Les Cahiers du GERAD ISSN: 0711–2440

MADNCL: a GPU implementation of algorithm NCL for large-scale, degenerate nonlinear programs

A. Montoison, F. Pacaud, M. Saunders, S. Shin, D. Orban

G–2025–67

October 2025

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2025 – Bibliothèque et Archives Canada, 2025

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2025 – Library and Archives Canada, 2025

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

MADNCL: a GPU implementation of algorithm NCL for large-scale, degenerate nonlinear programs

Alexis Montoison [a, e]

François Pacaud [b]

Michael Saunders [c]

Sungho Shin [d]

Dominique Orban [e]

[a] *Mathematics and Computer Science division, Argonne National Laboratory Lemont, Lemont (IL) 60439, USA*

[b] *Centre Automatique et Systèmes, Mines Paris-PSL, 75006 Paris, France*

[c] *ICME and Management Science & Engineering, Stanford University, Stanford (CA) 94305, USA*

[d] *Department of Chemical Engineering, MIT, Cambridge (MA) 02139, USA*

[e] *Département de mathématiques et de génie industriel, Polytechnique Montréal & GERAD, Montréal, (Qc), Canada, H3T 1J4*

dominique.orban@gerad.ca

October 2025
Les Cahiers du GERAD
G–2025–67

Abstract : We present a GPU implementation of Algorithm NCL, an augmented Lagrangian method for solving large-scale and degenerate nonlinear programs. Although interior-point methods and sequential quadratic programming are widely used for solving nonlinear programs, the augmented Lagrangian method is known to offer superior robustness against constraint degeneracies and can rapidly detect infeasibility. We introduce several enhancements to Algorithm NCL, including fusion of the inner and outer loops and use of extrapolation steps, which improve both efficiency and convergence stability. Further, NCL has the key advantage of being well-suited for GPU architectures because of the regularity of the KKT systems provided by quadratic penalty terms. In particular, the NCL subproblem formulation allows the KKT systems to be naturally expressed as either stabilized or condensed KKT systems, whereas the interior-point approach requires aggressive reformulations or relaxations to make it suitable for GPUs. Both systems can be efficiently solved on GPUs using sparse $\mathrm{LDL}^{\mathrm{T}}$ factorization with static pivoting, as implemented in NVIDIA cuDSS. Building on these advantages, we examine the KKT systems arising from NCL subproblems. We present an optimized GPU implementation of Algorithm NCL by leveraging MadNLP as an interior-point subproblem solver and utilizing the stabilized and condensed formulations of the KKT systems for computing Newton steps. Numerical experiments on various large-scale and degenerate NLPs, including optimal power flow, COPS benchmarks, and security-constrained optimal power flow, demonstrate that MadNCL operates efficiently on GPUs while effectively managing problem degeneracy, including MPCC constraints.

Keywords : Nonlinear programming; augmented Lagrangian method; interior-point methods; constraint qualifications; degeneracy; KKT systems; MPCC constraints; graphics processing units

1 Introduction

We consider the nonlinear programming (NLP) problem

$$\text{NLP} \qquad \begin{array}{ll} \underset{x\in\mathbb{R}^n}{\text{minimize}} & \phi(x) \\ \text{subject to} & c(x) = 0, \quad \ell \le x \le u, \end{array}$$

where $\phi : \mathbb{R}^n \to \mathbb{R}$ is a smooth nonlinear objective function and $c : \mathbb{R}^n \to \mathbb{R}^m$ is a vector of smooth nonlinear constraint functions. We assume that ϕ and c are twice continuously differentiable and their derivatives are accessible, thereby enabling the use of second-order algorithms. Our focus is on an augmented Lagrangian method for large-scale instances of NLP problems on graphics processing units (GPU) hardware. In particular, we are interested in problems with degenerate constraints, i.e., for which a standard constraint qualification condition does not hold.

Augmented Lagrangian method. The augmented Lagrangian method (ALM) is one of the major NLP algorithms [37]. While sequential quadratic programming (SQP) and interior-point methods gained prominence in the 1980s because of their fast convergence properties [37], ALM, first devised in the 1960s [28, 41], has the key advantage of robustly handling degenerate optimization problems and is capable of detecting infeasibility quickly [14]. We refer to [10, 11, 12] for comprehensive surveys on ALM. The augmented Lagrangian penalty can be interpreted as a quadratic regularization in the dual [41]. Hence, ALM subproblems are nondegenerate regardless of the reformulation being used. This property enables ALM to achieve superior robustness, especially for NLP problems with redundant constraints. Because of its robustness, ALM has witnessed a resurgence in the 2010s, coinciding with new methodological breakthroughs [24, 17, 30, 34]. It has been proven to exhibit fast local convergence under minimal assumptions [22]. Software-wise, ALM has been utilized effectively across a wide range of solver implementations, including the classical LANCELOT [15, 16] and MINOS [36] solvers. Most of these improvements have been aggregated into the solver ALGENCAN [2], which now offers competitive performance relative to LANCELOT. In parallel, there has been recent interest in merging the augmented Lagrangian with interior-point method (IPM) [6, 31].

There are two main approaches regarding the subproblem formulations within ALM, which greatly affect the numerical treatment of the subproblem solution procedures. The classical implementations (e.g., LANCELOT) are based on bound-constrained Lagrangian (BCL) subproblems:

$$\text{BC}_k \qquad \begin{array}{ll} \underset{x\in\mathbb{R}^n}{\text{minimize}} & \phi(x) - y_k^T c(x) + \frac{1}{2}\rho_k \|c(x)\|^2 \\ \text{subject to} & \ell \le x \le u, \end{array}$$

where $y_k \in \mathbb{R}^m$ is an estimate of Lagrange multipliers for the equality constraints $c(x) = 0$, $\rho_k > 0$ is a penalty parameter, and $k = 1, 2, \ldots, p$ is an iteration counter. In BC_k, the nonlinear constraints are included in the quadratic penalty term of the objective function, and the bound constraints are treated directly. This structure provides an advantage in applying active-set approaches, such as projected gradient or projected Newton methods, to solve the subproblems. Further, only the objective changes between two consecutive ALM iterations, enabling effective warm-starting of active-set methods. A disadvantage of the BCL formulation is that the quadratic penalty term can lead to fully dense Lagrangian Hessians when a dense row exists in the Jacobian of $c(x)$. Large dense Hessians can be challenging to handle unless their internal structure is exploited.

In Algorithm NCL [34], subproblem BC_k is formulated as

$$\text{NC}_k \qquad \begin{array}{ll} \underset{x\in\mathbb{R}^n,\, r\in\mathbb{R}^m}{\text{minimize}} & \phi(x) + y_k^T r + \frac{1}{2}\rho_k \|r\|^2 \\ \text{subject to} & c(x) + r = 0, \quad \ell \le x \le u, \end{array}$$

where the added free variables r render the nonlinear constraints linearly independent. Although NC_k is mathematically equivalent to BC_k, its solution needs a different numerical implementation.

In particular, it contains nonlinear constraints and many new variables r, requiring an IPM solver. The key advantage of the NCL formulation arises from its more flexible handling of constraints: the new variables r play the role of a natural regularizer and enable solution of problems with constraint degeneracy (violation of LICQ).

The performance of an ALM solver depends directly on the algorithms used to solve the subproblems. The outer iterations of ALM are typically simple factorization-free operations, and thus, most of the computational effort is spent on solving the subproblems. In NCL, the IPM subproblem solver for NC_k spends most of its time solving Karush-Kuhn-Tucker (KKT) systems to compute search directions. Thus, efficient subproblem solution strategies are key to the efficient implementation of NCL.

GPU implementations of optimization solvers. As GPUs become more prevalent in scientific computing, there is a stronger incentive to leverage GPU-accelerated routines inside optimization solvers. Current optimization solvers are largely based on algorithms developed in the 1980s and 1990s, and were not designed to take advantage of GPU architecture. Consequently, most optimization solvers are sequential in nature, utilizing parallel operations only at the linear algebra level (e.g., multi-thread parallelism within BLAS routines or in the linear solver used to compute each Newton direction). Given these limitations, the last several years have seen a surge in efforts to harness GPUs to accelerate the solution of large-scale optimization problems, resulting in several notable breakthroughs.

Recent developments in mathematical optimization on GPUs can be classified into two concurrent threads. The first thread employs first-order or factorization-free methods, with a focus on convex optimization problems. We refer to the recent GPU implementation of PDLP (Primal-Dual Hybrid Gradient for LP) to solve large-scale LPs [33, 32] and the ADMM algorithm implemented in OSQP [44]. These methods rely on efficient factorization-free implementations to leverage parallelism. For example, PDLP can be implemented using sparse matrix-vector multiplication, which is amenable to parallelization with efficient implementations on GPUs (e.g., via cuSPARSE or rocSPARSE libraries) [33]. Similarly for projected gradient-based implementations of operator-splitting methods [44].

The second thread of research focuses on second-order methods with direct sparse solvers, which have gained popularity for nonconvex NLPs (where first-order or factorization-free approaches are not effective). In this approach, IPM is used as a skeleton, and various linear algebra tricks are employed to handle the associated KKT systems. Until recently, the sparse factorizations available on GPUs were notoriously slow [46]. Consequently, solvers for general NLPs were not available, and domain-specific implementations (e.g., exploiting a nonsingular block within the constraint Jacobian) have been investigated [39]. This has changed with the recent release of NVIDIA cuDSS, an efficient general-purpose sparse linear solver for GPUs. Although cuDSS cannot be directly utilized within existing NLP solvers because of inefficiencies in handling indefinite systems, two reformulation strategies can be employed to enforce GPU-amenable structures in the KKT system: hybrid KKT system [40] and lifted KKT system [45]. With these reformulations, the KKT systems can be solved effectively on GPUs, making the implementation of a general-purpose GPU IPM solver practical. An example is MadNLP [45], which we use as a stand-alone IPM NLP solver, and as an NCL subproblem solver within MadNCL (our implementation of Algorithm NCL).

Although the lifted and hybrid KKT system reformulation strategies enable the solution of KKT systems on GPUs, they cannot achieve the same degree of robustness as the classical CPU implementations of IPM solvers, such as Ipopt or KNITRO. As a consequence, the default optimality tolerance for the GPU version of MadNLP is relaxed to 10^{-6}. This is significantly looser than the 10^{-8} tolerance in mature IPMs. The primary reason for this compromised solver robustness stems from the significantly increased condition number of the KKT system resulting from the condensation processes used within these approaches [38]. This is currently recognized as a fundamental limitation of GPU-accelerated IPM solvers.

Goal. Our main research question is:

> *Can a GPU implementation of Algorithm NCL overcome the limitations of current IPM GPU implementations in terms of robustness and achievable optimality tolerance while maintaining similar speed?*

We aim to answer this question by (i) developing efficient strategies to solve the KKT systems arising from NCL subproblems on GPUs, and (ii) demonstrating that MadNCL, the GPU implementation of NCL, can achieve competitive performance relative to existing GPU implementations of IPM solvers while maintaining robustness, particularly for degenerate problems. We develop these strategies by adapting the recent condensed-space method of [38] to handle the KKT systems within NCL subproblems. The condensed KKT systems within IPMs and NCL subproblems share substantial similarity, allowing us to leverage existing GPU-accelerated linear algebra kernels implemented in MadNLP. Further, the structure of subproblem NC_k leads naturally to the formulation of sparse *condensed KKT* or sparse *stabilized KKT* matrices, which can be factorized using static pivoting, respectively by the Cholesky and $\mathrm{LDL}^{\mathrm{T}}$ algorithms implemented in GPU solvers like cuDSS. In this article, we provide an optimized GPU implementation of ALM based on the NCL subproblem formulation, which we refer to as MadNCL, by leveraging the GPU-accelerated solver MadNLP [45] as an IPM subproblem solver and utilizing *condensed* and *stabilized* formulations of the KKT systems for computing Newton steps. Our numerical results show that when running on the GPU, NCL offers a more robust alternative to the condensed-space method of [38]. In particular, we demonstrate that although ALMs can be slightly slower than condensed-space IPM approaches, they are significantly more robust at solving degenerate problems, overcoming the limitations of existing GPU-accelerated IPM solvers.

Contributions. Our main contributions follow.

- We provide for the first time a GPU ALM implementation for solving large-scale NLPs. Notably, the previous implementations of Algorithm NCL used the IPM solvers Ipopt and KNITRO to solve subproblems NC_k [34, 35]. Instead, we use MadNLP [45], a GPU IPM implementation with a filter line search algorithm [48]. MadNCL has full control over MadNLP's internals, enabling it to exploit the structure of the KKT systems in the subproblems NC_k for more efficient linear algebra operations.
- We present two different KKT system formulations, called the *stabilized KKT system* (K_{2r}) and the *condensed KKT system* (K_{1s}), adopting the naming convention introduced in [23]. We show that both systems can be factorized efficiently on GPUs without the use of numerical pivoting. As a consequence, the K_{2r} and K_{1s} systems can be solved on the GPU by NVIDIA cuDSS. Furthermore, these reformulations inherit established properties of the commonly used augmented KKT system (K_2) formulation, thereby guaranteeing descent directions for the NCL subproblems without introducing additional computational overhead.
- Numerically, we demonstrate that NCL can effectively take advantage of GPU parallelism and exhibit superior robustness. In particular, NCL can be implemented seamlessly on a GPU, provided a GPU-based IPM solver like MadNLP is available, and the resulting MadNCL offers competitive performance relative to other GPU optimization solvers [38]. Further, we show that MadNCL is able to solve large real-world degenerate problems, such as large-scale mathematical programs with complementarity constraints (MPCC).

Outline. The remainder of this paper is organized as follows. Section 2 provides a brief overview of ALM and the formulation of NCL subproblems. Section 3 describes the KKT systems arising within NCL subproblems and their condensed forms. Section 4 outlines implementation details of MadNCL, our GPU implementation of ALM based on the NCL subproblem formulation. Section 5 presents numerical results for MadNCL on various degenerate NLPs. Section 6 concludes and discusses future work.

2 Augmented Lagrangian Method

By design, ALM is a two-level method: the penalty parameter and Lagrange multipliers are updated during an *outer loop* (see §2.1), whereas the subproblems BC_k or NC_k are solved in an *inner loop* (see §2.2). Our implementation MadNCL merges these two loops to improve the overall performance, following [6].

2.1 ALM outer loop

ALM solves the augmented Lagrangian subproblem with a fixed penalty parameter $\rho_k > 0$ and a multiplier estimate $y_k \in \mathbb{R}^m$. Updating either the multipliers or the penalty parameter depends on the observed primal infeasibility just after completion of the current inner loop (solution of the subproblem).

Upon obtaining a solution (x_{k+1}, r_{k+1}) for the augmented Lagrangian subproblem, ALM updates the multipliers y_k if the primal infeasibility is below a specified accuracy level η_k; otherwise, it increases the penalty parameter ρ_k. The following is a commonly used update rule:

$$(y_{k+1},\ \rho_{k+1}) = \begin{cases} (y_k - \rho_k c(x_{k+1}),\ \rho_k) & \text{if } \|c(x_{k+1})\|_\infty \le \eta_k, \\ (y_k,\ 10\rho_k) & \text{otherwise.} \end{cases} \tag{1}$$

In this update rule, the multipliers y_k are updated by subtracting the primal infeasibility $c(x_{k+1})$ scaled by the penalty parameter ρ_k whenever the primal infeasibility is sufficiently small. This can be interpreted as a gradient ascent step in the dual space, with the step size determined by the penalty parameter ρ_k [41]. If the primal infeasibility is not sufficiently small, the penalty parameter ρ_k is increased by a factor of 10 to encourage the next subproblem to exhibit less infeasibility. For nonconvex problems, sufficiently large penalty parameters must be employed to ensure that the subproblem can recover the exact solution when the multiplier estimates are accurate [37]. While traditional ALM implementations utilize a projected Newton method [16, 37] to solve BC_k, the solution of NC_k requires the use of an IPM solver, namely MadNLP here.

2.2 IPM inner loop

The nonlinearly constrained Lagrangian (NCL) subproblem formulation [34] adapts the classical BCL formulation by introducing free variables $r \in \mathbb{R}^m$ and equality constraints. This reformulation yields the *nonlinearly constrained subproblem* NC_k. Although NC_k is much larger than BC_k, its structure is more amenable to efficient solution using an IPM: the variables r render the constraints linearly independent. Further, as the parameters ρ_k and y_k appear only in the objective, it is easier to warm-start an IPM to solve NC_k with new parameters ρ_{k+1} and y_{k+1} derived from the usual update rule (1), as noted in [35].

For fixed parameters (y_k, ρ_k), let $y \in \mathbb{R}^m$ be the multipliers associated with the equality constraints in NC_k, and $(z_l, z_u) \in \mathbb{R}^n \times \mathbb{R}^n$ be the nonnegative multipliers related to the bound constraints on x. The Lagrangian for NC_k is

$$\mathcal{L}(x, r, y, z_l, z_u) = \phi(x) + y_k^\top r + \tfrac{1}{2}\rho_k \|r\|^2 - y^\top (c(x) + r) - z_l(x - \ell) - z_u(u - x), \tag{2}$$

and the associated KKT conditions are

$$\begin{aligned} \nabla\phi(x) - \nabla c(x)^\top y - z_l + z_u &= 0 \\ y_k + \rho_k r - y &= 0 \\ c(x) + r &= 0 \\ 0 \le x - \ell \perp z_l &\ge 0 \\ 0 \le u - x \perp z_u &\ge 0. \end{aligned} \tag{3}$$

Algorithm NCL [34] applies Newton's method to (3), globalized here with a filter line search [48]. Once a primal solution (x_{k+1}, r_{k+1}) is found, parameters y_k and ρ_k are updated based on rule (1), and we proceed by solving a new subproblem NC_{k+1}.

2.3 Fusion of ALM outer loop and IPM inner loop

MadNCL fuses the outer and inner loops described in §2.1 and in §2.2. In addition, MadNCL uses an extrapolation step to converge asymptotically at a superlinear rate [6, 21]. The complete algorithm is detailed in Algorithm 1.

Algorithm 1 MadNCL

Initialize variable x_0.
Set primal and dual tolerances $(\eta_\star, \omega_\star)$.
Set $\gamma = 0.05$, $\tau = 1.99$, $\mu_{\text{fac}} = 0.2$, $\rho_{\max} = 10^{14}$, $\theta = 0.5$.
Set initial parameters $\rho_0 \leftarrow 100$, $\mu_0 \leftarrow 0.1$.
Compute initial dual variable y_0 using least-squares.
for $k = 1, 2, \ldots$ **do**
  Solve $\nabla_w F_k(w_k) d_k + F_k(w_k) = 0$ and set $w_k^+ = w_k + \alpha_k d_k$.
  if $\|F_k(w_k^+)\|_\infty \le \theta \|F_k(w_k)\|_\infty + 10\alpha_k^{0.2}\mu_k$ **then**
    Set $w_{k+1} = w_k^+$
  else
    Find w_{k+1} satisfying $\|F_k(w_{k+1})\|_\infty \le \omega_k$ using MadNLP.
  end if
  if $\|r_{k+1}\|_\infty \le \eta_k$ **then**
    $y_{k+1} \leftarrow y_k + \rho_k r_{k+1}$
    $\mu_{k+1} \leftarrow \min\{(\mu_k)^\tau, \mu_{fac} \times \mu_k\}$
    $\eta_{k+1} \leftarrow \min\{(\mu_{k+1})^{1.1}, 0.1 \times \mu_k\}$
    $\omega_{k+1} \leftarrow 100 \times (\mu_{k+1})^{1+\gamma}$
    $\rho_{k+1} \leftarrow \rho_k$
  else
    $\rho_{k+1} \leftarrow \min\{\rho_{\max}, 10\rho_k\}$
    $(y_{k+1}, \mu_{k+1}) \leftarrow (y_k, \mu_k)$
    $(\eta_{k+1}, \omega_{k+1}) \leftarrow (\eta_k, \omega_k)$
  end if
  if $\|r_{k+1}\|_\infty \le \eta_\star$ and $\|\nabla f(x_{k+1}) - \nabla c(x_{k+1})^\top y_{k+1}\|_\infty \le \omega_\star$ **then**
    Solution is locally optimal, stop
  end if
  if $\rho \ge \rho_{max}$ and $\|r_{k+1}\|_\infty > \eta_\star$ **then**
    Problem is locally infeasible, stop
  end if
end for

Inner iteration. For a given barrier parameter $\mu_k > 0$, multiplier estimate y_k, and penalty ρ_k, IPM reformulates the KKT equations (3) using a homotopy continuation method. Let $w := (x, r, y, z_l, z_u)$ denote the vector of primal-dual variables. For primal variables in the strict interior, where $(x - \ell,\ u - x) > 0$, the inner iterations solve the following system for w:

$$F(w, \rho_k, y_k, \mu_k) = \begin{bmatrix} \nabla\phi(x) - \nabla c(x)^\top y - z_l + z_u \\ y_k + \rho_k r - y \\ c(x) + r \\ Z_l(x - \ell) - \mu_k e \\ Z_u(u - x) - \mu_k e \end{bmatrix} = 0, \tag{4}$$

with $Z_l = \mathrm{diag}(z_l)$ and $Z_u = \mathrm{diag}(z_u)$. As (4) is a smooth system of nonlinear equations, the inner IPM iterations utilize a globalized Newton method to solve it. For a specified tolerance ω_k, the next primal-dual iterate w_{k+1} solves

$$\|F(w_{k+1}, \rho_k, y_k, \mu_k)\|_\infty \le \omega_k. \tag{5}$$

To simplify notation, we define $F_k(w) := F(w, \rho_k, y_k, \mu_k)$.

Extrapolation step. Once it is close to convergence, MadNCL applies an extrapolation step to achieve fast superlinear convergence [5, 21]. Let $\theta \in (0,1)$ and $\varepsilon_k > 0$. Before solving (5), the extrapolation step solves the linear system

$$\nabla_w F_k(w_k) d_k + F_k(w_k) = 0 \tag{6}$$

and sets $w_k^+ = w_k + \alpha_k d_k$, with α_k computed using a fraction-to-boundary rule ensuring that w_k^+ remains strictly feasible (i.e., $\ell < x_k^+ < u$ and $(z_{\ell,k}^+, z_{u,k}^+) > 0$). If

$$\|F_k(w_k^+)\|_\infty \leq \theta \|F_k(w_k)\|_\infty + \varepsilon_k \,, \tag{7}$$

we set $w_{k+1} = w_k^+$ directly. Otherwise, we perform inner iterations to find an iterate w_{k+1} satisfying (5). Following [5, Section 5.3], we set $\varepsilon_k = 10\alpha_k^{0.2}\mu_k$.

In other words, if w_k^+ makes sufficient progress towards optimality, we discard the inner iteration and move directly to the next outer iteration by setting $w_{k+1} = w_k^+$. This ensures a full Newton step close to optimality, resulting in a superlinear rate of convergence in the final iterations [7].

Outer iteration. The outer iterations are a variant of the implementation in LANCELOT, as we simultaneously update the barrier parameter with the other augmented Lagrangian parameters (1). The update rules for tolerances η_k and ω_k are adapted from the Superb algorithm [26].

3 Analysis of the KKT systems

In this section, we analyze the linear systems arising from the NCL subproblem formulation and demonstrate how they can be transformed into a *stabilized KKT system* and a *condensed KKT system*, suitable for efficient GPU implementation.

3.1 Newton system

In Algorithm 1 (MadNCL), the inner iterations consist in applying Newton's method to (4) with fixed parameters (y_k, ρ_k, μ_k). Successive Newton iterations lead to a sequence of KKT systems. In deriving these linear systems, we explicitly distinguish between equality and inequality constraints to exploit the block structure associated with slack variables. Therefore, in this section, we base our analysis on the following Nonlinear Constrained Optimization (NCO) problem:

NCO	$\underset{t}{\text{minimize}}$	$f(t)$
	subject to	$c_{\mathcal{E}}(t) = 0, \quad \ell_s \leq c_{\mathcal{I}}(t) \leq u_s, \quad \ell_t \leq t \leq u_t.$

Compared to problem NLP, we treat equality and inequality constraints separately. This formulation can be transformed into problem NLP by setting

$$\phi(x) = f(t), \quad c(x) = \begin{bmatrix} c_{\mathcal{E}}(t) \\ c_{\mathcal{I}}(t) - s \end{bmatrix} = 0, \quad x = \begin{bmatrix} t \\ s \end{bmatrix}, \quad \ell = \begin{bmatrix} \ell_t \\ \ell_s \end{bmatrix}, \quad u = \begin{bmatrix} u_t \\ u_s \end{bmatrix}.$$

We now derive the various Newton systems. Assuming $x - \ell$ and $u - x$ are nonnegative, Algorithm 1 finds the Newton descent direction $\Delta w = (\Delta x, \Delta r, \Delta y, \Delta z_l, \Delta z_u)$ as the solution of

$$\nabla_w F(w, \rho_k, y_k, \mu_k) \Delta w = -F(w, \rho_k, y_k, \mu_k). \tag{8}$$

Note that Δw corresponds to d_k in Algorithm 1. This gives the linear system

$$\begin{bmatrix} H & 0 & -J^\top & -I & I \\ 0 & \rho_k I & -I & 0 & 0 \\ J & I & 0 & 0 & 0 \\ Z_l & 0 & 0 & X - L & 0 \\ -Z_u & 0 & 0 & 0 & U - X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta r \\ \Delta y \\ \Delta z_l \\ \Delta z_u \end{bmatrix} = - \begin{bmatrix} \nabla \phi(x) - J^\top y - z_l + z_u \\ y_k + \rho_k r - y \\ c + r \\ Z_l(x - \ell) - \mu_k e \\ Z_u(u - x) - \mu_k e \end{bmatrix}, \tag{K_3}$$

where $c = c(x)$, $J^\top = J(x)^\top = [\nabla c_1(x) \ \cdots \ \nabla c_m(x)]$, $S = \operatorname{diag}(s)$, $X = \operatorname{diag}(x)$, $L = \operatorname{diag}(\ell)$, $U = \operatorname{diag}(u)$, $Z_l = \operatorname{diag}(z_l)$, $Z_u = \operatorname{diag}(z_u)$, and H is the Hessian of the Lagrangian $\mathcal{L}(x, r, y, z_l, z_u)$ (with respect to x):

$$H = \nabla^2\phi(x) - \sum_{j=1}^{m} y_j \nabla^2 c_j(x).$$

Eliminating

$$\begin{aligned} \Delta z_l &= -(X - L)^{-1}\big(Z_l\,\Delta x - \mu_k e\big) - z_l, \\ \Delta z_u &= (U - X)^{-1}\big(Z_u\,\Delta x + \mu_k e\big) - z_u \end{aligned} \tag{9}$$

leads to the equivalent symmetrized Newton system (used by default in previous implementations of NCL [34, 35]):

$$\begin{bmatrix} \hat{H} & 0 & J^\top \\ 0 & \rho_k I & I \\ J & I & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta r \\ -\Delta y \end{bmatrix} = - \begin{bmatrix} \nabla\phi(x) - J^\top y - (X - L)^{-1}\mu_k e + (U - X)^{-1}\mu_k e \\ y_k + \rho_k r - y \\ c + r \end{bmatrix}, \tag{10}$$

where $\hat{H} = H + \Sigma$ and $\Sigma := (X - L)^{-1}Z_l + (U - X)^{-1}Z_u$ is diagonal.

To get a valid search direction $(\Delta x, \Delta r, -\Delta y)$, the filter line-search algorithm requires that (10) is nonsingular and the Hessian projected onto the null space of Jacobian J is symmetric and positive definite (SPD). This is equivalent to stating that the inertia (the number of positive, negative, and zero eigenvalues) of (10) is equal to $(n+m, m, 0)$ [9, 25]. The free variables r guarantee that the constraint Jacobian $\begin{bmatrix} J & I \end{bmatrix}$ has full row rank, ensuring that the matrix in (10) has no zero eigenvalues. To obtain the correct inertia, inertia-correction methods add a primal regularization term δ_k to the upper-left blocks, replacing the matrix in (10) by the regularized matrix

$$\begin{bmatrix} \hat{H} + \delta_k I & 0 & J^\top \\ 0 & (\rho_k + \delta_k) I & I \\ J & I & 0 \end{bmatrix}. \tag{K_2}$$

The regularization parameter δ_k is increased until the desired inertia is achieved, guaranteeing that a valid search direction is found for the filter line-search algorithm. To simplify notation, we incorporate the inertia regularization into the penalty parameter by defining

$$\hat{\rho}_k := \rho_k + \delta_k.$$

Sparse $\mathrm{LBL}^{\mathrm{T}}$ factorization. If a sparse $\mathrm{LBL}^{\mathrm{T}}$ factorization is computed, the inertia of (K_2) can be obtained from the block diagonal matrix B (with 1x1 or 2x2 diagonal blocks). Previous implementations of NCL-based ALM [34, 35] have utilized the Newton KKT system (K_2) to compute $(\Delta x, \Delta r, -\Delta y)$ during the Newton iterations. Direct sparse linear solvers such as Duff and Reid's MA27 [20] and MA57 [19] are frequently employed, all utilizing numerical pivoting to ensure numerical stability and to handle ill-conditioned indefinite linear systems. However, due to the challenges of implementing numerical pivoting on parallel architectures, these traditional direct linear solvers are difficult to port efficiently to GPUs.

Sparse $\mathrm{LDL}^{\mathrm{T}}$ factorization. $\mathrm{LDL}^{\mathrm{T}}$ factorization (with D diagonal) is more economical than $\mathrm{LBL}^{\mathrm{T}}$ factorization, but it is not guaranteed to exist for every symmetric indefinite matrix. We observe that under a suitable symmetric permutation, (K_2) may admit an $\mathrm{LDL}^{\mathrm{T}}$ factorization. In particular, with the permutation matrix

$$P = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & I \\ I & 0 & 0 \end{bmatrix},$$

we can transform (K_2) into block-diagonal form:

$$P^\top \begin{bmatrix} \hat{H} + \delta_k I & 0 & J^\top \\ 0 & \hat{\rho}_k I & I \\ J & I & 0 \end{bmatrix} P = \begin{bmatrix} I & 0 & 0 \\ \theta_k I & I & 0 \\ 0 & -\hat{\rho}_k J^\top & I \end{bmatrix} \begin{bmatrix} \hat{\rho}_k I & 0 & 0 \\ 0 & -\theta_k I & 0 \\ 0 & 0 & \hat{H} + \delta_k I + \hat{\rho}_k J^\top J \end{bmatrix} \begin{bmatrix} I & \theta_k I & 0 \\ 0 & I & -\hat{\rho}_k J \\ 0 & 0 & I \end{bmatrix}, \quad (11)$$

with $\theta_k := \hat{\rho}_k^{-1}$. This shows that an $\mathrm{LDL}^{\mathrm{T}}$ decomposition is always possible provided the matrix $\hat{H} + \delta_k I + \rho_k J^\top J$ admits itself an $\mathrm{LDL}^{\mathrm{T}}$ factorization. For example, the regularization δ_k can be sufficiently large. The positive definiteness of $\hat{H} + \delta_k I + \rho_k J^\top J$ plays a crucial role in the inertia analysis in Section 3.4. However, we emphasize $\mathrm{LDL}^{\mathrm{T}}$ decomposition is not guaranteed to exist for every permutation P because (K_2) is not strongly factorizable, implying that zero pivots can be encountered during the factorization.

When the $\mathrm{LDL}^{\mathrm{T}}$ factorization routine encounters a zero pivot, two cases can occur: (i) The factorization stops, returning an incorrect inertia to the nonlinear solver. In this case, the inertia regularization increases δ_k until the $\mathrm{LDL}^{\mathrm{T}}$ factorization succeeds. (ii) The factorization replaces a null pivot with a value $\pm\varepsilon$ close to machine precision and proceeds to complete the factorization. The sign of ε depends on which block of the matrix (before reordering) the pivot belongs to [1]. Some linear solvers, including cuDSS, do not offer the option to set the sign of ε, reducing control during factorization. If the linear solver returns the correct inertia, the nonlinear solver does not perform any inertia correction. However, the factorization using the pivot perturbation strategy decomposes a perturbation of the original KKT system. The descent direction can be recovered afterwards if iterative refinement is used, with a greater number of refinements needed as the optimal solution is reached [42, 43].

In the following sections, we present two alternative reformulations of KKT system (K_2) that preserve desirable definiteness and inertia properties.

3.2 Stabilized KKT system (K_{2r})

We condense the Newton system (K_2) to a smaller stabilized KKT system, easier to solve. Substituting

$$\Delta r = \theta_k(\Delta y - y_k + y) - r \quad (12)$$

into K_2 gives

$$\begin{bmatrix} \hat{H} + \delta_k I & J^\top \\ J & -\theta_k I \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = - \begin{bmatrix} \nabla\phi(x) - J^\top y - (X-L)^{-1}\mu_k e + (U-X)^{-1}\mu_k e \\ c - \theta_k(y_k - y) \end{bmatrix}. \quad (K_{2r})$$

In contrast to the original system (K_2), the lower right block in (K_{2r}) is negative definite. If problem NLP is strictly convex, the Hessian in the $(1,1)$ block of (K_{2r}) is SPD, and system (K_{2r}) is symmetric quasi definite (SQD) without regularization ($\delta_k = 0$). It is known that SQD matrices are strongly factorizable. Thus in the convex case there exists an $\mathrm{LDL}^{\mathrm{T}}$ decomposition for every symmetric permutation [47].

If the problem is nonconvex, system (K_{2r}) is no longer SQD for $\delta_k = 0$, and the matrix is not strongly factorizable. Nevertheless, one can show that if a proper permutation is used, the matrix in (K_{2r}) still admits a block $\mathrm{LDL}^{\mathrm{T}}$ factorization. For example, like its sibling (K_2), system (K_{2r}) can be decomposed after a symmetric permutation using a backward identity matrix as

$$\begin{aligned} \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} \begin{bmatrix} \hat{H} + \delta_k I & J^\top \\ J & -\theta_k I \end{bmatrix} \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} &= \begin{bmatrix} I & 0 \\ -\hat{\rho}_k J^\top & I \end{bmatrix} \begin{bmatrix} -\theta_k I & 0 \\ 0 & \hat{H} + \delta_k I + \hat{\rho}_k J^\top J \end{bmatrix} \begin{bmatrix} I & -\hat{\rho}_k J \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ 0 & L_{22}^\top \end{bmatrix}, \end{aligned} \quad (13)$$

with $L_{11} = I$, $D_{11} = -\theta_k$, $L_{21} = -\hat{\rho}_k J^\top$, and $L_{22} D_{22} L_{22}^\top = \hat{H} + \delta_k I + \hat{\rho}_k J^\top J := M$. Thus, the matrix in (K_{2r}) admits a block $\mathrm{LDL}^{\mathrm{T}}$ factorization if the Schur complement M is strongly factorizable (e.g., positive definite). In §3.4 we show that for δ_k and ρ_k large enough, the matrix M is positive definite.

3.3 Condensed KKT system (K_{1s})

The stabilized system (K_{2r}) can be reduced further by removing the blocks associated with the dual descent direction Δy. We obtain a smaller condensed KKT system (K_{1s}) suitable for a sparse Cholesky factorization, as we show in Section 3.4.

System (K_{2r}) is equivalent to the *condensed KKT system*

$$(\hat{H} + \delta_k I + \hat{\rho}_k J^\top J)\Delta x = J^\top(y_k - \hat{\rho}_k c) - (\nabla\phi(x) - (X-L)^{-1}\mu_k e + (U-X)^{-1}\mu_k e), \tag{K_1}$$

$$\Delta y = -\hat{\rho}_k(J\Delta x + c) + y_k - y\ . \tag{14}$$

If NLP represents a problem NCO with inequality constraints, we can reduce further the size of the condensed system by eliminating the slack update Δs stored implicitly within Δx. We prefer to use (K_{1s}) below instead of (K_1) whenever we have inequality constraints, as the resulting system is smaller. We introduce the function $h(t) = \begin{bmatrix} c_{\mathcal{E}}(t)^\top & c_{\mathcal{I}}(t)^\top \end{bmatrix}^\top$ together with the matrix $M^\top = \begin{bmatrix} 0 & -I \end{bmatrix}$ and the two diagonal matrices

$$\Sigma_t = (T - L_t)^{-1} Z_{l,t} + (U_t - T)^{-1} Z_{u,t} \quad \text{and} \quad \Sigma_s = (S - L_s)^{-1} Z_{l,s} + (U_s - S)^{-1} Z_{u,s}. \tag{15}$$

The final condensation is detailed in the next proposition.

Proposition 1 (Condensed KKT system). *If* (1) *has structure NCO, then* (K_1) *is equivalent to*

$$\left(W_t + \Sigma_t + \hat{\rho}_k J_{c_{\mathcal{E}}}^\top J_{c_{\mathcal{E}}} + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top \Omega_k J_{c_{\mathcal{I}}}\right)\Delta t = r_t + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top r_s\ , \tag{K_{1s}}$$

with

$$W_t := \nabla^2 f(t) - \sum_{i=1}^{m} y_i \nabla^2 h_i(t)\ , \tag{16}$$

$$K_t := W_t + \hat{\rho}_k J_{c_{\mathcal{E}}}^\top J_{c_{\mathcal{E}}} + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top J_{c_{\mathcal{I}}} + \Sigma_t, \tag{17}$$

$$\Omega_k := \Sigma_s(\hat{\rho}_k + \Sigma_s)^{-1}, \tag{18}$$

$$r_t := J_h^\top(y_k - \hat{\rho}_k c) - \nabla f(t) + (T - L_t)^{-1}\mu_k e - (U_t - T)^{-1}\mu_k e, \tag{19}$$

$$r_s := M^\top(y_k - \hat{\rho}_k c) + (S - L_s)^{-1}\mu_k e - (U_s - S)^{-1}\mu_k e. \tag{20}$$

Proof. As (1) has the NCO structure, the optimization variable can be decomposed as $x = (t, s)$, with s a slack variable. The condensed KKT system (K_1) becomes

$$\begin{bmatrix} K_t & -\hat{\rho}_k J_{c_{\mathcal{I}}}^\top \\ -\hat{\rho}_k J_{c_{\mathcal{I}}} & \Sigma_s + \hat{\rho}_k I \end{bmatrix} \begin{bmatrix} \Delta t \\ \Delta s \end{bmatrix} = \begin{bmatrix} r_t \\ r_s \end{bmatrix}. \tag{K_1'}$$

We can eliminate $\Delta s = (\Sigma_s + \hat{\rho}_k I)^{-1}(\hat{\rho}_k J_{c_{\mathcal{I}}}\Delta t + r_s)$ in (K_1') to obtain the condensed system

$$\left[K_t - \hat{\rho}_k^2 J_{c_{\mathcal{I}}}^\top (\Sigma_s + \hat{\rho}_k I)^{-1} J_{c_{\mathcal{I}}}\right]\Delta t = r_t + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top r_s. \tag{21}$$

The left-hand-side in (21) expands as

$$\begin{aligned} K_t - \hat{\rho}_k^2 J_{c_{\mathcal{I}}}^\top (\Sigma_s + \hat{\rho}_k I)^{-1} J_{c_{\mathcal{I}}} &= W_t + \Sigma_t + \hat{\rho}_k J_{c_{\mathcal{E}}}^\top J_{c_{\mathcal{E}}} + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top (I - \hat{\rho}_k(\hat{\rho}_k + \Sigma_s)^{-1}) J_{c_{\mathcal{I}}}, \\ &= W_t + \Sigma_t + \hat{\rho}_k J_{c_{\mathcal{E}}}^\top J_{c_{\mathcal{E}}} + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top \Sigma_s(\hat{\rho}_k + \Sigma_s)^{-1} J_{c_{\mathcal{I}}}, \\ &= W_t + \Sigma_t + \hat{\rho}_k J_{c_{\mathcal{E}}}^\top J_{c_{\mathcal{E}}} + \hat{\rho}_k J_{c_{\mathcal{I}}}^\top \Omega_k J_{c_{\mathcal{I}}}, \end{aligned}$$

where we used the identity $I - \hat{\rho}_k(\hat{\rho}_k I + \Sigma_s)^{-1} = \big((\hat{\rho}_k I + \Sigma_s) - \hat{\rho}_k I\big)(\hat{\rho}_k I + \Sigma_s)^{-1} = \Sigma_s(\hat{\rho}_k I + \Sigma_s)^{-1}$. By replacing this expression in (21), we obtain (K_{1s}). □

On one hand, if an inequality constraint i is active at a solution, then $(\Sigma_s)_{ii} \to +\infty$ as the iterates converge. Hence we have $(\Omega_k)_{ii} \to 1$: the constraint is treated asymptotically as an equality constraint by NCL. On the other hand, if the inequality constraint is inactive, then $(\Sigma_s)_{ii} \to 0$, leading to $(\Omega_k)_{ii} \to 0$. The block associated with the inactive constraints becomes negligible in (K_{1s}), as expected.

3.4 Analysis of the inertia of reformulated KKT systems

Proposition 2 shows how the inertia of (K_2) relates to that of (K_{2r}) and (K_{1s}).

Proposition 2. *If $\rho_k, \delta_k > 0$, the following statements are equivalent:*

(i) $\mathrm{In}(K_2) = (n+m,\, m,\, 0)$;
(ii) $\mathrm{In}(K_{2r}) = (n,\, m,\, 0)$;
(iii) $\mathrm{In}(K_{1s}) = (n,\, 0,\, 0)$.

Proof. It is sufficient to show the equivalence of (i) and (ii), as well as that of (ii) and (iii).

We first show (i) $\iff$ (ii). By Sylvester's law of inertia applied to K_2, we have

$$\begin{aligned}\mathrm{In}(K_2) = \mathrm{In}\begin{bmatrix}\hat{H}+\delta_k I & 0 & J^\top \\ 0 & \hat{\rho}_k I & I \\ J & I & 0\end{bmatrix} &= \mathrm{In}(\hat{\rho}_k I) + \mathrm{In}\begin{bmatrix}\hat{H}+\delta_k I & J^\top \\ J & -\theta_k I\end{bmatrix} \\ &= (m,0,0) + \mathrm{In}(K_{2r}),\end{aligned}$$

where the third equality follows from the fact that $\hat{\rho}_k > 0$. Therefore, (i) is equivalent to (ii).

Next, we show (ii) $\iff$ (iii). We apply Sylvester's law of inertia to the matrix in K_{2r} to obtain

$$\begin{aligned}\mathrm{In}(K_{2r}) = \mathrm{In}\begin{bmatrix}\hat{H}+\delta_k I & J^\top \\ J & -\theta_k I\end{bmatrix} &= \mathrm{In}(-\theta_k I) + \mathrm{In}(\hat{H}+\delta_k I + \hat{\rho}_k J^\top J) \\ &= (0,m,0) + \mathrm{In}(K_{1s}).\end{aligned}$$

Thus, $\mathrm{In}(K_{2r}) = (n,m,0)$ if and only if $\mathrm{In}(K_{1s}) = (n,0,0)$, which is equivalent to the matrix K_{1s} being SPD. □

Here, the assumption $\rho_k, \delta_k > 0$ is always satisfied if ρ_k and δ_k are selected by Algorithm 1.

Proposition 2 has important implications for inertia-based IPMs. It demonstrates that if either $\mathrm{In}(K_{2r}) = (n,\, m,\, 0)$ or $\hat{H} + \delta_k I + \hat{\rho}_k J^\top J \succ 0$, the solution of system (K_2) provides a search direction for NC_k. Thus, even if we base the solver on (K_{2r}) or (K_{1s}), their respective inertias determine if the regularization δ_k is sufficient to ensure that the Newton step is a descent direction. This guarantees that the solution proceeding with the (K_{2r}) or (K_{1s}) systems can achieve, in principle, the same degree of robustness in the convergence behavior as algorithms based on the original system (K_2).

4 Implementation

We discuss various aspects of our implementation of Algorithm NCL and the numerical benchmark studies.

4.1 MadNCL

We have implemented Algorithm 1 in the MadNCL solver, which is publicly available on GitHub.[1] We adapted the Julia implementation `NCL` [35] to utilize MadNLP to solve the NC_k subproblems. We leverage MadNLP's warm-start feature to reuse the data structure throughout the iterations, including the initial symbolic factorization. MadNCL employs the `AbstractKKTSystem` abstraction implemented within MadNLP to implement formulations (K_{2r}) and (K_{1s}) as `K2rAuglagKKTSystem` and `K1sAuglagKKTSystem` abstractions, respectively. By utilizing the MadNLP formalism, we can efficiently allocate all data structures and perform all computations on the GPU, minimizing the transfer of data between the host (CPU memory) and the device (GPU memory).

4.2 Scaling

The objective of subproblem NC_k combines the original objective $\phi(x)$ with the augmented Lagrangian term $y_k^\top r + \frac{\rho_k}{2}\|r\|^2$. It is therefore sensitive to problem scaling. For instance, if $\rho_k = 1000$, $y_k = 0$, and there are 10,000 constraints of magnitude $O(1)$ while the objective also has a magnitude of $O(1)$, we add an $O(1)$ term to a term of magnitude $O(10^7)$. This can lead to numerical instabilities.

To address this, we employ an automatic scaling strategy similar to that used in Ipopt [48]. Given the initial point x_0, $g_0 = \nabla f(x_0)$, and $g_i = \nabla c_i(x_0)$ for $i = 1, \ldots, m$, we scale the objective by σ_f and each constraint by $(\sigma_c)_i$, where

$$\sigma_f = \max\left\{10^{-8}, \min\left(1, \frac{\tau}{\|g_0\|_\infty}\right)\right\}, \quad (\sigma_c)_i = \max\left\{10^{-8}, \min\left(1, \frac{\tau}{\|g_i\|_\infty}\right)\right\}, \tag{22}$$

where $\tau > 0$. In our implementation we set $\tau = 1$, which differs from the parameter used in Ipopt ($\tau = 100$) but is the same as in ALGENCAN [13]. Despite being more aggressive, this scaling is more appropriate for the ALM.

4.3 Linear solver

On the CPU, MadNCL uses the linear solvers HSL MA27 and HSL MA57. On the GPU, we use the solver NVIDIA cuDSS for the LDL^T factorization in both (K_{2r}) and (K_{1s}).

On one hand, the condensed system (K_{1s}) is positive definite after regularization, meaning it is strongly factorizable. We use the default options in cuDSS for ordering, *pivot threshold* (used to determine if diagonal element is subject to pivoting and will be swapped with the maximum element in the row or column) and *pivot epsilon* (used to replace the small diagonal elements encountered during numerical factorization).

On the other hand, the stabilized system (K_{2r}) is not strongly factorizable (see §3.2). We use a pivot regularization strategy within cuDSS by setting the pivot epsilon to 10^{-10}. As a result, cuDSS returns the factorization of a slightly perturbed matrix. We recover the solution of the original system by using iterative refinement (Richardson iterations by default) if the inertia is correct. This strategy has been implemented before in Ipopt, when the KKT systems are solved with the linear solver Panua Pardiso [43]. We plan to investigate more sophisticated strategies for iterative refinement in the future [4].

4.4 Modeler and automatic differentiation

The benchmark instances are created using ExaModels [45], which allow for automatic differentiation on GPUs. In particular, ExaModels employs the SIMD abstraction to optimize the derivative evaluations over embarrassingly parallel objective and constraint function expressions. ExaModels is highly optimized, often rendering the derivative evaluation time a negligible portion of the overall solution time. Consequently, when tested with problem instances implemented using ExaModels, MadNCL's performance is primarily determined by linear algebra computations and solver internals.

[1] `https://github.com/MadNLP/MadNCL.jl`

5 Numerical results

We detail the performance of NCL (Algorithm 1) on various NLP instances. In §5.1 we present results on the CUTEst set [27] with MadNCL running on the CPU with HSL MA57. In §5.2 we analyze the performance of MadNCL running on the GPU with the linear solver NVIDIA cuDSS on large-scale OPF and COPS instances. Finally, §5.3 shows the performance of MadNCL on degenerate nonlinear programs arising from power systems.

All results on the CPU have been generated using an AMD EPYC 7443 (24-core) processor. The benchmarks are generated on the GPU using an NVIDIA H100. We use labels MadNCL-K2r and MadNCL-K1s to denote MadNCL computing the Newton descent direction with (K_{2r}) and (K_{1s}) respectively.

5.1 CUTEst benchmark

We start by analyzing the performance of MadNCL on problems from the CUTEst collection. We select all instances with more than 1,000 variables and at least 1 constraint. We compare MadNCL with Ipopt and MadNLP, which both use the $\mathrm{LBL}^{\mathrm{T}}$ factorization in HSL MA57, whereas MadNCL uses HSL MA57 with a pivot threshold set to 0.0 to deactivate numerical pivoting. Results with optimality tolerance `tol=1e-8` are displayed in Figure 1. We observe that MadNCL compares favorably with Ipopt and MadNLP. Despite not being the fastest solver, MadNCL is more robust than MadNLP and Ipopt. A closer look shows that MadNCL can solve to optimality instances where Ipopt and MadNLP are failing because (i) the problems do not have enough degrees of freedom (`CHAINWOONE`, `NINE5D`, `NINENEW`, `MODBEALENE`, `FIVE20B`, `FIVE20C`, `BDQRTICNE`, `HIER163A`, `HIE1327D`, `HIER133E`, `TWO5IN6`) (ii) the `IPM` algorithm never exits feasibility restoration (`BRAINPC1`, `BRAINPC5`, `BRAINPC9`, `SAROMM`, `ARTIF`), and (iii) the filter line-search IPM implemented in Ipopt and MadNLP performs too many primal-dual regularization (`MSS3`, `ORTHREGE`).

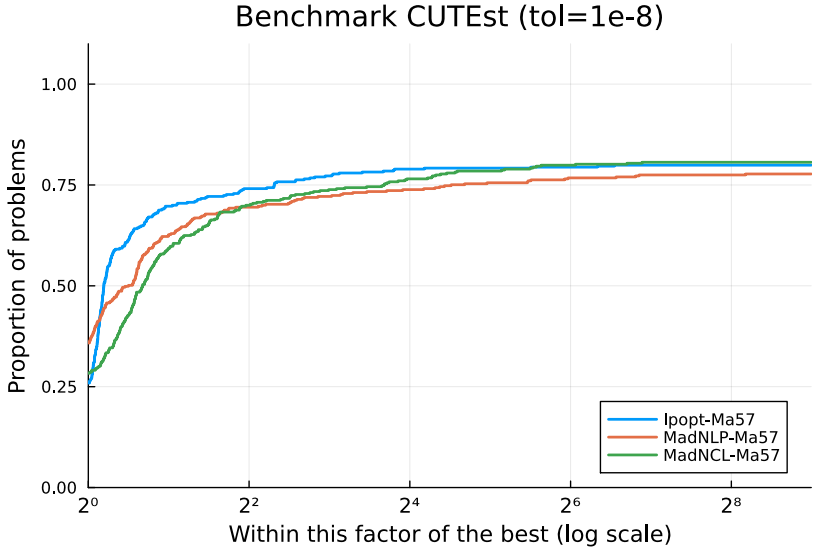

Figure 1: Results on the CUTEst benchmark (using `tol=1e-8`). We profile the time (in seconds) to achieve optimality. We have selected all instances with at least 1,000 variables and 1 constraint. MadNCL is running on the CPU with HSL MA57.

In terms of solving time, we observe that MadNCL is particularly effective at solving the nonconvex QP instances in CUTEst: on `NCVXQP7` and `NCVXQP8` it takes respectively 32s and 56s compared to 193s and 460s for Ipopt. Similar speed-ups are reported for other nonconvex QPs (`AOESSNDL`, `AONSSSSL`, `AOENSNDL`, `AONNSNSL`, `A2ESSNDL`). However, we observe that MadNCL can be significantly slower on

some specific instances where it converges in 10 times more IPM iterations than Ipopt or MadNLP (`BLOWEYA`, `YAO`, `CHEMRCTA`, `UBH5`, `SEMICON1`, `POROUS1`, `FERRISDC`, `READING4`), even though we warm-start on each subproblem NC_k except the first.

5.2 Achieving scalability on GPU

The CUTEst benchmark presented in §5.1 runs entirely on the CPU. In this subsection, we assess the performance achieved by MadNCL on the GPU. To do so, we move the models evaluation to the GPU using the modeler ExaModels. The linear systems (K_{2r}) and (K_{1s}) are factorized on the GPU using NVIDIA cuDSS. The resulting algorithm runs entirely on the GPU, from the evaluation of the derivatives to the computation of the Newton step. This avoids costly transfers between the host and the device memory. Our GPU-accelerated benchmark includes OPF instance from PGLIB [8] and large-scale nonlinear programs from COPS [18]. Unfortunately, CUTEst does not support evaluating the models on the GPU, in constrast to ExaModels.

5.2.1 Assessing MadNCL's performance on the GPU

We assess the speed-up obtained when MadNCL solves on the GPU the large optimal power flow (OPF) instance `78484epigrids`, compared to a CPU implementation. The instance has 674,562 variables and 1,039,062 constraints. We set the tolerance to `tol=1e-8`. On the CPU we use the linear solver HSL MA27 (which is faster than MA57 for OPF problems [46]), and on the GPU we use NVIDIA cuDSS.

Results from the largest instance `78484epigrids` show that the linear solver cuDSS is effective when using the formulation (K_{2r}): MadNCL converges in 330 IPM iterations in 54.4s, an 18x speed-up compared to MadNCL on the CPU with HSL MA27. For comparison, the reference method MadNLP (using HSL MA27) converges in 328 seconds to the same solution. The performance difference observed between CPU and GPU is largely explained by the faster solution time in the linear solver, with cuDSS being effective at factorizing (K_{2r}) on the GPU. If we report the time per iteration, cuDSS is 17 times faster than HSL MA27 at factorizing system (K_{2r}).

MadNCL fails to converge when using the K_{1s} formulation. The algorithm struggles to address the increased ill-conditioning in the condensed KKT system (K_{1s}) in this case. (We see in the next section that this is not always the case.) Overall, (K_{1s}) proves to be significantly less robust than the formulation (K_{2r}). A closer investigation has shown that (K_{1s}) is not able to find an accurate descent direction in the final IPM iterations (despite the iterative refinement we are using), impairing MadNCL's convergence. As ρ_k increases, the matrix (K_{1s}) becomes too ill-conditioned and the descent direction is not sufficiently accurate.

Table 1: Performance comparison of MadNCL on a large-scale OPF instance using different formulations for the KKT systems. As a baseline, we give the time spent in MadNLP on the CPU in the first row. The columns are (i) **flag**: return status for MadNLP (1: locally optimal, 0: failed to converge); (ii) **it**: total number of IPM iterations; (iii) **lin**: time spent in the linear solver (in seconds); (iv) **total**: time spent in the solver (in seconds).

			78484epigrids			
solver	KKT	linear solver	flag	it	lin	**total**
MadNLP	K_2	ma27	1	104	312.6	**353.9**
MadNCL	K_2	ma27	1	322	1053.6	**1182.0**
MadNCL	K_{2r}	ma27	1	322	847.9	**971.5**
MadNCL	K_{2r}	cudss-ldl	1	330	50.1	**54.4**
MadNCL	K_{1s}	ma27	0	1000	3222.2	**3848.6**
MadNCL	K_{1s}	cudss-ldl	0	1000	154.1	**170.2**

5.2.2 Comparing MadNCL with other GPU-accelerated solvers

In this section, we present our principal benchmark on large-scale OPF and COPS instances [8, 18]: we compare MadNCL running on the GPU with MadNLP (running on the CPU using HSL MA27,

and on the GPU using the two GPU-accelerated KKT linear solvers HyKKT and LiftedKKT [38]). The results are presented in Table 2. We observe that MadNCL exhibits better performance on the COPS instances than on the OPF ones.

The OPF instances have very sparse Jacobians. We observe that MadNCL-K2r is able to solve all cases except `10480_goc`. This is much better than MadNCL-K1s, which fails on all instances. MadNCL-K2r requires more IPM iterations than MadNLP, as expected for an Augmented Lagrangian method (superlinear convergence is achieved only when we enter the extrapolation step). However, the GPU-acceleration benefits MadNCL-K2r, which is overall faster than MadNLP running on the CPU with HSL MA27: MadNCL-K2r achieves a 6x speed-up on the largest instance `78484_epigrids`. On the other hand, MadNCL-K2r remains slower than MadNLP running on the GPU, with LiftedKKT being the fastest method here.

Results on the COPS instances are different from the OPF instances. In contrast to what we observed earlier, both MadNCL-K2r and MadNCL-K1s exhibit reliable convergence and are significantly faster than MadNLP on the CPU with HSL MA27. MadNCL-K1s is here the fastest NCL variant, and compares favorably w.r.t. MadNLP running on the GPU with HyKKT and LiftedKKT. In contrast to the OPF instances, the number of IPM iterations in MadNCL does not increase significantly compared to MadNLP. The failure observed on the `rocket` instance is noteworthy, as this problem is known to exhibit a specific type of degeneracy: the reduced Hessian is nearly singular at optimality. In that situation, we have no guarantee that the Augmented Lagrangian method used in MadNCL would converge [22]. This clarifies that the issue arises from a different form of degeneracy than that handled by NCL.

Table 2: Comparing the performance of MadNLP and MadNCL on the GPU for large-scale OPF instances from PGLIB [8] and large-scale COPS instances [18]. The tolerance is `tol=1e-8`. The columns are: (i) **flag**: return status for MadNLP (1: locally optimal, 2: solved to acceptable level, 0: failed to converge); (ii) **it**: total number of IPM iterations; (iii) **lin**: time spent in the linear solver (in seconds); (iv) **total**: time spent in the solver (in seconds).

	MadNLP-K2-Ma27				MadNLP-HyKKT-cuDSS				MadNLP-LiftedKKT-cuDSS				MadNCL-K2r-cuDSS				MadNCL-K1s-cuDSS			
case	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**
9241_pegase	1	75	7.04	**9.77**	1	72	0.88	**1.75**	0	1000	11.85	**22.50**	1	301	6.25	**8.29**	0	835	43.81	**51.91**
9591_goc	1	68	12.92	**15.52**	1	194	1.75	**3.88**	1	85	1.50	**2.33**	1	89	1.73	**2.74**	0	1000	47.00	**57.76**
10000_goc	1	85	7.37	**10.20**	1	85	0.58	**1.35**	1	65	0.73	**1.33**	1	108	1.65	**2.43**	0	1000	27.36	**35.36**
10192_epigrids	1	56	9.88	**12.78**	2	69	0.74	**2.15**	1	58	1.28	**2.15**	1	143	2.37	**3.37**	0	1000	71.93	**81.12**
10480_goc	1	72	14.21	**17.52**	1	340	4.42	**7.92**	1	68	0.91	**1.75**	0	1000	40.63	**48.21**	0	1000	62.12	**71.12**
13659_pegase	1	65	12.48	**16.44**	1	64	0.70	**1.75**	1	72	0.85	**1.89**	1	254	6.18	**8.20**	0	1000	35.31	**44.17**
19402_goc	1	72	54.93	**62.31**	1	541	9.26	**14.71**	1	72	1.21	**3.12**	1	163	5.97	**7.17**	0	657	55.65	**63.24**
20758_epigrids	1	53	27.81	**33.31**	1	55	0.88	**2.78**	0	1000	26.26	**34.82**	1	284	12.85	**15.04**	0	1000	93.64	**104.63**
24464_goc	1	64	40.26	**48.12**	2	609	13.71	**20.39**	1	65	1.05	**3.03**	1	554	32.80	**36.87**	0	945	58.38	**68.17**
30000_goc	1	229	159.86	**182.19**	1	231	3.76	**5.92**	1	153	2.36	**3.91**	1	343	19.34	**22.06**	0	1000	61.73	**72.21**
78484_epigrids	1	104	312.58	**353.85**	0	1000	42.39	**55.47**	1	105	4.58	**10.10**	1	330	50.10	**54.39**	0	1000	154.14	**170.23**
bearing	1	18	13.35	**22.61**	1	18	0.76	**3.55**	1	15	0.12	**2.86**	1	11	0.18	**3.36**	1	11	0.16	**1.41**
catmix	1	20	1.30	**7.44**	1	18	0.05	**2.39**	1	26	0.18	**2.51**	1	114	0.70	**3.75**	1	39	0.19	**1.58**
channel	1	6	1.09	**7.29**	1	6	0.03	**2.74**	1	7	0.03	**2.61**	1	24	0.28	**1.90**	1	21	0.14	**1.36**
elec	1	209	98.03	**125.83**	1	103	0.69	**4.26**	1	237	1.71	**7.02**	1	199	3.33	**8.01**	1	141	1.00	**3.07**
gasoil	1	20	1.47	**16.90**	1	21	0.13	**3.19**	1	43	0.86	**4.93**	1	18	0.28	**2.81**	1	18	0.14	**2.11**
marine	1	14	2.19	**9.45**	1	11	0.07	**2.81**	1	25	0.35	**3.14**	1	22	0.23	**1.97**	1	22	0.15	**1.59**
pinene	1	12	2.40	**10.60**	1	13	0.15	**1.13**	1	19	0.27	**1.38**	1	51	0.74	**0.98**	1	51	0.41	**1.53**
polygon	1	32	0.02	**0.03**	1	31	0.06	**0.20**	1	189	0.30	**1.03**	1	67	0.17	**0.53**	1	67	0.25	**0.58**
robot	1	31	1.52	**11.81**	1	87	162.30	**167.62**	1	26	0.06	**3.35**	1	53	0.30	**6.49**	1	67	0.36	**3.68**
rocket	1	75	1.11	**10.56**	1	183	21.27	**27.10**	1	111	0.32	**4.46**	0	140	0.53	**4.93**	1	183	0.42	**3.66**
steering	1	17	0.18	**7.79**	1	16	0.04	**2.56**	1	14	0.06	**2.43**	1	15	0.07	**3.11**	1	15	0.06	**1.69**
torsion	1	14	0.74	**0.96**	1	14	0.08	**0.29**	1	15	0.05	**0.25**	1	12	0.09	**0.16**	1	12	0.10	**0.17**

5.3 NLPs with degenerate constraints

The previous benchmark in §5.2 focused on solving regular nonlinear programs. Now we assess how MadNCL performs on degenerate NLPs. We consider practical degenerate instances arising in power system applications: the security-constrained optimal power flow (SCOPF), here formulated as mathematical programs with complementarity constraints (MPCC). The detailed formulation can be found in [3]. Here, the problem associates each contingency scenario with a given line failure. The total number of contingencies is denoted by n_K: the number of complementarity constraints increases linearly w.r.t. n_K, rendering the problem more difficult to solve for large n_K. By nature, MPCCs are degenerate: the Mangasarian-Fromovitz constraint qualification (MFCQ) is violated at every feasible point. Despite this degeneracy, it has been proven that the Augmented Lagrangian method can converge

(globally) to a strongly stationary solution, provided a regularity condition holds at the solution [29]. The SCOPF instances here are large-scale and degenerate, offering a good use case to test MadNCL's performance. As these instances are harder to solve, we relax the tolerance to `tol=1e-5`.

Classical nonlinear IPM solvers like Ipopt and MadNLP fail to solve the SCOPF instances to optimality: both solvers suffer from the lack of problem regularity and they converge to an infeasible solution after entering feasibility restoration. In contrast, MadNCL converges reliably despite the ill-conditioning of linear systems (K_{2r}) and (K_{1s}). The results are displayed in Table 3. MadNCL-K2r is able to solve all instances to `tol=1e-5` on the CPU (using HSL MA57) and on the GPU (using cuDSS). As in §5.2.1, we observe that MadNCL-K2r is more robust than MadNCL-K1s, which suffers from the increased ill-conditioning of system (K_{1s}). In addition, MadNCL-K2r is significantly faster when using GPU-acceleration, with up to a 10x speed-up on the largest instances.

Table 3: Comparing the performance of MadNCL on large-scale SCOPF instances. The tolerance is `tol=1e-5`. Column n_K shows the number of contingencies used in the SCOPF. Columns n and m show the number of variables and constraints. The other columns are (i) **flag**: return status for MadNLP (1: locally optimal, 2: solved to acceptable level, 0: failed to converge); (ii) **it**: total number of IPM iterations; (iii) **lin**: time spent in the linear solver (in seconds); (iv) **total**: time spent in the solver (in seconds).

				MadNCL-K2r-Ma57				MadNCL-K1s-MA57				MadNCL-K2r-cuDSS				MadNCL-K1s-cuDSS			
case	n_K	n	m	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**	**flag**	**it**	**lin**	**total**
118	100	131588	168853	1	25	3.25	**7.90**	1	25	2.79	**8.90**	1	25	0.62	**0.89**	1	25	1.54	**5.08**
300	100	268282	350967	1	49	13.23	**19.44**	1	49	9.94	**16.15**	1	49	1.95	**2.64**	1	49	1.29	**1.75**
ACTIVSg200	100	162356	211571	1	31	8.63	**14.17**	1	31	6.11	**11.20**	1	33	1.19	**3.63**	1	33	0.71	**2.43**
1354pegase	8	109056	144327	1	42	13.38	**16.44**	1	42	11.51	**16.43**	1	45	1.41	**3.62**	0	250	12.94	**17.08**
1354pegase	16	206920	273999	1	42	31.83	**36.76**	1	42	31.82	**37.20**	1	46	2.90	**3.29**	1	42	1.90	**2.28**
1354pegase	32	402648	533343	1	165	202.25	**234.95**	0	250	447.04	**499.25**	1	218	54.42	**56.74**	1	248	35.51	**39.86**
2869pegase	8	242102	323479	1	45	25.60	**31.32**	0	250	182.80	**212.34**	1	45	3.80	**4.22**	0	250	25.95	**28.82**
2869pegase	16	459118	613727	1	48	63.34	**75.21**	1	56	85.30	**98.42**	1	50	8.32	**8.90**	0	250	52.15	**55.71**

6 Conclusions and future work

We have explored a GPU implementation of Algorithm NCL for solving large-scale NLP problems, especially ones whose constraints fail LICQ at a solution. NCL's need for an IPM subproblem solver is provided by MadNLP. We focused on some limitations of existing GPU-accelerated NLP solvers. We demonstrated that problem structure within MadNLP can be leveraged at the linear algebra level by introducing the stabilized KKT system K_{2r} and the condensed KKT system K_{1s}, which facilitate efficient linear algebra computation on GPUs by utilizing MadNLP's flexible abstraction of KKT systems. From a computational standpoint, we have established the first GPU-friendly augmented system that avoids explicit formation of a Schur complement, offering significant advantages for large problems while remaining general enough to address a broad class of problems. Extensive numerical experiments have validated the performance of MadNCL on GPUs, showcasing it as a reliable solver for optimization problems regardless of their degeneracy.

MadNCL holds great potential for future research in addressing challenges such as failure of LICQ at a solution, absence of strict complementarity, or the presence of complementarity constraints (MPCC) as in the SCOPF examples studied in §5.3. Incorporating outer/inner loop structures inspired by NCL into existing frameworks like Superb [26] could broaden the applicability of this approach.

References

[1] Erling D. Andersen, Jacek Gondzio, Csaba Mészáros, and Xiaojie Xu. Implementation of Interior-Point Methods for Large Scale Linear Programs, pages 189–252. Springer US, Boston, MA, 1996.

[2] Roberto Andreani, Ernesto G Birgin, José Mario Martínez, and María Laura Schuverdt. On augmented Lagrangian methods with general lower-level constraints. SIAM J. Optim., 18(4):1286–1309, 2008.

[3] Ignacio Aravena, Daniel K Molzahn, Shixuan Zhang, Cosmin G Petra, Frank E Curtis, Shenyinying Tu, Andreas Wächter, Ermin Wei, Elizabeth Wong, Amin Gholami, et al. Recent developments in security-

constrained AC optimal power flow: Overview of challenge 1 in the ARPA-E grid optimization competition. Operations research, 71(6):1997–2014, 2023.

[4] Mario Arioli, Iain S Duff, Serge Gratton, and Stéphane Pralet. A note on GMRES preconditioned by a perturbed LDLT decomposition with static pivoting. SIAM J. Sci. Comput., 29(5):2024–2044, 2007.

[5] Paul Armand, Joël Benoist, and Dominique Orban. From global to local convergence of interior methods for nonlinear optimization. Optimization Methods and Software, 28(5):1051–1080, 2013.

[6] Paul Armand and Riadh Omheni. A mixed logarithmic barrier-augmented Lagrangian method for nonlinear optimization. J. Optim. Theory and Applics., 173(2):523–547, 2017.

[7] Sylvain Arreckx and Dominique Orban. A regularized factorization-free method for equality-constrained optimization. SIAM J. Optim., 28(2):1613–1639, 2018.

[8] Sogol Babaeinejadsarookolaee, Adam Birchfield, Richard D Christie, Carleton Coffrin, Christopher DeMarco, Ruisheng Diao, Michael Ferris, Stephane Fliscounakis, Scott Greene, Renke Huang, et al. The power grid library for benchmarking AC optimal power flow algorithms. arXiv preprint arXiv:1908.02788, 2019.

[9] Michele Benzi, Gene H Golub, and Jörg Liesen. Numerical solution of saddle point problems. Acta Numerica, 14:1–137, 2005.

[10] Dimitri P Bertsekas. Multiplier methods: A survey. Automatica, 12(2):133–145, 1976.

[11] Dimitri P Bertsekas. Constrained Optimization and Lagrange Multiplier Methods. Academic Press, 2014.

[12] Ernesto G Birgin and José Mario Martínez. Practical Augmented Lagrangian Methods for Constrained Optimization. SIAM, 2014.

[13] Ernesto G Birgin and José Mario Martínez. Complexity and performance of an augmented Lagrangian algorithm. Optim. Methods and Softw., 35(5):885–920, 2020.

[14] Alice Chiche and Jean Charles Gilbert. How the augmented Lagrangian algorithm can deal with an infeasible convex quadratic optimization problem. Journal of Convex Analysis, 3(4):5, 2014.

[15] Andrew R Conn, Nicholas I M Gould, and Philippe L Toint. A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds. SIAM J. Numer. Anal., 28:545–572, 1991.

[16] Andrew R Conn, Nicholas I M Gould, and Philippe L Toint. LANCELOT: A Fortran Package for Large-scale Nonlinear Optimization (Release A). Lecture Notes in Computational Mathematics 17. Springer Verlag, 1992.

[17] Frank E Curtis, Hao Jiang, and Daniel P Robinson. An adaptive augmented Lagrangian method for large-scale constrained optimization. Math. Program., 152(1):201–245, 2015.

[18] Elizabeth D Dolan, Jorge J Moré, and Todd S Munson. Benchmarking optimization software with COPS 3.0. Technical report, Argonne National Lab., Argonne, IL (US), 2004.

[19] Ian S Duff. MA57: a Fortran code for the solution of sparse symmetric definite and indefinite systems. ACM Trans. Math. Softw., 30(2):118–144, 2004.

[20] Iain S Duff and John K Reid. The multifrontal solution of indefinite sparse symmetric linear equations. ACM Trans. Math. Softw., 9(3):302–325, 1983.

[21] Jean-Pierre Dussault. Numerical stability and efficiency of penalty algorithms. SIAM J. Numer. Anal., 32(1):296–317, 1995.

[22] Damián Fernández and Mikhail V Solodov. Local convergence of exact and inexact augmented Lagrangian methods under the second-order sufficient optimality condition. SIAM J. Optim., 22(2):384–407, 2012.

[23] Alexandre Ghannad, Dominique Orban, and Michael A. Saunders. Linear systems arising in interior methods for convex optimization: a symmetric formulation with bounded condition number. Optim. Methods and Softw., 37(4):1344–1369, 2022.

[24] Philip E Gill and Daniel P Robinson. A primal-dual augmented Lagrangian. Comput. Optim. Appl., 51(1):1–25, 2012.

[25] Nicholas I M Gould. On practical conditions for the existence and uniqueness of solutions to the general equality quadratic programming problem. Math. Program., 32(1):90–99, 1985.

[26] Nicholas I M Gould, Dominique Orban, and Philippe L Toint. An Interior-Point ℓ1-Penalty Method for Nonlinear Optimization. Springer, 2003.

[27] Nicholas I M Gould, Dominique Orban, and Philippe L Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. Comput. Optim. Appl., 60(3):545–557, 2015.

[28] Magnus R Hestenes. Multiplier and gradient methods. J. Optim. Theory and Applics., 4(5):303–320, 1969.

[29] Alexey F Izmailov, Mikhail V Solodov, and EI Uskov. Global convergence of augmented Lagrangian methods applied to optimization problems with degenerate constraints, including problems with complementarity constraints. SIAM J. Optim., 22(4):1579–1606, 2012.

[30] Alexey F Izmailov, Mikhail V Solodov, and EI Uskov. Combining stabilized SQP with the augmented Lagrangian algorithm. Comput. Optim. Appl., 62:405–429, 2015.

[31] Renke Kuhlmann and Christof Büskens. A primal–dual augmented Lagrangian penalty-interior-point filter line search algorithm. Mathematical Methods of Operations Research, 87(3):451–483, 2018.

[32] Haihao Lu, Zedong Peng, and Jinwen Yang. cuPDLPx: A Further Enhanced GPU-Based First-Order Solver for Linear Programming, 2025.

[33] Haihao Lu and Jinwen Yang. cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. arXiv preprint arXiv:2311.12180, 2023.

[34] Ding Ma, Kenneth L Judd, Dominique Orban, and Michael A Saunders. Stabilized optimization via an NCL algorithm. In M. Al-Baali, Lucio Grandinetti, and Anton Purnama, editors, *Numerical Analysis and Optimization, NAO-IV, Muscat, Oman, January 2017*, volume 235 of Springer Proceedings in Mathematics and Statistics, pages 173–191. Springer International Publishing Switzerland, 2018.

[35] Ding Ma, Dominique Orban, and Michael A Saunders. A Julia implementation of Algorithm NCL for constrained optimization. In M. Al-Baali, A. Purnama, and L. Grandinetti, editors, *Numerical Analysis and Optimization, NAO-V, Muscat, Oman, January 2020*, volume 354 of Springer Proceedings in Mathematics and Statistics, pages 133–182. Springer, 2021.

[36] Bruce A Murtagh and Michael A Saunders. MINOS nonlinear optimization solver. `https://www.gams.com/latest/docs/solvers/minos/`, accessed May 2017.

[37] Jorge Nocedal and Stephen J Wright. Numerical Optimization. Springer series in Operations Research. Springer, New York, 2nd edition, 2006.

[38] François Pacaud, Sungho Shin, Alexis Montoison, Michel Schanen, and Mihai Anitescu. Condensed-space methods for nonlinear programming on GPUs. arXiv preprint arXiv:2405.14236, 2024.

[39] François Pacaud, Sungho Shin, Michel Schanen, Daniel Adrian Maldonado, and Mihai Anitescu. Accelerating condensed interior-point methods on SIMD/GPU architectures. J. Optim. Theory and Applics., 202(1):184–203, 2024.

[40] Shaked Regev, Nai-Yuan Chiang, Eric Darve, Cosmin G Petra, Michael A Saunders, Kasia Świrydowicz, and Slaven Peleš. HyKKT: a hybrid direct-iterative method for solving KKT linear systems. Optim. Methods and Softw., 38(2):332–355, 2023.

[41] Ralph Tyrrell Rockafellar. Augmented Lagrange multiplier functions and duality in nonconvex programming. SIAM J. Control Optim., 12(2):268–285, 1974.

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

[43] Olaf Schenk, Andreas Wächter, and Michael Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. Comput. Optim. Appl., 36:321–341, 2007.

[44] Michel Schubiger, Goran Banjac, and John Lygeros. GPU acceleration of ADMM for large-scale quadratic programming. Journal of Parallel and Distributed Computing, 144:55–67, 2020.

[45] Sungho Shin, Mihai Anitescu, and François Pacaud. Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. Electric Power Systems Research, 236:110651, 2024.

[46] Byron Tasseff, Carleton Coffrin, Andreas Wächter, and Carl Laird. Exploring benefits of linear solver parallelism on modern nonlinear optimization applications. arXiv preprint arXiv:1909.08104, 2019.

[47] Robert J. Vanderbei. Symmetric quasi-definite matrices. SIAM J. Optim., 5:100–113, 1995.

[48] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program., 106(1):25–57, 2006.