fixed. Then, if the requested accuracy ϵ is not met, it successively tries new NASOQ passes with the sequence of configurations in Table 1.

Config	max_iter	diag_perturb	stop_tol
1	2	10^{-9}	10^{-15}
2	2	10^{-13}	10^{-15}
3	2	10^{-7}	10^{-15}
4	2	10^{-11}	10^{-17}
5	3	10^{-10}	10^{-15}
6	3	10^{-9}	10^{-17}
7	3	10^{-11}	10^{-17}

Table 1. List of NASOQ-Tuned parameters. Each row contains parameters used in one pass of NASOQ-Tuned.

In practice, for all but 21 examples in our benchmark, we find that NASOQ-Tuned successfully converges at all requested accuracies. See Section 6.3 for details.

6 EVALUATION

In this section we evaluate NASOQ against other solvers on a large set of QP problems from diverse applications. First, we describe our experimental setup (Section 6.1) and our new collection of 1513 sparse QP problems collected from a wide variety of applications (Section 6.2). We then evaluate NASOQ for the full benchmark set, comparing accuracy, efficiency, and scalability against existing tools (Section 6.3). We then describe the effect of numerical range on NASOQ's ability to attain convergence (Section 6.4). Finally, we explore the impact of SoMod on overall performance (Section 6.5).

Across all QP problems in our repository, NASOQ converges for over 99.5% of the problems for accuracies ranging from 10^{-3} to 10^{-9} with average speedups ranging from 1.7× to 24.8× over the best competing method. For requested accuracies of 10^{-3} and 10^{-6}, NASOQ-Tuned has no failures, while only 21 problem instances (out of 1513, or 1.4%) fail to reach the 10^{-9} accuracy threshold. Our analysis shows that these few failures occur due to the numerical range of the problems themselves and that other solvers likewise fail to solve these problems. NASOQ demonstrates consistent efficiency and speedups across all application types, and we see that the SoMod algorithm plays a critical role in the performance of NASOQ.

6.1 Experimental Setup

Testbed architecture. All experiments are performed on a 6-core 3.30GHz Intel Core i7-5820K processor with 32GB of main memory and turbo-boost disabled, running Ubuntu 16.04 with Linux kernel 4.4.0. NASOQ and all open-source solvers are compiled with GCC v5.4.0 using the `-O3` option. MKL 2019.1.144 is used wherever dense BLAS routines are required. Throughout this section, *convergence time* refers to the wall clock time to reach convergence, measured using the standard C++ `chrono` library. We use a time limit of 30 minutes for all solvers; if a solver does not converge in 30 minutes for a problem instance, we consider it a failure for that instance.

Termination criteria. We set all solvers, where possible, to use common, absolute (rather than relative) termination criteria, i.e., a common accuracy threshold ϵ for all four measures in Equations 3-6. Relative tolerances are often specific to the algorithm and/or particular domain, and are often highly susceptible to falsely reporting convergence when an algorithm stagnates (e.g. small relative errors only tell us iterates have stopped progressing) rather than reaching a low error solution. Although accuracies for each of the four optimality measures in Equations 3-6 change depending on application, we believe a desirable goal for a general-purpose QP solver is to solve every reasonable problem to any requested accuracy given commensurate time, and to only report success when accuracy is achieved. We don't presume to know a priori per problem type what measures are most important; instead, we evaluate fitness by asking each method to drive all measures down below each specified error tolerance according to the infinity norms of Equations 3-6. Details for each solver are explained separately below.

Solver settings. We compare NASOQ with four widely-used state-of-the-art QP solvers: OSQP [Stellato et al. 2020], Gurobi [Optimization 2014], MOSEK [Mosek 2015], and QL [Schittkowski 2003]. These tools are selected to represent different QP solver methods. OSQP applies a first-order method, supports sparse problems, and parallelism. Gurobi and MOSEK are both commercial tools based on barrier methods; both support parallel execution and sparse QP problems. To compare NASOQ to an alternative Goldfarb-Idnani algorithm implementation, we include QL, a robust GI implementation; however, QL does not support sparsity nor parallelism.

NASOQ is implemented in C++ with double precision, with METIS 5.1.0 for reordering the inclusive matrix, and MKL BLAS [Wang et al. 2014] for dense operations within LBL. All other QP solvers are set to their default modes and only settings related to the requested accuracy or ϵ in Equations 3-6 are changed when exposed and necessary, see below[4].

OSQP is an open-source first-order solver designed for sparse QP problems. We use OSQP 0.6.0 and build in double precision using the MKL Pardiso solver. In OSQP the user-requested accuracy is scaled by the norm of matrix H (see [Stellato et al. 2020] Section 3.4). This scaling leads to early termination and thus inaccurate/non-optimal solutions. For fair comparison, we change OSQP's termination criteria to use the absolute requested accuracy – leaving the algorithm otherwise unchanged[5].

Gurobi and MOSEK are two commercial solvers that apply barrier methods to solve QP problems. For both packages, we utilize default settings for the solvers. We use currently latest releases of MOSEK (v8) and Gurobi (v8). Gurobi uses an absolute termination criteria and so can be applied in our comparison directly. MOSEK, on the other hand, does not allow absolute error tolerances for convergence and instead applies algorithm-specific, relative measures. As MOSEK is closed source (and so its termination criteria can not be modified) we experimented with a range of its different exposed parameter settings, seeking to maximize MOSEK's success. Discussion of our experiments with MOSEK and details of MOSEK's behavior applied with its most successful settings for comparison are covered in Section 6.3.1 below.

QL is a dense active-set solver based on the GI algorithm, implemented in Fortran. We convert all sparse matrices to dense prior to using QL, as it only supports dense matrices. Conversion time is not included in reported solve times. Large-scale QP problems cannot be converted due to memory limitations of the testbed architecture.

Performance profile. Aggregating combined performance and failure data in plots across methods on a significantly-sized benchmark is always challenging. Thus, to compare the convergence speed of different solvers, following existing work [Dolan and Moré 2002; Pandala et al. 2019; Stellato et al. 2020; Wong 2011] we utilize a performance profile plot. To define performance profiles, we use the performance ratio $r_{p,s} = \frac{t_{p,s}}{min_s t_{p,s}}$ where $t_{p,s}$ is the time for QP solver s to solve problem instance p. When solver s fails for problem p, its performance ratio is set to infinity, i.e., $r_{p,s} = \infty$. After the performance ratio for all pairs of solvers and problem instances is obtained, we compute the performance profile, function f_s, that maps any $r_{p,s}$ to $[0, 1]$ and is computed as: $f_s(\tau) = \frac{1}{n_p} \sum_p \alpha_{\leq\tau}(r_{p,s})$ where $\alpha_{\leq\tau} = 1$ if $r_{p,s} \leq \tau$ and n_p is the number of problems in our repository. $f_s(\tau)$ denotes the fraction of solved problems within $\tau\times$ the time of the best solver. Thus, in Figure 4 for example, faster performance for a given fraction of problems means the line is to the left, while more problems with successful convergence lead to lines that are higher on the y-axis.

Speedup. In addition to performance profiles, we also provide detailed, per-category analyses and breakdowns using speedup and failure rate (Section 6.3). The reported average speedup throughout the paper is computed using normalized shifted geometric mean [Mittelmann 2020; Stellato et al. 2020]. Given $t_{p,s}$ is the time for QP solver s to solve problem instance p, shifted geometric mean of solver s across n problems is computed as: $g_s = \sqrt[n]{\prod_p (t_{p,s} + k)} - k$ where k is the shift and selected to be one [Stellato et al. 2020]. When solver s fails in the problem p, $t_{p,s} = 1800$ which is the 30 minute time limit in seconds. To avoid overflow we use the logarithmic form of the geometric mean. Given g_s for each solver, the speedup of solver $s1$ over $s2$ is computed by $\frac{g_{s2}}{g_{s1}}$.

6.2 Benchmark Repository for Sparse Quadratic Programs

We assemble a repository for sparse QP problems of different scales, most of which come from applications in animation, geometry processing, and simulation. Existing QP problem benchmarks are not large enough to stress-test large-scale QP solvers. For example, the largest QP problem instance in terms of the number of variables in the Maros-Mészáros repository [Maros and Mészáros 1999] only has 10k variables, which is far smaller than real-world large-scale QP problems. Existing QP solvers are either tested for a limited number of problems or are tested for randomly generated problems [Pandala et al. 2019; Stellato et al. 2020]. To address this shortcoming, we gathered existing strictly-convex QP benchmark problems and also added a set of new QP problem instances mostly arising from computer graphics applications.

Our repository includes QP instances from shape deformation, contact simulation, model reconstruction, and cloth simulation from

[4]The list of parameters related to user-requested accuracy for each solver is provided in supplemental material.
[5]In the supplemental material, we provide detailed information on the sole parts of the OSQP code we modify.

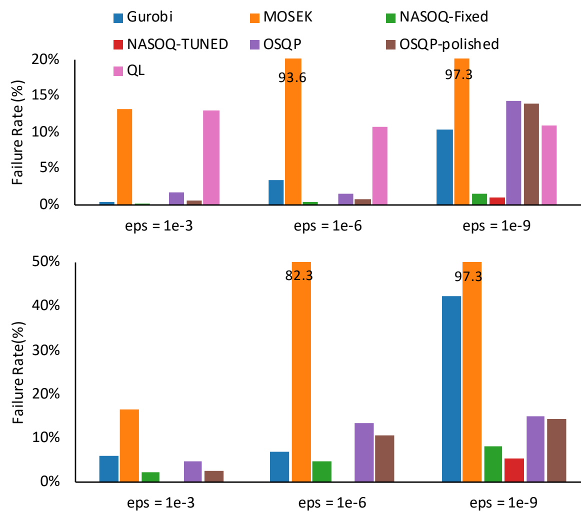


Fig. 3. Failure rate of NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (10^{-3}, 10^{-6}, and 10^{-9}) and for both small-scale (top) and large-scale (bottom) QP problems. NASOQ-tuned has the lowest failure rate compared to all other QP solvers for problems of different scales and for different requested accuracies.

computer graphics; model predictive control (MPC) [Segata 2019] from robotics; and strictly-convex QP problems from the Maros-Mészáros repository [Maros and Mészáros 1999]. The number of variables ranges from 50–114309 and the number of constraints ranges from 20–10k. Each QP for image deformation comes from Jacobsen *et. al.* [Jacobson et al. 2011] and is created using libigl [Jacobson et al. 2018]. Contact simulation QPs correspond to QP problems that must be solved in each timestep of the simulation and are created using the GAUSS library [Levin 2019]. Model reconstruction instances are QP problems that compute the third dimension of a 2D mesh, explained in [Dvorožňák et al. 2018; Sýkora et al. 2014]. Cloth simulation QPs arise from each timestep of the cloth simulation in [Weidner et al. 2018].

6.3 Accuracy, Efficiency, and Scalability of NASOQ

NASOQ can solve a large range of QP problems from different application types and across a range of problem scales. In this section, we first compare the efficiency and scalability of NASOQ to other QP solvers and demonstrate NASOQ's superior performance. We also explore the performance of NASOQ versus other tools for different types of applications. Finally, we discuss the effect of using the full-space method in NASOQ.

6.3.1 Overall performance. As discussed in Section 5, NASOQ-Fixed and NASOQ-Tuned target different points in the trade-off between efficiency and accuracy. NASOQ-Tuned sweeps through a set of parameters to deliver improved accuracy for problems where accuracy is critical. Thus, as shown in Figure 3, NASOQ-Tuned always converges for requested accuracy thresholds of 10^{-3} and 10^{-6}, while NASOQ-Fixed fails for 1.2% of problems (there are 21 problem instances that NASOQ-Tuned fails for 10^{-9}; this is explained in Section 6.4). Since NASOQ-Tuned starts from the NASOQ-Fixed configuration, the performance profiles of both variants are similar, as shown in Figure 4 and the small difference is due to problems that NASOQ-Tuned converges and NASOQ-Fixed fails. The convergence behaviour of both variants of NASOQ is consistently better than other solvers for both large- and small-scale problems (Figure 3).

OSQP uses several lightweight iterations to incrementally improve the accuracy of the solution to the QP problem. However, when an accurate solution is needed, the number of iterations significantly increases in OSQP, leading to reduced efficiency. Like NASOQ-Tuned, OSQP also has a variant, called OSQP-polished, that trades off efficiency for accuracy in problems where accuracy is critical. OSQP-polished uses an additional step after OSQP to refine accuracy and obtain solutions for some problems when the accuracy range is 10^{-9}. OSQP and OSQP-polished collectively solve 94% percent of all problems in our repository for accuracy ranges of 10^{-3}, 10^{-6}, 10^{-9} which is quite good, but still considerably less than the 99% obtained from NASOQ (see Figure 3). NASOQ is more efficient than OSQP across all problem scales and for different accuracy thresholds. For example, the average speedup of NASOQ-Fixed over OSQP for thresholds of 10^{-3} and 10^{-6} is 2.7× and 2.3× respectively.

Gurobi, in contrast to NASOQ, has a high failure rate and does not scale to larger problems. Unlike NASOQ and OSQP, Gurobi does not provide different variants to balance accuracy and efficiency. In Gurobi, the number of iterations typically remains unchanged for different requested accuracies. Thus, in Figure 4, all performance profiles for Gurobi follow similar trends across different requested accuracies and different problem scales. For accuracies of 10^{-3} and 10^{-6}, Gurobi's failure rate is similar to that of OSQP; however, compared to NASOQ, Gurobi fails in more problems. Furthermore, Gurobi exhibits a high failure rate for large-scale problems with lower requested error. For example, for the threshold of 10^{-9}, Gurobi fails for 42.25% of large-scale problems as shown in Figure 3.

MOSEK is another barrier method that converges in a bounded number of computationally-heavy iterations. MOSEK does not converge for most large-scale problems with accuracy thresholds lower than 10^{-3}. As shown in Figure 3, the failure rate of MOSEK for smaller requested accuracy thresholds is more than 82%, which is significantly higher than the failure rate of all other solvers. As discussed in Section 6.1, MOSEK doesn't allow absolute error tolerances and instead applies algorithm-specific, relative measures. We experimented with a number of different parameter settings, attempting to improve MOSEK's success. During this process we observed that decreasing requested accuracies further below 10^{-10} produces slower performance and increased failures. For example, requesting 10^{-16} accuracy leads failure rates to increase to 86%. We find setting to the requested accuracy works best for MOSEK in terms of combined performance and failure rate reduction. We also set MOSEK's infeasibility tolerance parameter to the default: 10^{-12}. We find no change for high accuracy benchmarks (i.e. 10^{-6} and 10^{-9}) and a 0.2% reduction in failure rate for 10^{-3}. We observe, however, consistent with [Stellato et al. 2020] the speed and failure rate of MOSEK generally lags behind OSQP at low accuracy and Gurobi at higher accuracies.

QL is a dense active-set solver and thus can only solve small-scale problems. For small QP problems, QL's failure rate is 11% for each

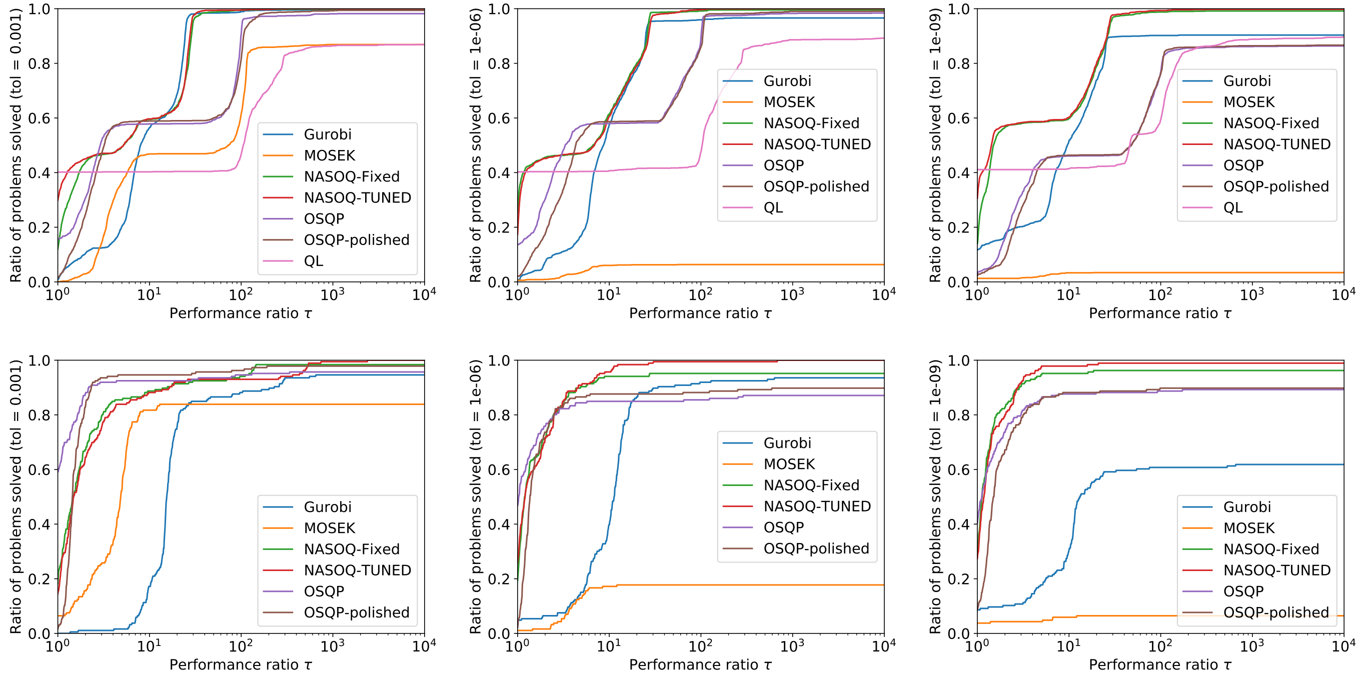


Fig. 4. Performance profiles for NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (from left to right: 10^{-3}, 10^{-6}, and 10^{-9}) and for small-scale (top) and large-scale (bottom) QP problems from our repository. Lines to the left are more efficient, and lines higher on the y-axis solve a greater percentage of problems within a given performance threshold. The figures show that NASOQ-Fixed and NASOQ-Tuned are, for almost all accuracies and all problem scales, more efficient than available QP solvers and are able to solve more of the QP problems in our repository.

of the accuracy thresholds as shown in Figure 3. The figure also shows that the performance profile and efficiency of QL in Figure 4 is inferior compared to other QP solvers including NASOQ, due to its lack of support for sparsity and parallelism.

6.3.2 Effect of Different Applications. Different applications create varying types of QP problems that pose different challenges to solvers. We examine the obtained accuracy and efficiency with different QP solvers as we vary QP problem types. Our analysis shows that unlike other QP solvers, NASOQ performs well across different application domains.

To show this variation, we compare NASOQ-Tuned, NASOQ-Fixed, OSQP, and Gurobi across different application types for the accuracy threshold of 10^{-6}; the trend holds for other accuracies. QL and MOSEK do not successfully converge for larger problem sizes, so we exclude them from our comparison.

For contact simulation problems, NASOQ-Tuned and NASOQ-Fixed provide the lowest failure rates (0% and 0.15%, respectively) compared to all other solvers. Although OSQP's failure rate (1.07%) is higher than NASOQ, it still performs better than Gurobi, which fails for 3.44% of these instances. The efficiency of these solvers also follows the same trend where both NASOQ solvers are faster than OSQP in 80% of contact simulation problems with an average of 2.1× speedup across all contact simulation instances. OSQP also exhibits better efficiency than Gurobi.

In shape deformation and model reconstruction, NASOQ-Fixed and NASOQ-Tuned do not fail for any problems while OSQP and Gurobi fail in 12.5% of instances. NASOQ is 22.8× and 24× faster than OSQP and Gurobi respectively for these problems.

For Maros-Mészáros problems, NASOQ-Tuned does not fail for any problem and the nearest competitors are Gurobi and NASOQ-Fixed with failure rates of 15% and 24.5%, respectively. NASOQ-Tuned is on average 8.9× faster than Gurobi and NASOQ-Fixed is slower than Gurobi by 3.2×. OSQP does not perform well for Maros-Mészáros problems (45% failure rate).

For model predictive control (MPC) problems, NASOQ-Tuned and NASOQ-Fixed show no failures while OSQP's failure rate is 2.5%, which is relatively high compared to Gurobi, which fails in only 0.83% of instances. Both NASOQ-Tuned and NASOQ-Fixed solvers are faster than OSQP and Gurobi. For example, NASOQ-Fixed obtains an average speedup of 3.4× over OSQP-polished.

Unlike other existing solvers, NASOQ provides consistent efficiency and good accuracy across all problem types. Both variants of NASOQ are more efficient and accurate compared to all solvers, with the exception of the failure rate of NASOQ-Fixed for Maros-Mészáros problems [6].

[6]The breakdown by application for each user-requested accuracy and for each solver is provided in supplemental material.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-9}$
NASOQ-Range-Space	0%	0.23%	2.42%
NASOQ-Fixed	0.1511%	0.45%	1.44%
NASOQ-Tuned	0%	0%	0.91%

Table 2. Failure rate of NASOQ for different ranges of accuracy using range-space (NASOQ-Range-Space) and full-space (NASOQ-Fixed and NASOQ-Tuned) methods for small-scale problems in our QP repository. NASOQ-Fixed has a failure rate comparable to that of NASOQ-Range-Space. NASOQ-Tuned outperforms NASOQ-Range-Space and has no failures for accuracies $\epsilon = 10^{-3}$ and $\epsilon = 10^{-6}$.

6.3.3 Effect of the Full-Space Approach. As discussed in Section 5, NASOQ replaces the range-space method in GI with a full-space approach. In this section we examine the effect of the full-space approach on the accuracy of NASOQ to demonstrate that the use of a full-space method does not negatively affect the accuracy of NASOQ compared to a range-space approach. To show the accuracy of NASOQ's full-space method, we integrate a range-space method inside NASOQ and use it to solve the KKT systems. We call this implementation NASOQ-Range-Space. Cholesky decomposition along with the QR decomposition are used instead of SoMod in NASOQ-Range-Space. However, due to the use of QR decomposition that has intensive memory usage, NASOQ-Range-Space is limited to solving small-scale problem instances. Table 2 shows the failure rates of NASOQ-fixed, NASOQ-Tuned, and NASOQ-Range-Space for small-scale problems in our QP dataset. NASOQ-Fixed has a failure-rate comparable to NASOQ-Range-Space and NASOQ-Tuned performs significantly better than NASOQ-Range-Space. Thus, using SoMod and the full-space method in NASOQ does not reduce the accuracy of the QP solver and can even improve accuracy with an appropriate choice of parameters for NASOQ-Tuned.

6.4 Effect of Numerical Range

NASOQ-Tuned and other QP solvers fail to solve 21 problem instances in our benchmark suite for the accuracy of 10^{-9}; Gurobi is an exception, but it can only solve 2 of these 21 problems. This section discusses properties of these 21 problems and explores why existing QP solvers and NASOQ-Tuned fail to solve them.

These problem instances have a large *numerical range* which can be classified into two categories: *(1)* problems that contain a large value (10^6 or larger) in either their input matrices or vectors (matrices H, A, and C and vectors q, b, and d in Equation 1); and *(2)* problems with large values in their primal or dual variables (vectors x, y, and z in Equations 1–2). This large numerical range limits the accuracy QP solvers can achieve in double precision.

This issue can be resolved if *(i)* for the first category, a scaling technique is used to normalize the range, and *(ii)* for the second category, an implementation with higher precision is used; for example, using floating-point types with 128 bits of precision.

Gurobi is the only QP solver that converges for two of these problem instances. While NASOQ-Tuned is able to get close to the accuracy threshold of 10^{-9} (because the stationarity norm for these two problems in NASOQ-Tuned is 4.7×10^{-9} and 4.4×10^{-9}), the maximum value of the Lagrange multipliers in these two problems is about 10^6, which leads to inaccurate solutions for some intermediate KKT systems and thus leads to failure in NASOQ-Tuned.

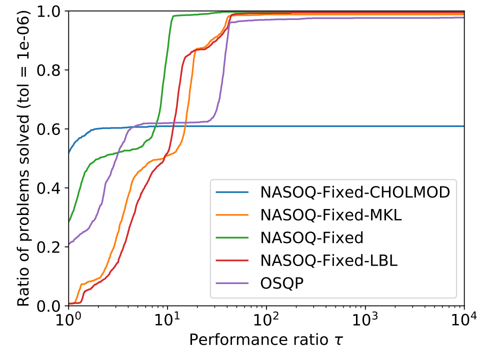


Fig. 5. Performance profile of NASOQ using SoMod (NASOQ-Fixed), using CHOLMOD row modification (NASOQ-Fixed-CHOLMOD), solving from scratch using LBL (NASOQ-Fixed-LBL), and solving from scratch using MKL (NASOQ-Fixed-MKL). OSQP is also shown as a reference solver. NASOQ-Fixed (green line) performs better than the modified versions of NASOQ. Note that this performance profile contains both small and large QP instances, unlike Figure 4.

6.5 Effect of SoMod

As discussed in Section 4, NASOQ uses SoMod to efficiently solve the successive KKT systems arising in active-set methods. In this section we analyze the effect of SoMod on NASOQ's performance and also separately demonstrate the efficiency of using LBL in NASOQ. We use the NASOQ-Fixed variant of NASOQ throughout this section because the effects of SoMod are the same in both variants. Figure 5 compares the performance profile of NASOQ-Fixed for $\epsilon = 10^{-9}$ with three different modifications of NASOQ: *(i)* NASOQ-Fixed-CHOLMOD, which uses CHOLMOD [Chen et al. 2008] instead of SoMod in NASOQ; *(ii)* NASOQ-Fixed-LBL, which instead of using row modification in NASOQ solves all KKT systems from scratch using LBL; and *(iii)* NASOQ-Fixed-MKL which solves all KKT systems in NASOQ using the MKL-Pardiso solver. In all modifications, the same number of accuracy refinement iterations is used. Overall, NASOQ-Fixed is faster than all other implementations while achieving the fewest failures.

NASOQ-Fixed-CHOLMOD replaces SoMod's row modification and LBL phases with row modification and the solver used in CHOLMOD [Chen et al. 2008]; the iterative refinement from SoMod is still used in NASOQ-Fixed-CHOLMOD because CHOLMOD does not come with refinement. CHOLMOD's row modification primarily supports symmetric positive definite (SPD) matrices. CHOLMOD will fail or provide inaccurate results for some indefinite systems: *(i)* unlike SPD systems, the diagonal value of the L-factor in an indefinite KKT system can sometimes be negative, so CHOLMOD may fail for these systems as it uses square root in computations that involve the diagonal value; *(ii)* to update the new L-factor, CHOLMOD uses the already computed L-factors, and thus numerical errors and inaccuracies may propagate to subsequent computations. NASOQ however re-computes the affected columns using the input matrix;

thus its L-factor is more accurate compared to that of CHOLMOD's as the number of iterations in the QP solver increase. Adding a perturbation value to the diagonal entries will remove some failures in CHOLMOD and so it can solve some (but not all) indefinite systems. With perturbation, the accuracy of the KKT solve using CHOLMOD is still low and leads to failure in 40% of QP problems. As shown in Figure 5, NASOQ-Fixed-CHOLMOD mostly converges for small-scale QP problems when the number of variables is fewer than 50, and needs few iterations to converge. Unlike NASOQ, NASOQ-Fixed-CHOLMOD does not need to create an initial inclusive matrix and thus its initial setup time is small; this leads to faster performance for very small QP problem instances. NASOQ-Fixed is overall faster than NASOQ-Fixed-CHOLMOD and results in the fewest failures.

NASOQ-Fixed is on average 3.1× faster than NASOQ-Fixed-LBL and NASOQ-Fixed-MKL. This demonstrates the importance of using the factor modification method of SoMod in NASOQ to avoid solving the KKT systems from scratch. In addition, NASOQ-Fixed-LBL and NASOQ-Fixed-MKL demonstrate a similar performance profile which warrants the use of LBL as a replacement solver for MKL in SoMod while benefiting from the unique features of LBL that facilitate the implementation of row modification in SoMod.

To separately measure the performance of LBL in NASOQ, we use the indefinite solver from MKL-Pardiso instead of LBL for solving the initial KKT system in NASOQ; we call this variant NASOQ-Fixed-Initial-MKL. All other components of SoMod that solve the successive KKT systems remain unchanged. NASOQ-Fixed obtains a similar performance to that of NASOQ-Fixed-Initial-MKL: it is roughly 1.01× faster. The reason for the small effect of LBL on overall performance of NASOQ is that only a small fraction of the overall time is spent on the initial factorization; on average initial factorization only accounts for 25% of NASOQ time.

7 CONCLUSION

NASOQ now enables simultaneously accurate and efficient solves for large and sparse QP problems across application domains. To better understand QP computational challenges and solver performance, we gathered a comprehensive benchmark set comprising a wide range of application-based QP problems. We hope that its application will lead to improved testing and further development of performant QP solvers. We are releasing both NASOQ and our new benchmark suite for QP problems as open-source projects to enable application of fast, numerically-accurate QP solutions.

To enable NASOQ we have constructed our new sparsity-oriented SoMod row modification method and LBL, our fast LDL factorization for indefinite systems. Together they enable the efficient updates and accurate solutions of the iteratively modified KKT systems critical to accurate QP solves.

Looking ahead there remain many interesting questions both in terms of both improving efficiency and per-problem automatic scaling for the most challenging QP problems we have identified. These should lead to even further improvements in robustness, efficiency and accuracy. At the same time there are many promising potential applications extending the building blocks we have developed here towards equality-constrained problems, non-linear optimization, the solution of more general saddle-point systems and beyond to other applications where factorization and update of indefinite systems remains critical.

ACKNOWLEDGMENTS

We thank Alec Jacobson, David I.W. Levin, Kevin Wampler, Seungbae Bang, Daniel Sỳkora, Marek Dvorožňák, Matthew Overby, and Oded Stein for their assistance. This work was supported in part by NSERC Discovery Grants (RGPIN-06516, DGECR-00303), the Canada Research Chairs program, and U.S. NSF awards NSF CCF-1814888, NSF CCF-1657175; used the Extreme Science and Engineering Discovery Environment (XSEDE) [Towns et al. 2014] which is supported by NSF grant number ACI-1548562; and was enabled in part by Compute Canada and Scinet (www.computecanada.ca).

REFERENCES

Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J. Zico Kolter. 2019. Differentiable Convex Optimization Layers. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc. http://papers.nips.cc/paper/9152-differentiable-convex-optimization-layers.pdf

Brandon Amos and J. Zico Kolter. 2017. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org.

M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. 2007. A Note on GMRES Preconditioned by a Perturbed LDL^T Decomposition with Static Pivoting. *SIAM J. Sci. Comput.* 29, 5 (Sept. 2007).

Jernej Barbic. 2007. *Real-Time Reduced Large-Deformation Models and Distributed Contact for Computer Graphics and Haptics*. Ph.D. Dissertation. USA.

Michele Benzi, Gene H Golub, and Jörg Liesen. 2005. Numerical solution of saddle point problems. *Acta numerica* 14 (2005).

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine Learning* 3, 1 (2011).

Stephen Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008).

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 13. https://doi.org/10.1145/3126908.3126936

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 62. http://dl.acm.org/citation.cfm?id=3291656.3291739

Timothy A Davis. 2006. *Direct methods for sparse linear systems*. Vol. 2. Siam.

Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019).

Timothy A Davis and William W Hager. 1999. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999).

Timothy A Davis and William W Hager. 2005. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 26, 3 (2005).

Timothy A Davis and William W Hager. 2009. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions on Mathematical Software (TOMS)* 35, 4 (2009).

Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91, 2 (2002).

A. Domahidi, E. Chu, and S. Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*.

Iain S. Duff. 2004. MA57—a Code for the Solution of Sparse Symmetric Definite and Indefinite Systems. *ACM Trans. Math. Softw.* 30, 2 (June 2004).

Marek Dvorožňák, Saman Sepehri Nejad, Ondřej Jamriška, Alec Jacobson, Ladislav Kavan, and Daniel Sýkora. 2018. Seamless Reconstruction of Part-Based High-Relief Models from Hand-Drawn Images. In *Proceedings of the Joint Symposium on Computational Aesthetics and Sketch-Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering (Expressive '18)*. Association for Computing Machinery, New York, NY, USA, Article 5.

AS El-Bakry, Richard A Tapia, T Tsuchiya, and Yin Zhang. 1996. On the formulation and theory of the Newton interior-point method for nonlinear programming. *Journal of Optimization Theory and Applications* 89, 3 (1996).

Kenny Erleben. 2013. Numerical methods for linear complementarity problems in physics-based animation. In *Acm Siggraph 2013 Courses*.

Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. 2014. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation* 6, 4 (2014).

M Fesanghary, Mehrdad Mahdavi, M Minary-Jolandan, and Y Alizadeh. 2008. Hybridizing harmony search algorithm with sequential quadratic programming for engineering optimization problems. *Computer methods in applied mechanics and engineering* 197, 33-40 (2008).

Roger Fletcher. 2013. *Practical methods of optimization*. John Wiley & Sons.

E. Michael Gertz and Stephen J. Wright. 2003. Object-Oriented Software for Quadratic Programming. *ACM Trans. Math. Softw.* 29, 1 (March 2003).

Philip E Gill, Walter Murray, Michael A Saunders, and Elizabeth Wong. 2005. User guide for SQOPT 7: Software for large-scale linear and quadratic programming.

Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. 1991. Inertia-controlling methods for general quadratic programming. *Siam Review* 33, 1 (1991).

Donald Goldfarb and Ashok Idnani. 1983. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical programming* 27, 1 (1983).

Gene H Golub and Charles F Van Loan. 2012. *Matrix computations*. Vol. 3. JHU press.

Jacek Gondzio and Andreas Grothey. 2006. Solving nonlinear financial planning problems with 109 decision variables on massively parallel architectures. *WIT Transactions on Modelling and Simulation* 43 (2006).

Nicholas Gould. 2006. An introduction to algorithms for continuous optimization.

Nicholas IM Gould, Mary E Hribar, and Jorge Nocedal. 2001. On the solution of equality constrained quadratic programming problems arising in optimization. *SIAM Journal on Scientific Computing* 23, 4 (2001).

Nicholas IM Gould, Dominique Orban, and Philippe L Toint. 2003. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Transactions on Mathematical Software (TOMS)* 29, 4 (2003).

William W Hager. 1989. Updating the inverse of a matrix. *SIAM review* 31, 2 (1989).

Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O'Brien. 2012. Updated Sparse Cholesky Factors for Corotational Elastodynamics. *ACM Trans. Graph.* 31, 5, Article 123 (Sept. 2012).

P. Hénon, P. Ramet, and J. Roman. 2002. PASTIX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. *Parallel Comput.* 28, 2 (Feb. 2002).

Philipp Herholz and Marc Alexa. 2018. Factor Once: Reusing Cholesky Factorizations on Sub-Meshes. *ACM Trans. Graph.* 37, 6, Article 230 (Dec. 2018).

Philipp Herholz, Timothy A. Davis, and Marc Alexa. 2017. Localized Solutions of Sparse Linear Systems for Geometry Processing. *ACM Trans. Graph.* 36, 6, Article 183 (Nov. 2017).

Toby Heyn, Mihai Anitescu, Alessandro Tasora, and Dan Negrut. 2013. Using Krylov subspace and spectral methods for solving complementarity problems in many-body contact dynamics simulation. *Internat. J. Numer. Methods Engrg.* 95, 7 (2013).

Jonathan D. Hogg and Jennifer A. Scott. 2013. Pivoting Strategies for Tough Sparse Indefinite Systems. *ACM Trans. Math. Softw.* 40, 1, Article 4 (Oct. 2013).

Hanh M Huynh. 2008. *A large-scale quadratic programming solver based on block-LU updates of the KKT system*. Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. 2011. Bounded Biharmonic Weights for Real-Time Deformation. *ACM Trans. Graph.*, Article 78 (July 2011).

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

G. Karypis. 1997. METIS : Unstructured graph partitioning and sparse matrix ordering system. *Technical Report* (1997).

Danny M. Kaufman, Shinjiro Sueda, Doug L. James, and Dinesh K. Pai. 2008. Staggered Projections for Frictional Contact in Multibody Systems. *ACM Trans. Graph.*, Article 164 (Dec 2008).

David I.W. Levin. 2019. GAUSS Library. https://github.com/dilevin/GAUSS.

Joseph W.H. Liu. 1990. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.* 11, 1 (1990).

Christopher Mario Maes. 2011. *A regularized active-set method for sparse convex quadratic programming*. Ph.D. Dissertation. Stanford University, USA.

Istvan Maros and Csaba Mészáros. 1999. A repository of convex quadratic programming problems. *Optimization Methods and Software* 11, 1-4 (1999).

Jacob Mattingley and Stephen Boyd. 2012. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering* 13, 1 (2012).

Hans Mittelmann. 2020. Benchmarks for Optimization Software. Retrieved April 13, 2020 from http://plato.asu.edu/bench.html

ApS Mosek. 2015. The MOSEK optimization toolbox for MATLAB manual.

Jorge Nocedal and Stephen Wright. 2006. *Numerical optimization*. Springer Science & Business Media.

Gurobi Optimization. 2014. Inc.,"Gurobi optimizer reference manual," 2015.

Abhishek Goud Pandala, Yanran Ding, and Hae-Won Park. 2019. qpSWIFT: A Real-Time Sparse Quadratic Program Solver for Robotic Applications. *IEEE Robotics and Automation Letters* 4, 4 (2019).

Michael James David Powell. 1985. On the quadratic programming algorithm of Goldfarb and Idnani. In *Mathematical Programming Essays in Honor of George B. Dantzig Part II*. Springer.

L. Righetti and S. Schaal. 2012. Quadratic programming for inverse dynamics with optimal distribution of contact forces. In *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*.

Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics.

Olaf Schenk and Klaus Gärtner. 2002. Two-Level Dynamic Scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems. *Parallel Comput.* 28, 2 (Feb. 2002).

Olaf Schenk and Klaus Gärtner. 2006. On fast factorization pivoting methods for sparse symmetric indefinite systems. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]* 23 (2006). http://eudml.org/doc/127439

K Schittkowski. 2003. QL: A Fortran code for convex quadratic programming-User guide. *Report, Department of Mathematics, University of Bayreuth* (2003).

Michele Segata. 2019. MPC Library. https://github.com/michele-segata/mpclib.

Breannan Smith, Danny M. Kaufman, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. 2012. Reflections on Simultaneous Impact. *ACM Trans. Graph.* 31, 4, Article 106 (July 2012).

Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. 2020. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation* (2020).

Daniel Sýkora, Ladislav Kavan, Martin Čadík, Ondřej Jamriška, Alec Jacobson, Brian Whited, Maryann Simmons, and Olga Sorkine-Hornung. 2014. Ink-and-Ray: Bas-Relief Meshes for Adding Global Illumination Effects to Hand-Drawn Characters. *ACM Trans. Graph.* 33, 2, Article 16 (April 2014).

John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. 2014. XSEDE: accelerating scientific discovery. *Computing in science & engineering* 16, 5 (2014).

Andreas Wächter and Lorenz T. Biegler. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming* 106, 1 (01 Mar 2006).

Richard A Waltz and Jorge Nocedal. 2004. KNITRO 2.0 User's Manual. *Ziena Optimization, Inc.[en ligne] disponible sur http://www. ziena. com (September, 2010)* 7 (2004).

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer.

Nicholas J. Weidner, Kyle Piddington, David I. W. Levin, and Shinjiro Sueda. 2018. Eulerian-on-Lagrangian Cloth Simulation. *ACM Trans. Graph.* 37, 4, Article 50 (July 2018).

Elizabeth Lai Sum Wong. 2011. *Active-set methods for quadratic programming*. Ph.D. Dissertation. UC San Diego, USA.

Stephen J Wright. 1997. *Primal-dual interior-point methods*. Vol. 54. Siam.

Jiaxian Yao, Danny M. Kaufman, Yotam Gingold, and Maneesh Agrawala. 2017. Interactive Design and Stability Analysis of Decorative Joinery for Furniture. *ACM Trans. Graph.* 36, 4, Article 157a (March 2017).

Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. 2016. Interactively Cutting and Constraining Vertices in Meshes Using Augmented Matrices. *ACM Trans. Graph.* 35, 2, Article 18 (Feb. 2016).

Changxi Zheng and Doug L. James. 2011. Toward High-Quality Modal Contact Sound. *ACM Trans. Graph.*, Article 38 (July 2011).

Yufeng Zhu, Robert Bridson, and Danny M. Kaufman. 2018. Blended Cured Quasi-Newton for Distortion Optimization. *ACM Trans. Graph.* 37, 4, Article 40 (July 2018).