NASOQ: Numerically Accurate Sparsity-Oriented QP Solver

KAZEM CHESHMI, University of Toronto
DANNY KAUFMAN, Adobe research
SHOAIB KAMIL, Adobe research
MARYAM MEHRI DEHNAVI, University of Toronto

Quadratic programs (QP) are at the heart of algorithms in many domains including geometry processing, animation, and engineering. Accurate, reliable, efficient, and scalable solutions to QP problems are critical for the efficient and correct function of such methods. Available QP methods currently require choosing between accuracy and scalability. Some methods reliably solve QP problems to high accuracy but only work for small-scale QP problems due to their use of dense matrices. Alternatively, many other QP solvers scale well using 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 of QP problems. To enable NASOQ we construct a new row modification method along with a fast implementation of LDL factorization for indefinite systems, to efficiently update and solve the sequence of iteratively modified KKT system required for accurate QP solutions. While QP methods have been previously tested on large synthetic benchmarks, they have not been consistently tested on computer graphics problems which exercise QPs in challenging ways. To test NASOQ we thus collect 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 develop and test two NASOQ variants against the 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 percent of problems across a range of requested accuracies from 1×10^{-3} to 1×10^{-9} with average speedups ranging from 3.2× to 15.8× over the fastest competing methods.

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

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

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

1 INTRODUCTION

Solving a quadratic program (QP) is a core numerical task critical in domains spanning across geometry processing [Jacobson et al. 2011], animation [Jacobson et al. 2011; Righetti and Schaal 2012], physical simulation [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 well 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 [Zheng and James 2011].

The key challenge for solving QPs is to identify *active sets* [Fletcher 2013], which are the subset of a QP's linear inequality constraints that are treated as equalities at optimality. If an active set is known, the problem then reduces to solving a QP subject to just its active constraints.

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 [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 but also less-accurate methods for solving linear system that once again unacceptably reduce accuracy [Stellato et al. 2018].

To address these issues we construct NASOQ (Numerically Accurate Sparsity-Oriented QP Solver), a new, general-purpose, active-set algorithm for the combined accurate, efficient, and scalable solution of QPs. NASOQ is built upon three core contributions: a new LDL factorization algorithm called LBL for accurate and fast factorization of indefinite matrices that arise in KKT problems; a novel sparsity-oriented row modification method to enable fast factorization of subsequent KKT matrices using efficient updates to previously computed factors; and two new variants of the Goldfarb-Idnani (GI) active-set method [Goldfarb and Idnani 1983] with different trade-offs between speed and accuracy.

We design a new algorithm for rapid and accurate solutions of the many successively-updated KKT systems encountered during an active-set QP solve. This algorithm, which we call SoMod, uses a novel sparsity-oriented row modification method combined with LBL, an efficient factorization method for sparse symmetric indefinite systems, and an efficient triangular solve implementation that provides the solution for a specific system. 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. For the initial factorization, we utilize LBL, a novel implementation of the LDL factorization using Load-Balanced Level Coarsening [Cheshmi et al. 2018] for parallelization that provides state-of-the-art performance for solving KKT systems while providing the capability to precompute the required symbolic analysis for subsequent factorization updates.

With these building blocks we construct and analyze NASOQ, a new active-set QP algorithm. To consistently evaluate NASOQ with 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 2018; Weidner et al. 2018]. As we demonstrate in Section 6, for three different requested accuracies across this benchmark NASOQ is more accurate than other solvers by converging for 99.5% of problems while the best convergence of other solvers is 94%. NASOQ is also more efficient than exising QP solvers by providing an average speedup of 3.2×−15.8× for the three requested accuracy ranges compared to the fastest competing QP solvers.

Contributions. In summary, towards the efficient, accurate, and robust solution of convex QPs across scales and types we develop the following contributions:

- A new sparsity-oriented row modification method, part of the SoMod algorithm, that enables fast factorization of KKT matrices with efficient updates to previously-computed factors. This row modification method is used in SoMod to solve the many sequentially-updated KKT systems encountered in an active-set QP solve. It enables the reuse of previously-computed solutions for fast and scalable updates and is suitable for both inequality, equality, and mixed-constraint QPs.
- LBL, a new parallel LDL factorization method used inside SoMod for sparse symmetric indefinite systems that provides state-of-the-art efficiency and also enables the subsequent application of row modification.
- NASOQ, a new full-space QP algorithm for solving strictly convex quadratic programs at any scale. NASOQ converges accurately for different requested accuracy ranges and is faster than existing QP solvers for the problem instances in our benchmark set.
- A new QP benchmark composed of practical, real-world stress-test QPs taken from a wide range of geometry, simulation, and design applications combined with a prior QP benchmarks from the optimization literature for a practical analysis of QP performance.

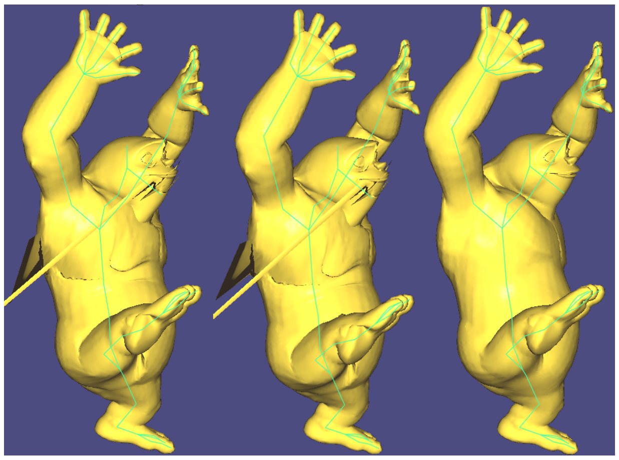
Fig. 1. The effect of accuracy in computing biharmonic weights on the visual appearance Gump after applying a similar deformation. The accuracy thresholds are set to 1×10^{-3}, 1×10^{-4}, and 1×10^{-6} in order from left to right. Gump is a mesh with 38833 vertices and 90 handles. Computing the biharmonic weights requires solving 90 QP problems.

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. Matrices H, A and C are often large and sparse.

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

```
Data: $H$, $q$, $A$, $b$, $C$, $d$
Result: $x$ , $y$, $z$
/* Feasibility phase                                   */
1  Initialize z0;
2  Solve Equation 7 to initialize x0, y0;
/* Optimality phase                                    */
3  while is not primal feasible do
4      Solve Equation 8 to compute descent $\Delta x$,$\Delta y$,$\Delta z$ ;
5      Compute dual and primal step length of t;
6      if $t = \infty$ then
7          Problem is unbounded.;
8      else
9          Update dual and/or primal variables;
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,y,z are optimal;
```

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 $z \in \mathbb{R}^m_+$ are the QP problem's dual Lagrange multiplier variables [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 four corresponding measures

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

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

$$\text{Complementarity:}\ |z_i (Cx - d)_i| < \epsilon, \quad \forall i \in [1, m] \tag{5}$$

$$\text{Non-negativity:}\ \ |\min(0, z_i)| < \epsilon, \quad \forall i \in [1, m] \tag{6}$$

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. Even small errors in constraint satisfaction lead to unacceptable failures in applications that depend on accurate constraint resolution at each call of a QP solve.

[1]Here $x \perp y$ is the *complementarity condition* $x_i y_i = 0$, $\forall$ corresponding entries i in vectors x and y.

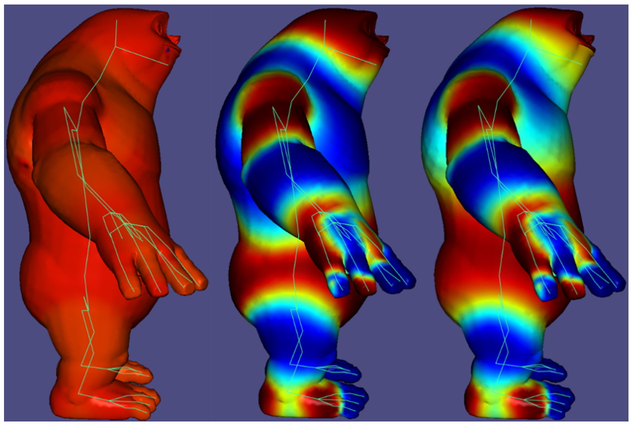

Fig. 2. Visual representation of some of the biharmonic weights that are stressed during the deformation shown in Figure 1. Dark red shows maximum effect and blue refers to zero weights. The weights are computed for 1×10^{-3}, 1×10^{-4}, and 1×10^{-6} in order from left to right.

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 how accurately 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 blow-ups and unacceptable modeling errors for engineering applications.

Complementarity measures the pairwise products of dual variables and their corresponding constraints and is important for correctly capturing active sets. In multi-body simulations [Erleben 2013; Heyn et al. 2013] dual variables often represent contact forces while constraints measure intersection constraints. Complementarity encodes the property that contact forces can not be applied unless objects are touching. Large violations of complementarity can create instabilities and visual artifacts of floating bodies with contact forces applying action at a distance.

Non-negativity then ensures dual variables are positive. Negative dual variables again can have serious consequences for stability and quality in applications. As an example, when computing bounded biharmonic weights [Jacobson et al. 2011] resultant skinning deformations when non-negativity is violated cause severe and unsightly artifacts, see Figures 1 and 2.

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. 2018] 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 a feasibility condition. Active-set methods are either primal-feasible, preserving the primal-feasibility condition or are dual-feasible, preserving the non-negativity condition. In this paper we focus on Goldfarb-Idnani (GI) [Goldfarb and Idnani 1983] which is a dual-feasible active-set solver where the initialization phase is less expensive to compute than primal-feasible methods [Wong 2011] .

High-level pseudocode of the GI method is shown in Algorithm 1. At the beginning (lines 1–2), the algorithm initializes an empty active-set proposal, $\mathcal{W} = \emptyset$ and zero for dual variables, $z_0 = 0$. The resulting initial KKT system to solve is then the indefinite linear system problem,

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

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

Then, at each successive active set iteration of the GI method, shown in lines 3–14 of Algorithm 1, the method improves the solution by updating the active set proposal $\mathcal{W}$ and so the corresponding active set constraint matrix $C_{\mathcal{W}}$ and its right-hand side constraint vector $c_{\mathcal{W}}$.

The method finds the next descent direction for the QP 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 updated by multiplying step length and the computed descent direction. The step length ensures that the activated constraint becomes primal-feasible and all dual variable remain dual-feasible. Each iteration's linear solve of the updated indefinite KKT system in (8) is expensive as QP system sizes and constraint numbers grow, but at the same time, the 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].

However, a key observation is that each update to $\mathcal{W}$ and correspondingly to the matrix in (8) is small and in fact generally involves 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 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 fill-in (additional nonzeros) 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 a direct factorization and back solve of each iteration's KKT system is possible via indefinite factorization methods [Maes 2011]. This leads to accurate and scalable solutions but is inefficient due to the repeated cost of recomputing the factorization. Using iterative methods such as Krylov subspace methods [Gould et al. 2001] instead of a direct factorization do not often happen due to difficulty of finding an effective preconditioned for KKT matrices with an aribtrary active-set [Maes 2011]. Reusing existing factorizations to compute the solution of the modified KKT system improves the speed of the QP solve. The only existing such method, CHOLMOD row modification [Davis and Hager 2005], is designed for symmetric positive definite (SPD) matrices and thus provides inaccurate solutions for 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. Solving a strictly convex quadratic programming problem is common in several computer graphics applications such as contact simulation [Barbic and James 2007; Righetti and Schaal 2012], shape deformation [Jacobson et al. 2011], and model reconstruction [Dvorožňák et al. 2018; Sýkora et al. 2014]. There are three general classes of algorithms for solving QP problems: barrier (also called interior-point), first-order, and active-set methods.

Barrier and first-order methods. Barrier methods [Gertz and Wright 2003; Mattingley and Boyd 2012; Mosek 2015; Optimization 2014; Pandala et al. 2019] use a weighted barrier function in the objective instead of inequality constraints, converting the inequality constraint QP problem into an equivalent equality-constrained nonlinear problem, and then solving it using Newton's method [El-Bakry et al. 1996]. Interior-point methods are popular for solving a QP problem in bounded number of iterations even though these methods require large amounts of work per iteration. Several interior-point based solvers exist including commercial solvers [Mosek 2015; Optimization 2014] and open source solvers [Gertz and Wright 2003]. Even though interior point methods bound the number of iterations required for convergence, they do not scale well and do not always meet the user-requested accuracy. First-order methods place all constraints into a single large KKT system and then use low cost iterations to compute the optimal solution using first order information from the cost function. Although first-order methods such as OSQP [Stellato et al. 2018] scale well for large-scale problems, they cannot always provide an accurate solution.

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 solution, 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 excessive computations in the QR and Schur complement factorization. Full-space methods [Gould et al. 2003; Huynh 2008], on the other hand, work directly with the KKT system. Factorizing 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 using Schur complement [Gould et al. 2003] or Block-LU method [Huynh 2008]. Both of these methods require large amounts of storage that limits their scalability and efficiency. However, full-space methods have the interesting property of preserving the sparsity pattern of the KKT system that enables re-using factors from previous iterations if an efficient approach of re-using factors exists. In this work we introduce a new full-space active-set method based of the GI algorithm that directly factorizes the successive KKT systems using SoMod. SoMod has a novel row modification and an indefinite factorization method that enables reusing the factorization of previous iterations for the solution of the KKT system in subsequent 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; Yeung et al. 2016] and is a subroutine in full-space QP solvers. Several existing factorization methods are designed for solving a sparse positive definite (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 may cause failures during factorization due to creating negative values under the square root when factorizing diagonals. Using these solvers with regularization [Herholz and Alexa 2018] prevents the factorization from failure but provides inaccurate solutions. Some existing indefinite factorization methods are square-root free [Golub and Van Loan 2012] but are slow such as Suitesparse LDL [Davis 2019] which is a single-thread implementation. Parallel indefinite solvers such as MKL Pardiso [Wang et al. 2014] and 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 describe LBL, a new parallel indefinite square-root free solver with pivoting that also enables modifying already-computed factors efficiently. LBL uses the parallelism strategy in [Cheshmi et al. 2018] in which the authors present an efficient parallel factorization. However, their proposed strategy is not applicable to indefinite factorization in which pivoting introduces new dependencies. LBL adapts the parallelism strategy of [Cheshmi et al. 2018] by using a restricted pivoting method [Schenk and Gärtner 2006] and structuring the factorization process so it does not limit parallelism.

L-factor modification. Modifying the L-factor to avoid re-factoring a symmetric matrix after a small change [Davis and Hager 1999, 2009] has received a lot of attention in computer graphics [Hecht et al. 2012; Herholz and Alexa 2018], circuit simulation [Davis and Hager 2005; Hager 1989], and in mathematical programming [Davis and Hager 2005]. In all of these applications, the linear system of equations changes by either a rank update/downdate (adding or subtracting the outer product of a row by itself) or a row modification. Most existing modification techniques [Davis and Hager 1999; Herholz and Alexa 2018] perform rank update/downdate ($A+w*w^T$) on a symmetric positive definite matrix. These types of modifications are not applicable to our active-set QP solver where a row/column is modified in each iteration. To the best of our knowledge, the only existing system with sparse row modification is CHOLMOD [Davis and Hager 2005] which is designed for SPD matrices and supports some indefinite systems. The major issue with CHOLMOD row modification is that it does not provide an accurate solution for most indefinite systems such as the KKT systems in this work, and therefore leads to failures in the QP solver. The proposed row modification in SoMod uses the sparsity pattern of constraint rows and modifies the L-factor of an indefinite factorization accurately and efficiently.

4 SOMOD: SPARSITY-ORIENTED ROW MODIFICATION

A scalable solution to a dual-feasible active-set QP depends on an efficient solution to the 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 application of 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),

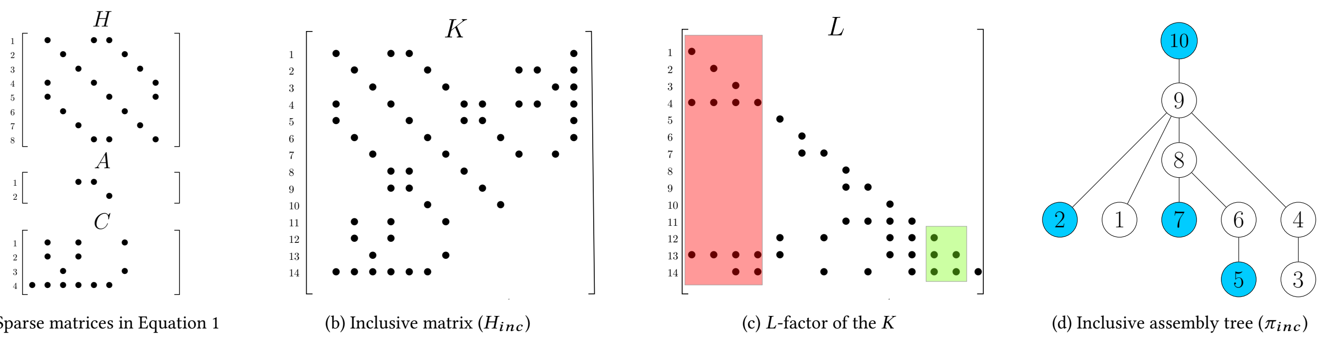
H
A
C
K
L
10
9
8
2
1
7
6
4
5
3
(a) Sparse matrices in Equation 1
(b) Inclusive matrix (H_{inc})
(c) L-factor of the K
(d) Inclusive assembly tree (π_{inc})

Fig. 3. The symbolic initialization phase of SoMod starts with creating an inclusive matrix, shown in Figure 3b from the matrices in Figure 3a 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 3b is computed and shown in Figure 3c. The colored columns show two supernodes, each containing more than one column. The remaining columns are supernodes each with the size of one. The corresponding inclusive (assembly) tree of the L-factor of Figure 3c is shown in Figure 3d. The colored nodes correspond to the inequality constraint rows (matrix C in Figure 3a). 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 3c.

P_{fill} is a fill-reducing ordering such as [Karypis and Kumar 1995], 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 creates *symbolic information* that guides the *numeric factorization*, which operates on the actual numeric values of K to compute the nonzero values of L and D. L and D are then used to find x with a lower triangular system solve [Davis 2006]. 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 from the initialization phase and its 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

The initialization phase solves the initial KKT problem using an LDL factorization, by first computing symbolic information that guides the numeric solution. In SoMod, unlike prior work, this initialization phase produces symbolic information that can be reused by subsequent factorizations in the factor modification phase, via using an *inclusive matrix* when performing analysis. 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 0. 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 3b shows an example of an inclusive matrix created from the matrices in Figure 3a.

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. Thus a node in the assembly tree represents a supernode, which is a group of columns, while a node in the elimination tree represents just a single column. 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.

The pruned inclusive assembly tree. The pruned inclusive assembly tree is built by removing some or all rows/columns of the inequality constraint matrix from the inclusive assembly tree and is represented with an array of parents denoted with π. Because the inclusive tree includes dummy entries for inequality constraints, they must be removed before performing the factorization in this phase. 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.

The initial KKT matrix factorized with only equality constraints in Equation 9 (see also line 2 of Algorithm 1) does not include the inequality constraint matrix C. Thus, the initial pruned inclusive tree is created by excluding all nodes corresponding to rows of C from the inclusive tree. Therefore, 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 3d, nodes 1, 3, and 6 can run in parallel, since none of them depend on lower non-colored nodes in the tree.

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 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 (lines 11-12). 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 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 2-13), 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 14). After obtaining the factorization, SoMod then uses triangular solve to

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.

```
Data: $K, L, D, super, \mathcal{M}$
Result: $L, D, P_S$
1  $T(:,:) = 0$
2  for $j \in \mathcal{M}$ do
3      $b = super_j$
4      $u = super_{j+1}$
5      $L_{:,b:u} = 0$
6      for $r \in L_{0:b,:}$ do
7          $T = T + L_{b:n,r} \times D_{r,r} \times L^T_{b:u,r}$
8      end
9      $[L_{b:u,b:u}, D_{b:u,b:u}, P_{S\,b:u,b:u}] = \mathrm{LDL}(K_{b:u,b:u} - T_{b:u,:})$
       /* Applying column permutation */
10     $L_{:,b:u} = L_{:,b:u} \times P_{S\,b:u,b:u}$
11     $LD = L_{b:u,b:u} \times D_{b:u,b:u}$
12     $L_{u:n,b:u} = LD \,/\, K_{u:n,b:u} - T_{u:n,:}$
13 end
   /* Applying row permutation */
14 $L = P_S \times L$
```

efficiently obtain a solution to the initial KKT problem, as described in Section 4.3.

4.2 Factor Modification

After the initialization phase, a large number of successive symmetric indefinite KKT systems are solved (line 4 of Algorithm 1). The factor modification phase in SoMod efficiently solves these successive KKT 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 2 (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.

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.

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

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 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 4b shows a pruned inclusive assembly tree with two already-added constraint rows 2 and 10 and Figure 4c 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.

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 the information of its ancestor to update the pruned inclusive tree and thus will need to search for its children in 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 4a shows the pruned inclusive tree with no constraints and Figure 4b 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 3d, 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.

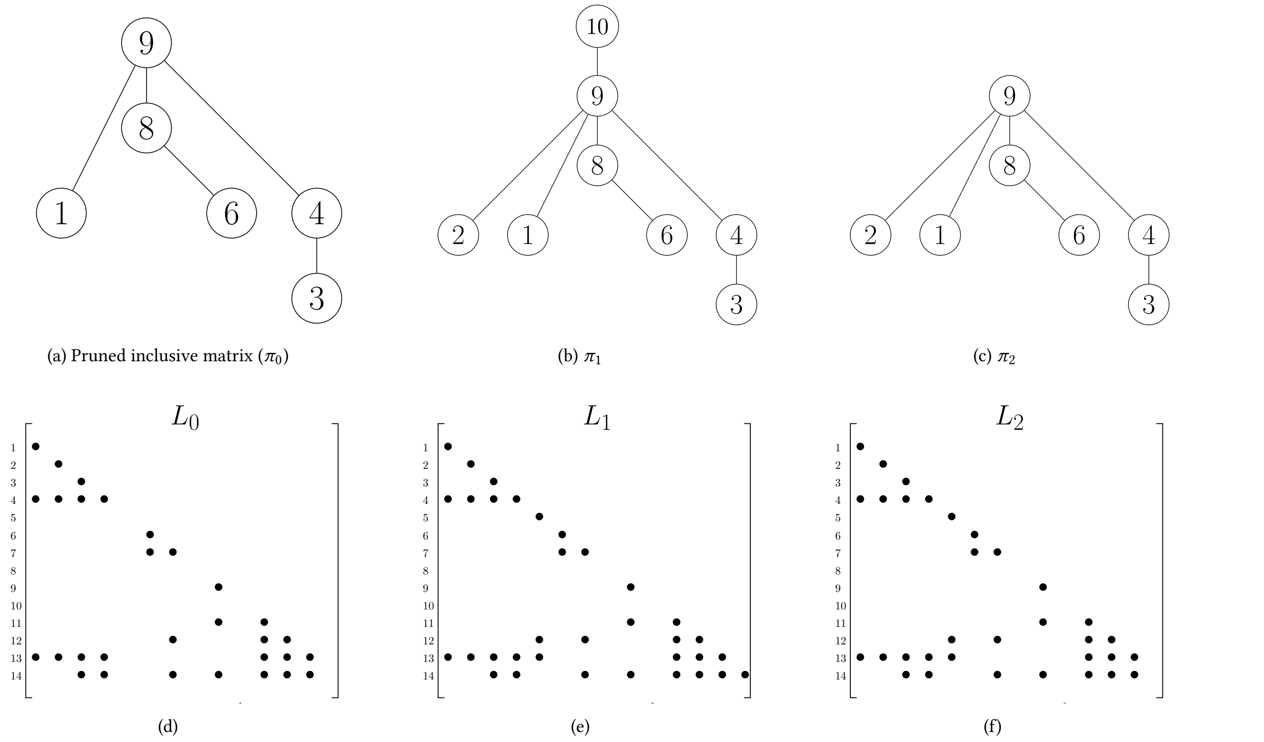

Fig. 4. An example for factor modification starting with the pruned inclusive tree in Figure 4a and the L-factor in Figure 4d from the initialization phase. All nodes corresponding to the inequality matrix are removed from the pruned inclusive matrix initially. SoMod symbolically adds rows corresponding 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 in Figure 4b. The corresponding supernodes in the L-factor in Figure 4e are also visible and will be updated using the numerical modification algorithm. Figure 4c is the result of removing node 10 from Figure 4b by using the symbolic row removal algorithm. Column 14 of the L-factor (which corresponds to node 10 in the tree) in Figure 4f becomes invisible after row removal.

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 3d 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. Algorithm 4 before numeric modification allows the numeric modification algorithm to involve values of the added row during the L-factor update.

4.3 Triangular Solve & 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 [Cheshmi et al. 2018] 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.

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

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

```
Data: $\pi$, $\pi_{inc}$, $v$, $r$, $k$
Result: $\pi$,$v$,$r$
/* Find the first visible parent of $k$ */
1  $f = \min \{j | j \in \rho_{inc}(k) \wedge v(j)\}$
2  if $f$ is a node then
       /* Find all nodes that $k$ is their least ancestor */
3      for $j \in \pi^{-1}(f)$ do
4          if $k \in \rho_{inc}(j)$ then
5              $\pi(j) = k$
6          end
7      end
8  else
       /* Look for any missing child in root nodes */
9      for $j \in r$ do
10         if $k \in \rho_{inc}(j)$ then
11             $\pi(j) = k$
12             $r = r - \{j\}$
13         end
14     end
15 end
/* Update the assembly tree based on the inclusive assembly tree */
16 $\pi(k) = f$
17 $v(k)$=true
```

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 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 development SoMod in place, we now can define our two closely related QP solution algorithms, NASOQ-Fixed and NASOQ-Tuned. Both methods similarly integrate our 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. Here our two variants, 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 together, the two methods each individually offer different balances in the trade-off between efficiency and accuracy in larger scale problems.

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  z0=0;
3  active-set=$\emptyset$;
4  modify=ADD;
5  SoMod.symbolic_initialization(); SoMod.LBL(q, b);
6  SoMod.solve();
/* Optimality phase */
7  while is not primal feasible do
8      if modify == ADD then
9          SoMod.symbolic_row_addition(act);
10         SoMod.numerical_modification(act);
11     else
12         SoMod.numerical_modification(drp);
13         SoMod.symbolic_row_removal(drp);
14     end
15     SoMod.solve();
16     Compute the step length t;
17     if $t = \infty$ then
18         Problem is unbounded.;
19     else
20         Update dual and/or primal variables;
21         Update the active-set;
22         add or remove a row to/from KKT;
23         Set modify to either ADD or REMOVE;
24     end
25     k = k + 1 ;
26 end
27 x,y,z are optimal;
```

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 direct and natural extension of NASOQ-Fixed.

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 row modification and LBL. This allows NASOQ-fixed to replace the standard GI dense and indirect solve using Cholesky and QR.

As a result, both the initial and all successive KKT system solves in GI are now computed with our direct, sparse KKT system solver implemented in SoMod. This is in place of the accurate but dense, indirect solvers previously applied in GI implementations.

In our construction of SoMod's LBL and row modification components there are three parameters we can expose that enable us to balance efficiency against accuracy for different applications, and problem scales. They are

- *max_iter*: the maximum number of refinement iterations that incrementally improves the solution of a KKT system after the solve phase, described in Section 4.3
- *diag_perturb*: the value added to some diagonals of the KKT matrix, as described in Equation 12 to control the numerical instability of LBL or help with row modification in SoMod.
- *stop_tol*: a threshold that defines an upper bound for the residual accuracy of a KKT system during the refinement phase as described in Section 4.3.

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 have sought the most effective values per input QP characteristics.

NASOQ-Fixed sets *diag_perturb* and *stop_tol* to fixed values *for all* input QP problems to respectively 1×10^{-9} and 1×10^{-15}. Here we find that per problem adapting *max_iter*, based on the requested accuracy ϵ is most effective (with the assumption that we will only have a single attempt at the solve). As lower accuracy ϵ's are requested, NASOQ-Fixed applies less 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

$$max\_iter = \begin{cases} 1, & \epsilon > 1 \times 10^{-4} \\ 2, & 1 \times 10^{-8} \leq \epsilon \leq 1 \times 10^{-4} \\ 3, & \epsilon < 1 \times 10^{-8} \end{cases} \quad (14)$$

Finally, for very small QP problems, i.e., fewer than 100 nonzeros, QPN-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, unlike barrier and first-order approaches, terminate in bounded time with respect to number of constraints, and, in practice generally much more rapidly [Maes 2011; Wright 1997]. Thus the cost of running multiple passes of NASOQ to determine whether a choice setting of our three above parameters successfully matches a requested accuracy is generally acceptable when accuracy is critical and some efficiency can be sacrificed.

NASOQ-Tuned thus sweeps through a set of empirically determined parameters that we find each work best across a wide range of QPs tested. NASOQ-Tuned begins with an initial pass of NASOQ-Fixed. Then, if the requested ϵ accuracy is not met it successively tries new NASOQ passes with the sequence parameters listed as different configurations in Table 1.

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

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

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

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).

Overall, across all QP problems in our repository, NASOQ converges for over 99.5% of the problems for accuracies ranging from 1×10^{-3} to 1×10^{-9} with average speedups ranging from 3.2× to 15.8× over the best competing method. For requested accuracies of 1×10^{-3} and 1×10^{-6}, NASOQ-Tuned has no failures. Only 21 problem instances (out of 1513, or 1.4%) fail to reach the 1×10^{-9} accuracy threshold. Our analysis shows that these few failures occur due to the numerical range of the problems themselves and that other solvers also fail to solve these problems. NASOQ demonstrates consistent efficiency and speedups across all application types, and the SoMod algorithm plays a critical role in the performance of NASOQ.

6.1 Experimental Setup

Testbed architecture. All experiments are performed on a 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 library 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.

Solver settings. We compare NASOQ against four widely-used state-of-the art QP solvers: OSQP [Stellato et al. 2018], Gurobi [Optimization 2014], MOSEK [Mosek 2015], and QL [Schittkowski 2003]. The tools are selected to represent different QP solver methods. OSQP uses a first-order method, supports sparse problems, and parallelism. Gorubi and MOSEK are both commercial tools based on barrier methods; both support parallel execution and sparse QP problems. To compare the performance NASOQ to an alternative that uses the Goldfarb-Idnani algorithm, we use QL, which also implements GI, though it does not support sparsity or 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 Equation 2 are changed[2].

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. 2018] Equation 34). This scaling leads to early termination and thus an inaccurate/non-optimal solution. For fair comparison, we changed OSQP's termination criteria to use the absolute requested accuracy[3].

Gurobi and MOSEK are two commercial solvers that use barrier methods to solve QP problems. For both packages, we use an Eigen wrapper from the code provided in [Weidner et al. 2018] and use the default settings for both solvers. We use the latest releases of Mosek and Gurobi, which are Mosek v8 and Gurobi v8.

QL is a dense active-set solver based on the GI algorithm, implemented in Fortran. QL works with dense matrices only, and thus we convert all sparse matrices to dense prior to using QL. The conversion time is not included in the reported solve time. However, large-scale QP problems cannot be converted due to memory limitations of the testbed architecture.

Performance profile. To compare the convergence speed of different solvers, a performance profile plot is often used [Dolan and Moré 2002; Pandala et al. 2019; Stellato et al. 2018; Wong 2011]. 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. $f_s(\tau)$ demonstrates the fraction of solved problems within $\tau\times$ the time of the best solver. Thus, in Figure 6 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.

6.2 Benchmark Repository for Sparse Quadratic Programming

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-Meszaros 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 problems or are tested for randomly generated problems [Pandala et al. 2019; Stellato et al. 2018]. 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 computer graphics problems; model predictive control (MPC) [Segata 2018] from robotics; and strictly convex QP problems from the Maros-Meszaros 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 [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 others 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 that accuracy is critical. Thus, as shown in Figure 5, NASOQ-Tuned always converges for requested accuracy thresholds of 1×10^{-3} and 1×10^{-6}, while NASOQ-Fixed fails for 1.2% of problems (there are 21 problem instances that NASOQ-Tuned fails for 1×10^{-9}; we explain this 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 6. The trade-off in efficiency for accuracy can be observed when measuring average speedups across all problems: NASOQ-Fixed is 1.5× faster that NASOQ-Tuned. Compared to other solvers, the convergence behaviour of both variants of NASOQ is consistently better than other solvers for both large- and small-scale problems (see Figure 5).

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 OQSP to refine accuracy and obtain solutions for some problems when the accuracy range is 1×10^{-9}. OSQP and OSQP-polished collectively

[2]The list of parameters related to user-requested accuracy for each solver is provided in supplemental material.
[3]In the supplemental material, we provide information on the parts of OSQP we modified.

solve 94% percent of all problems in our repository for accuracy ranges of 1×10^{-3}, 1×10^{-6}, 1×10^{-9} which is quite good, but still considerably less than the 99% obtained from NASOQ (see Figure 5). 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 1×10^{-3} and 1×10^{-6} is 4× and 15.8× respectively.

Gurobi, in contrast to NASOQ, has a high failure rate and is not scalable. 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. This is observable in Figure 6, where all performance profiles of Gurobi follow similar trends across different requested accuracies and different problem scales. For accuracies 1×10^{-3} and 1×10^{-6}, Gurobi's failure rate is similar to that of OSQP; however, compared to NASOQ, Gurobi fails in more problems. Also, Gurobi does not demonstrate good scaling and has a high failure rate for large-scale problems with lower requested error. For example, for the threshold of 1×10^{-9}, Gurobi fails for 42.25% of large-scale problems as shown in Figure 5.

Mosek is another barrier method that converges in a bounded number of computationally heavy iterations. Mosek does not scale well for accuracy thresholds lower than 1×10^{-3}. As shown in Figure 5, 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.

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 of the accuracy thresholds as shown in Figure 5. The figure also shows that the performance profile and efficiency of QL in Figure 6 is inferior compared to other QP solvers including NASOQ, due to its lack of support for sparsity and paralellism.

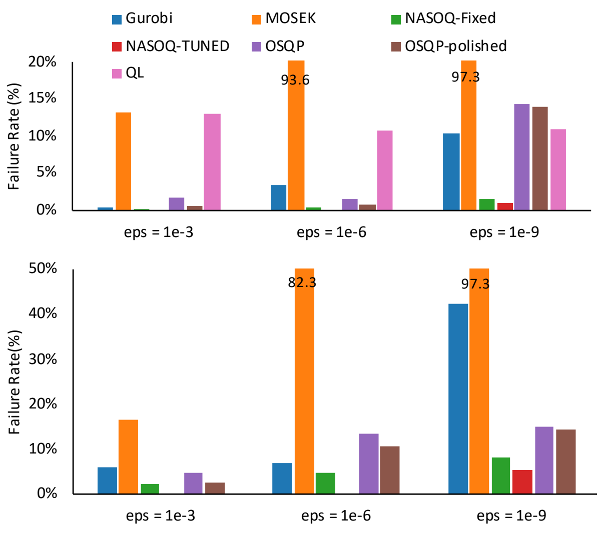

Fig. 5. Failure rate of NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (1×10^{-3}, 1×10^{-6}, and 1×10^{-9}) and for both small-scale (top) and large-scale (bottom) QP problems. NASOQ-tuned has the lowest failure rate compared to all other QP solvers for problems of different scales and for different requested accuracies.

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 1×10^{-6}; the trend holds for other accuracies. QL and MOSEK are not scalable 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 3× speedup across all contact simulation instances. Similar to the failure rate, OSQP's efficiency is also better than Gurobi.

In shape deformation and model reconstruction problems, NASOQ-Fixed and NASOQ-Tuned do not fail for any problems while OSQP and Gurobi fail in 12.5% of problem instances. NASOQ is 6× faster than both OSQP and Gurobi for more than 75% of these problems.

For Maros-Meszaros 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-Fixed and NASOQ-Tuned are on average 4.8× and 2.7× faster than Gurobi. OSPQ does not perform well for Maros-Meszaros problems (45% failure rate).

For 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-Tuned obtains an average speedup of 3.5× over OSQP.

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-Meszaros problems.

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 the NASOQ-range-space. Cholesky decomposition along with the QR decomposition are used instead of SoMod in

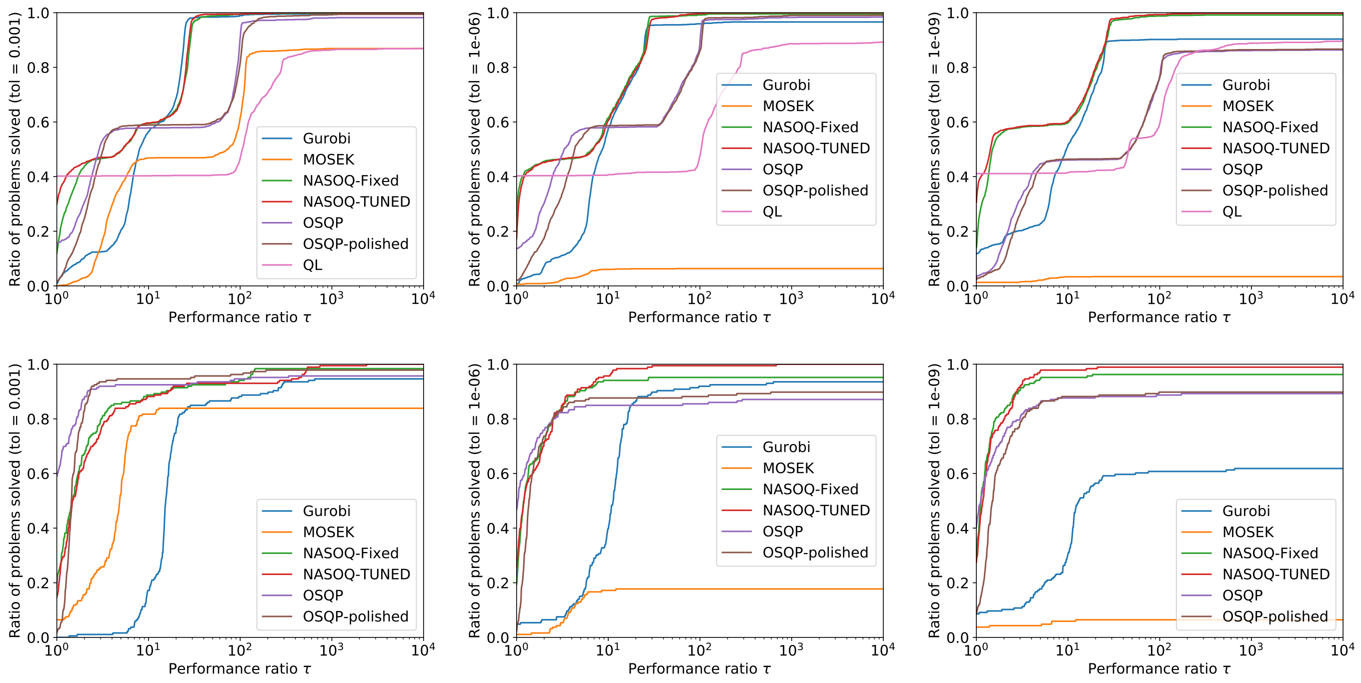

Fig. 6. Performance profiles for NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (from left to right: 1×10^{-3}, 1×10^{-6}, and 1×10^{-9}) and for small-scale (top) and large-scale (bottom) QP problems from our repository. Lines to the left are 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.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-9}$
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 $\epsilon = 1 \times 10^{-3}$ and $\epsilon = 1 \times 10^{-6}$.

NASOQ-range-space. However, due to the use of QR decomposition, 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 in 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 threshold of 1×10^{-9}; Gurobi is an exception, but it can only solve 2 of these 21 problems. In this section we discuss properties of these 21 problems instances and explore why existing QP solvers and NASOQ-Tuned fail to solve these problems.

These problem instances have a large *numerical range* which can be classified into two categories: *(1)* Problems that contain a large value (1×10^{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 Equation 1). This large numerical range limits the accuracy that QP solvers can achieve when operating 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 is a possible solution.

Gurobi is the only QP solver that converges for two of these problems instances. While NASOQ-Tuned is able to get close to the accuracy threshold of 1×10^{-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 1×10^{6}, which leads to inaccurate solutions for some intermediate KKT systems and thus leads to failure in NASOQ-Tuned.

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 7 compares the performance profile of NASOQ-Fixed for $\epsilon = 1 \times 10^{-9}$ 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. Overall, NASOQ-Fixed is faster than all other implementations with the least number of failures.

NASOQ-Fixed-CHOLMOD replaces SoMod's row modification and LBL phases with ROWMOD 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 matrices. However, by adding a perturbation value to the diagonal entries 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 7, 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. From Figure 7, NASOQ-Fixed is overall faster than NASOQ-Fixed-CHOLMOD with the least number of failures.

NASOQ-Fixed is on average 3× 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 QPN-Fixed-initial-MKL, 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.

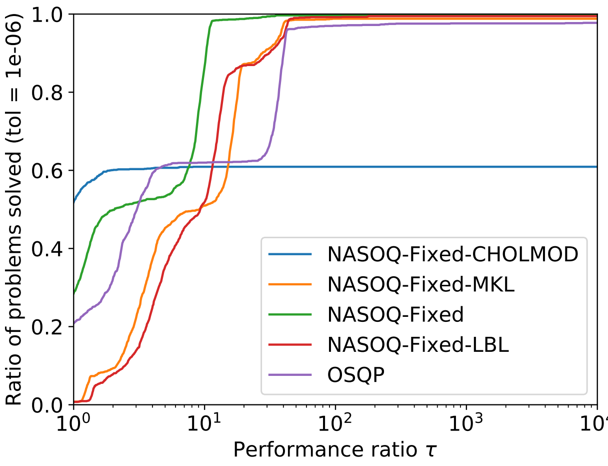

Fig. 7. Performance profile of NASOQ using SoMod (NASOQ-Fixed), using CHOLMOD row modification (NASOQ-Fixed-CHOLMOD), solving from scratch using LBL (NASOQ-Fixed-LBL), and solving from scratch using MKL (NASOQ-Fixed-MKL). OSQP is also shown as a reference solver. NASOQ-Fixed (blue line) performs better than the modified versions of NASOQ. Note that this performance profile contains both small and large QP instances, unlike Figure 6.

7 CONCLUSION & FUTURE WORK

In this work we propose NASOQ, a numerically-accurate efficient solver for quadratic programs arising in many domains such as graphics, animation, and geometry processing. We also create the largest-scale test set for QP problems from a variety of applications. We demonstrate that NASOQ with the help of the SoMod algorithm outperforms the best state-of-the-art QP libraries and accurately solves over 99% of problems in our test suite which exceeds the number of problems sets solved by competitive QP solvers. NASOQ is also faster than available state-of-the-art commercial and open source solvers.

In the future, we plan to explore the limits of NASOQ and find ways to mitigate the effects of numerical range on its convergence capabilities. We further wish to explore future applications of these building blocks on general equality-constrained problems, nonlinear optimization, and saddle-point systems, in computer graphics and other applications. We plan on releasing both NASOQ and our new benchmark suite for QP problems as open source projects to further the development and usage of fast, numerically-accurate QP solvers across all domains.

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*. 9558–9570.

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*. JMLR. org, 136–145.

Mario Arioli, Iain S Duff, Serge Gratton, and Stéphane Pralet. 2007. A note on GMRES preconditioned by a perturbed LDLˆT decomposition with static pivoting. *SIAM Journal on Scientific Computing* 29, 5 (2007), 2024–2044.

Jernej Barbic and Doug L James. 2007. Real-time reduced large-deformation models and distributed contact for computer graphics and haptics. *Carnegie Mellon University* (2007).

Michele Benzi, Gene H Golub, and Jörg Liesen. 2005. Numerical solution of saddle point problems. *Acta numerica* 14 (2005), 1–137.

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 TrendsÂő in Machine Learning* 3, 1 (2011), 1–122. https://doi.org/10.1561/2200000016

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), 1–14.

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, 13 pages. 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, 15 pages. 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), 1–25.

Timothy A Davis and William W Hager. 1999. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 606–627.

Timothy A Davis and William W Hager. 2005. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 26, 3 (2005), 621–639.

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), 1–23.

Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91, 2 (2002), 201–213.

Iain S Duff. 2004. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software (TOMS)* 30, 2 (2004), 118–144.

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 International Symposium on Sketch-Based Interfaces and Modeling*. 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), 507–541.

Kenny Erleben. 2013. Numerical methods for linear complementarity problems in physics-based animation. In *Acm Siggraph 2013 Courses*. 1–42.

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), 327–363.

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), 3080–3091.

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 Transactions on Mathematical Software (TOMS)* 29, 1 (2003), 58–81.

Philip E Gill, Walter Murray, Michael A Saunders, and Elizabeth Wong. 2005. UserâĂŹs 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), 1–36.

Donald Goldfarb and Ashok Idnani. 1983. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical programming* 27, 1 (1983), 1–33.

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

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), 1376–1395.

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), 353–372.

William W Hager. 1989. Updating the inverse of a matrix. *SIAM review* 31, 2 (1989), 221–239.

Florian Hecht, Yeon Jin Lee, Jonathan R Shewchuk, and James F O'Brien. 2012. Updated sparse cholesky factors for corotational elastodynamics. *ACM Transactions on Graphics (TOG)* 31, 5 (2012), 1–13.

Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

Philipp Herholz and Marc Alexa. 2018. Factor once: reusing cholesky factorizations on sub-meshes. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–9.

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), 541–561.

Jonathan D Hogg and Jennifer A Scott. 2013. Pivoting strategies for tough sparse indefinite systems. *ACM Transactions on Mathematical Software (TOMS)* 40, 1 (2013), 1–19.

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 Popovic, and Olga Sorkine. 2011. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.* 30, 4 (2011), 78.

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

George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

Danny M Kaufman, Shinjiro Sueda, Doug L James, and Dinesh K Pai. 2008. Staggered projections for frictional contact in multibody systems. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 164.

David Levin. 2019. GAUSS. Retrieved March 2, 2020 from https://github.com/dilevin/GAUSS

Joseph WH Liu. 1990. The role of elimination trees in sparse factorization. *SIAM journal on matrix analysis and applications* 11, 1 (1990), 134–172.

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

Istvan Maros and Csaba Mészáros. 1999. A repository of convex quadratic programming problems. *Optimization Methods and Software* 11, 1-4 (1999), 671–681.

Jacob Mattingley and Stephen Boyd. 2012. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering* 13, 1 (2012), 1–27.

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

Gurobi Optimization. 2014. Inc.,âĂIJGurobi optimizer reference manual,âĂİ 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), 3355–3362.

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, 46–61.

Ludovic Righetti and Stefan 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)*. IEEE, 538–543.

Yousef Saad. 2003. *Iterative methods for sparse linear systems*. Vol. 82. siam.

Olaf Schenk and Klaus Gärtner. 2002. Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Comput.* 28, 2 (2002), 187–197.

Olaf Schenk and Klaus Gärtner. 2006. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis* 23, 1 (2006), 158–179.

K Schittkowski. 2003. QL: A Fortran code for convex quadratic programming-UserâĂŹs guide. *Report, Department of Mathematics, University of Bayreuth* (2003), 64–71.

Michele Segata. 2018. mpclib. Retrieved March 2, 2020 from https://github.com/michele-segata/mpclib

Breannan Smith, Danny M Kaufman, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. 2012. Reflections on simultaneous impact. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 106.

Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. 2018. OSQP: An operator splitting solver for quadratic programs. In *2018 UKACC 12th international conference on control (CONTROL)*. IEEE, 339–339.

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 Transaction on Graphics* 33, 2 (2014), 16.

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 PhiâĎć*. Springer, 167–188.

Nicholas J Weidner, Kyle Piddington, David IW Levin, and Shinjiro Sueda. 2018. Eulerian-on-lagrangian cloth simulation. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–11.

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

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 Transactions on Graphics (TOG)* 36, 4 (2017), 157a.

Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. 2016. Interactively cutting and constraining vertices in meshes using augmented matrices. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–17.

Changxi Zheng and Doug L James. 2011. Toward high-quality modal contact sound. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 38.

Yufeng Zhu, Robert Bridson, and Danny M Kaufman. 2018. Blended cured quasi-newton for distortion optimization. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 40.