Disciplined Nonlinear Programming

Daniel Cederberg William Zhang Parth Nobel
Stephen Boyd

December 8, 2025

Abstract

We introduce *disciplined nonlinear programming* (DNLP), a syntax for specifying nonlinear programming problems. DNLP is inspired by disciplined convex programming (DCP) and allows smooth functions to be freely mixed with nonsmooth convex and concave functions, with rules governing how the nonsmooth functions can be used. Problems expressed in DNLP form can be automatically canonicalized to a standard nonlinear programming (NLP) form and passed to a suitable NLP solver. As in DCP, the canonicalization relaxes nonsmooth convex and concave functions in a lossless way, allowing them to be handled by NLP solvers that require smooth functions. In addition to extending NLP to include useful nondifferentiable convex and concave functions, transforming the original problem to an equivalent NLP form offers several advantages, including simpler problem initialization. We describe the language and our open-source implementation of DNLP as an extension of CVXPY, a parser for DCP.

Contents

1 Introduction

Nonlinear programming (NLP) has a long and well-established history [39], with successful applications spanning decades in fields such as chemical engineering [12], topology optimization [9], optimal control [11], aerospace design [49], and design optimization [61], among others. This breadth of applications highlights the remarkable generality of NLP as a unifying framework for modeling and solving problems.

However, the generality of NLP comes at a cost. With the exception of global optimization methods [62], which are often computationally prohibitive, there are no universal guarantees of achieving global optimality, and in many cases solving NLPs remains as much an art as a science. While the usual concern is the lack of global optimality guarantees, other pathologies can occur, including failure to converge to a feasible point even when one exists. NLP solvers will do their best to find a solution, but success depends on how the problem is formulated, the choice of algorithm, its hyperparameters, and the initialization. Nonetheless, NLP remains a powerful and widely used tool, as evidenced by the popularity of general-purpose NLP solvers such as Ipopt [72].

To interface with NLP solvers, several modeling languages have been developed. Classic examples include the commercial systems AMPL [33], AIMMS [13], and GAMS [16], which are based on their own domain-specific programming languages. More recent open-source frameworks are instead embedded in general-purpose languages, such as YALMIP [56] in MATLAB, JuMP [27] in Julia, Pyomo [46, 19] in Python, and CasADi [2] in C++. These modeling languages facilitate the specification of NLPs but largely treat user-specified problem formulations as black boxes. As a result, a poorly structured formulation may be passed to the solver, making it difficult for the solver to find a solution. (We give two such examples in §4.3.)

In this paper, we take the viewpoint that an NLP modeling language should (to the extent possible) exploit the structure of the user-specified problem formulation and reformulate it to increase the likelihood that the solver succeeds. To this end, we introduce a grammar for specifying NLPs, which we call *disciplined nonlinear programming* (DNLP). To handle nonsmooth convex and concave functions, DNLP adopts the same core idea as *disciplined convex programming* (DCP) [40, 42], a grammar for specifying convex optimization problems, and analyzes monotonicity to relax nonsmooth functions into equivalent smooth formulations [41]. The popular convex optimization modeling language CVXPY [26, 1] is based on DCP, and we have implemented a rewriting system based on DNLP as an extension to CVXPY. This extension allows users to seamlessly specify NLPs as long as they conform to a minimal set of rules, and the problem is then (hopefully) solved by an NLP solver.

It is important to note, however, that the discipline imposed by DNLP does not, in itself, guarantee that a solver will succeed and be able to compute a solution. But we believe that following the DNLP ruleset increases the likelihood of successful convergence. This should be contrasted with convex optimization and DCP, where the benefits of imposing such discipline are much stronger: any formulation conforming to DCP is automatically certified as convex and can be solved reliably and efficiently to global optimality (up to some practical problem size limits and solver tolerances).

The remainder of this paper begins with a brief overview of NLP. In §3, we introduce DNLP and its (minimal) ruleset, while §4 describes the canonicalization process and explains how DNLP allows nonsmooth problems to be relaxed (without loss) into equivalent smooth formulations. Finally, in §5 we present several numerical examples.

2 Nonlinear programming

A *nonlinear program* is an optimization problem of the form

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & c(x) = 0 \\ & \ell \leq x \leq u, \end{array} \tag{1}$$

or one that can be readily converted into this form. Here, $x \in \mathbf{R}^n$ is the optimization variable, $\ell \in \mathbf{R}^n$ and $u \in \mathbf{R}^n$ are given variable bounds, and $f : \mathbf{R}^n \to \mathbf{R}$ and $c : \mathbf{R}^n \to \mathbf{R}^m$ are differentiable functions that are allowed to be nonconvex. An inequality constraint of the form $c_i(x) \leq 0$ can be expressed in this form by introducing a slack variable $s_i \geq 0$ together with the constraint $c_i(x) + s_i = 0$. An unbounded variable x_i can be specified by setting $\ell_i = -\infty$ and $u_i = \infty$.

In this section we provide a survey of NLP, including common variations on the standard form given above, algorithms and solvers, and theoretical properties. For more background we refer the reader to the many excellent textbooks on the subject [60, 31, 36, 10].

2.1 Standard forms and oracles

Many NLP solvers have been developed over the years (we name a few of these in §2.2), each with its own interface and its own standard form. While different solvers have their own standard forms, they are all closely related to (1), or they convert problems into this form internally. For example, Ipopt [72] requires constraints to be

given as two-sided inequalities of the form $\ell \leq g(x) \leq u$, and internally transforms the constraint into the form (1) by introducing a new variable s together with the equality constraint $g(x) - s = 0$ and the bounds $\ell \leq s \leq u$. Other solvers, such as Knitro [20] or SNOPT [35], allow users to specify linear constraints separately for further efficiency. Manually reformulating an optimization problem to match a solver's standard form is tedious and prone to errors. Modeling languages automate this process, allowing users to switch seamlessly between solvers with different standard forms.

In addition to transforming user-specified problems into the standard form expected by solvers, NLP modeling languages are responsible for providing oracles that evaluate the objective and constraint functions and their derivatives. Most, if not all, modeling languages construct these oracles using *automatic differentiation* [43].

2.2 Algorithms and solvers

Algorithms for solving NLPs have been studied since at least the 1940s (see, *e.g.*, [39]), but only in the past few decades, with advances in software, have these methods become accessible to a broader audience. The two most common types of algorithms implemented in modern NLP solvers are *interior-point methods* (IPMs) and *sequential quadratic programming* (SQP).

Interior-point methods reduce (1) to a sequence of equality-constrained problems by incorporating the inequality constraints into the objective using *barrier functions*. A large body of theory on barrier functions for solving NLPs was developed during the 1960s [29], but researchers lost interest in the most basic IPM—the *primal barrier method*—due to concerns about ill-conditioning [57] that later proved unfounded [74, 32]. Much later, more sophisticated IPMs for NLP were developed, and today many of the most popular solvers implement IPMs, including the open-source solvers Ipopt [72] and Uno [70], as well as the commercial solvers LOQO [71], Knitro [20], and Gurobi [44].

Sequential quadratic programming methods reduce (1) to a sequence of quadratic programs. The constraints of each quadratic subproblem are linearizations of the constraints in the original problem, and the objective is a quadratic approximation of the Lagrangian function. SQP methods were first proposed in the 1960s [73], and modern solvers implementing SQP include the commercial packages SNOPT [35], Knitro-Active [20], and WORHP [18], as well as the open-source solver GRANSO [25, 55].

While IPMs and SQP methods are the most commonly implemented algorithms, several solvers also implement *augmented Lagrangian methods*, including the open-

source solver Algencan [3] and the commercial solvers MINOS [58], Lancelot [24], and the recent Knitro-Augmented [7]. These methods reduce (1) to a sequence of subproblems in which the objective is the Lagrangian augmented with a penalty term for constraint violations, where some or all of the constraints are incorporated into the penalty and the remaining constraints are enforced explicitly.

Given the many NLP solvers available, a natural question is which solver to use for a given problem. While there is no definitive answer, the conventional wisdom is that IPMs are faster and more reliable when solving a problem from scratch, *i.e.*, without a good initial point [37]. However, IPMs such as Ipopt may struggle with problems that violate standard regularity conditions (see, *e.g.*, [12, §11] or [69]), in which case augmented Lagrangian methods can be more robust [48]. For example, Knitro states on their website that the primary advantage of their augmented Lagrangian method over IPMs is that it is "designed to better handle difficult problems with degenerate constraints where the linear independence constraint qualification (LICQ) is not satisfied". Nevertheless, we recommend trying Ipopt first, because it is open-source, widely adopted (as evidenced by its citation count), and performs well across many applications. In our experience, it works very well.

2.3 Theoretical properties

Because NLP covers a vast range of problems, including many known to be NP-hard, it is unrealistic to expect NLP solvers to guarantee convergence to a *global* minimizer (*i.e.*, a feasible point achieving the smallest possible objective value). In fact, a common misconception is that solvers are guaranteed to converge even to *local* minimizers (*i.e.*, feasible points achieving the smallest objective value within some neighborhood). In practice, most solvers at best guarantee (under regularity conditions on the constraints) convergence to a point that approximately satisfies a set of necessary but not sufficient optimality equations known as the *Karush-Kuhn-Tucker* (KKT) conditions [10, §4]. When an NLP solver claims that it has solved a nonlinear program, it typically means that it has found a KKT point, *i.e.*, a point satisfying the KKT conditions within some tolerances. However, not all KKT points are local minimizers, so solvers incorporate various heuristics and techniques to steer iterates away from such undesirable KKT points.

While not every KKT point is a local minimizer, the converse is true under so-called *constraint qualifications* [10, §4.3.4]. The derivation of the KKT conditions is based on the idea of linearizing the constraints around a local minimizer, and constraint qualifications are conditions that ensure that this linearization is a good approximation of the true constraints. A common constraint qualification assumed

by NLP solvers is the *linear independence constraint qualification* (LICQ), which for problem (1) requires that the set consisting of the gradients of the active bound constraints and the gradients of the equality constraints is linearly independent. If LICQ holds at a local minimizer $x^\star$, then $x^\star$ is guaranteed to also be a KKT point, and the so-called *Lagrange multipliers* which are auxilliary variables in the KKT conditions, are guaranteed to be unique. LICQ also seems to play a role in practice. If LICQ does not hold at a local minimizer, some solvers are less robust and may fail to converge.

3 Disciplined nonlinear programming

Nondifferentiable functions often arise in applications and pose significant challenges for most NLP solvers. A naive approach is to simply ignore these nondifferentiabilities or assume they will not occur in practice, but this often leads to poor performance and solver failure. The difficulty is that the points of nondifferentiability are often precisely the points of interest. For example, in problems with ℓ_1-regularization, the goal is typically to find a solution in which the argument is sparse, which is a point where the ℓ_1 norm is nondifferentiable.

To support nondifferentiable functions in nonlinear programs without compromising solver reliability, we introduce the notion of *disciplined nonlinear programming* (DNLP). It consists of two key components:

- An *atom library*—a collection of functions that can be used to describe a problem. These functions have known attributes including smoothness, sign, monotonicity, and curvature.
- The *DNLP ruleset*—a set of rules specifying how atoms may be combined to form more complicated expressions, and how these expressions may appear in objectives and constraints.

This framework guarantees that any problem with nonsmooth functions complying with the DNLP ruleset admits an equivalent smooth formulation that takes standard NLP regularity conditions, such as LICQ, into account. For problems that only involve smooth functions, DNLP imposes no additional restrictions.

DNLP is heavily inspired by DCP, so readers familiar with DCP will find many similarities. Roughly speaking, DNLP mirrors the structure of DCP, with smooth functions playing the role of affine functions, and generalizations of convex and concave functions that can be mixed with smooth functions.

3.1 Atoms

The rules of DNLP depend on the smoothness and curvature properties of atoms. We classify atoms into three categories: *smooth*, *nonsmooth-convex* (NS-convex), and *nonsmooth-concave* (NS-concave). We list some atoms and their classifications in table 1.

Smooth atoms. An atom is *smooth* if it is twice continuously differentiable in the *interior* of its domain. For example, the atoms $\phi_{\log}$ and ϕ_{sqrt} defined by $\phi_{\log}(x) = \log x$ with $\mathbf{dom}\,\phi_{\log} = \mathbf{R}_{++}$, and $\phi_{\text{sqrt}}(x) = \sqrt{x}$ with $\mathbf{dom}\,\phi_{\text{sqrt}} = \mathbf{R}_{+}$, are both smooth, and so is any affine or trigonometric atom. In contrast, the atom ϕ_{abs} defined by $\phi_{\text{abs}}(x) = |x|$ with $\mathbf{dom}\,\phi_{\text{abs}} = \mathbf{R}$ is not smooth.

Nonsmooth-convex atoms. An atom is *NS-convex* if it is convex and not twice continuously differentiable in the interior of its domain. Two examples are $\phi_{\max}$ and ϕ_{norm2} defined by $\phi_{\max}(x, y) = \max(x, y)$ and $\phi_{\text{norm2}}(x) = \|x\|_2$. (The latter is not differentiable at $x = 0$.)

Nonsmooth-concave atoms. An atom is *NS-concave* if it is concave and not twice continuously differentiable in the interior of its domain. Two examples are $\phi_{\min}$ and $\phi_{\text{sum\_smallest}}$ defined by $\phi_{\min}(x, y) = \min(x, y)$ and $\phi_{\text{sum\_smallest}}(x) = \sum_{i=n-k+1}^{n} x_{[i]}$, where $x_{[i]}$ is the ith largest element of $x \in \mathbf{R}^n$, and $k \in \{1, \ldots, n\}$ is a fixed parameter.

Additional attributes. Functions in the atom library are also characterized by their sign and monotonicity. Three categories of monotonicity are considered: *nondecreasing*, *nonincreasing*, and *nonmonotonic*. The usual mathematical definitions of monotonicity apply. For functions with multiple arguments, we specify the monotonicity with respect to each argument separately. Furthermore, we use *sign-dependent* monotonicity, *i.e.*, the monotonicity of an atom can depend on the signs of its arguments. For example, the atom defined by $\phi(x) = x^3$ is classified as nondecreasing for $x \geq 0$.

3.2 Expressions

An *expression* is recursively defined as an atom evaluated at a *subexpression*. The subexpression can be a variable, a constant, or another expression itself. Mathematically, an expression is of the form $f(x) = \phi(g(x))$ where ϕ is the atom and $g(x) = (g_1(x), \ldots, g_k(x))$ is its argument, the subexpression. We classify expressions

Table 1: Some atoms and their classifications. If the domain of an atom is not specified, it means that the atom has full domain.

Atom	Definition	Domain
	Smooth, nonconvex and nonconcave	
`multiply`	$\phi(x, y) = xy$	
`quad_form`	$\phi(x) = x^T Q x$ where $Q \in \mathbf{S}^n$	
`sin`	$\phi(x) = \sin x$	
`cos`	$\phi(x) = \cos x$	
`tan`	$\phi(x) = \tan x$	$x \in (-\pi/2, \pi/2)$
`sinh`	$\phi(x) = (e^x - e^{-x})/2$	
`tanh`	$\phi(x) = (e^x - e^{-x})/(e^x + e^{-x})$	
`asinh`	$\phi(x) = \ln(x + \sqrt{x^2 + 1})$	
`atanh`	$\phi(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$	$x \in (-1, 1)$
`sigmoid`	$\phi(x) = \frac{1}{1+e^{-x}}$	
	Smooth, convex or concave	
`exp`	$\phi(x) = e^x$	
`log`	$\phi(x) = \log x$	$x > 0$
`log_sum_exp`	$\phi(x) = \log\left(\sum_{i=1}^n e^{x_i}\right)$	
`power`	$\phi(x) = x^p$ where $p > 0$ is an integer	
`power_pos`	$\phi(x) = x^p$ where $p > 0$	$x \geq 0$
`sqrt`	$\phi(x) = \sqrt{x}$	$x \geq 0$
`inv_pos`	$\phi(x) = 1/x$	$x > 0$
`quad_over_lin`	$\phi(x, y) = x^T x/y$	$y > 0$
	Nonsmooth, convex	
`abs`	$\phi(x) = \lvert x \rvert$	
`max`	$\phi(x) = \max\{x_1, x_2, \ldots, x_n\}$	
`norm1`	$\phi(x) = \lVert x \rVert_1$	
`norm2`	$\phi(x) = \lVert x \rVert_2$	
`norm_inf`	$\phi(x) = \lVert x \rVert_\infty$	
`huber`	$\phi(x) = \begin{cases} x^2, & \lvert x \rvert \leq M \\ 2M\lvert x \rvert - M^2, & \lvert x \rvert > M, \end{cases}$ where $M \geq 0$	
`sum_largest`	$\phi(x) = \sum_{i=1}^k x_{[i]}$ where $k \in \{1, \ldots, n\}$	
	Nonsmooth, concave	
`min`	$\phi(x) = \min\{x_1, x_2, \ldots, x_n\}$	
`sum_smallest`	$\phi(x) = \sum_{i=n-k+1}^k x_{[i]}$ where $k \in \{1, \ldots, n\}$	

into the three categories *smooth*, *linearizable-convex* (L-convex), and *linearizable-concave* (L-concave).

Smooth expressions. An expression $f(x) = \phi(g(x))$ is defined to be *smooth* if both the atom ϕ and the subexpression $g(x)$ are smooth. Constant expressions and variable expressions are considered as smooth, so any smooth atom ϕ, evaluated at variables or constants, is a smooth expression $\phi(x)$.

L-convex expressions. An expression $f(x) = \phi(g(x))$ is defined to be *L-convex* if the atom ϕ is smooth or NS-convex, and for each $i = 1, \ldots, k$, one of the following holds: $g_i(x)$ is smooth; or $g_i(x)$ is L-convex and ϕ is nondecreasing in its ith argument; or $g_i(x)$ is L-concave and ϕ is nonincreasing in its ith argument.

L-concave expressions. An expression $f(x) = \phi(g(x))$ is defined to be *L-concave* if the atom ϕ is smooth or NS-concave, and for each $i = 1, \ldots, k$, one of the following holds: g_i is smooth; or g_i is L-convex and ϕ is nonincreasing in its ith argument; or g_i is L-concave and ϕ is nondecreasing in its ith argument.

Simple consequences of the definitions. We mention that any smooth expression is also both L-convex and L-concave. Furthermore, the sum of two L-convex expressions is L-convex, and the sum of two L-concave expressions is L-concave. (All these statements follow directly from the definitions of L-convexity and L-concavity.) This logic is analogous to how, in convex optimization, affine expressions are both convex and concave, and the sum of two convex (concave) expressions is convex (concave).

3.3 Constraints and objectives

For an optimization problem to be a *disciplined nonlinear program*, its objective and constraints must satisfy the following rules.

Objective. A valid objective is either the minimization of an L-convex expression or the maximization of an L-concave expression. Maximizing an L-convex expression or minimizing an L-concave expression is not valid (unless the expression is also smooth).

Constraints. A valid constraint is one of the following:

- An equality constraint between a smooth left-hand side (LHS) and a smooth right-hand side (RHS).
- A less-than-or-equal-to inequality with an L-convex LHS and an L-concave RHS.
- A greater-than-or-equal-to inequality with an L-concave LHS and an L-convex RHS.

A problem description that conforms to these rules is called *DNLP-compliant*. We will see that such a problem formulation can be canonicalized to an equivalent (smooth) NLP without introducing LICQ violations.

3.4 Examples

DNLP expressions. We now give a few examples of expressions that conform to the DNLP ruleset, and others that do not.

- The function $f(x, y) = x/y$ with $y > 0$ can be expressed as

  $$\texttt{multiply}(x, \texttt{inv\_pos}(y)).$$

  When expressed this way, $f(x)$ is a smooth expression since it is the composition of the smooth atom `multiply` with two smooth expressions. (A variable or a smooth atom by itself is considered a smooth expression; see §3.2.)

- The function $f(x) = c^T x/(x^T A x)$ with $A \in \mathbf{S}^n_{++}$ can be expressed as

  $$\texttt{multiply}(c \texttt{ @ } x, \texttt{inv\_pos}(\texttt{quad\_form}(x, A))).$$

  When expressed this way, $f(x)$ is a smooth expression since it is the composition of the smooth atom `multiply` with two smooth expressions. (The second argument of `multiply` is a smooth expression since it is itself the composition of the smooth atom `inv_pos` with a smooth expression.)

- The function $f(x) = |c^T x/(x^T A x) - b|$ with $A \in \mathbf{S}^n_{++}$ can be expressed as

  $$\texttt{abs}(\texttt{multiply}(c \texttt{ @ } x, \texttt{inv\_pos}(\texttt{quad\_form}(x, A))) - b).$$

  When expressed this way, $f(x)$ is an L-convex expression since it is the composition of the NS-convex atom `abs` with a smooth expression.

- The function $f(x) = (\|x - a\|_2 - b)^2$ can be expressed as

  $$\texttt{square}(\texttt{norm2}(x - a) - b).$$

  When expressed this way, $f(x)$ is *not* DNLP-compliant since the atom `square` is not monotone and its argument is not smooth. However, when we rewrite it as $f(x) = (\sqrt{\|x - a\|_2^2} - b)^2$ and express it as

  $$\texttt{square}(\texttt{sqrt}(\texttt{sum\_squares}(x - a)) - b),$$

  then the expression is smooth since it is the composition of the smooth atom `square` with a smooth expression. (The first term of the argument of `square` is a smooth expression since it is itself the composition of the smooth atom `sqrt` with a smooth expression.)

- The function $f(x) = (\sin x)^2$ can be expressed as $\texttt{square}(\texttt{sin}(x))$. When expressed this way, $f(x)$ is a smooth expression since it is the composition of the smooth atom `square` with a smooth expression.

- The function $f(x) = |x|^2$ can be expressed as $\texttt{square}(\texttt{abs}(x))$. When expressed this way, $f(x)$ is *not* a smooth expression since the atom `abs` is not smooth. Although $f(x)$ simplifies to the differentiable function $f(x) = x^2$, the expression as written is not smooth.

DNLP objectives and constraints. DNLP supports many types of nonconvex objectives and constraints.

- An avoidance constraint of the form $\|x-a\|_2 \geq r$, where $a \in \mathbf{R}^n$ and $r \in \mathbf{R}_+$ are given, can be expressed as `sum_squares(x - a) >= r ** 2`. This is DNLP-compliant since the left-hand side is an L-concave expression (as it is a smooth expression) and the right-hand side is an L-convex expression (as it is constant and thus a smooth expression).

- A discretized dynamics constraint of the form $x_1 = x_0 + s\cos(\theta)$, where x_1, x_0, s, and θ are variables, can be expressed as `x1 == x0 + multiply(s, cos(theta))`. This is DNLP-compliant since both sides are smooth expressions.

- Minimizing an objective function of the form $\|(Ax)^2 - b\|_1$, where the square is taken elementwise, is DNLP-compliant when the objective is expressed as the L-convex expression `norm1(square(A @ x) - b)`.

In §5 we will see applications where constraints and objectives of these forms arise.

3.5 Connection to DCP

As discussed in previous sections, DNLP is closely related to DCP. We now make this relationship explicit.

> *A DNLP-compliant problem is one that is DCP if all its smooth atoms are treated as affine.*

In other words, a DNLP-compliant problem is one that becomes DCP if all the smooth atoms are linearized. More precisely, the following statements hold.

- An L-convex expression is one that becomes convex when all smooth atoms are linearized around some current point. This property justifies the terminology *linearizable-convex*; such expressions are not necessarily convex, but they become convex after linearizing each smooth atom they contain.
- An L-concave expression is one that becomes concave after linearizing all smooth atoms it contains.

4 Canonicalization

In this section we describe how problems conforming to DNLP are canonicalized to a standard NLP form. Our canonicalization differs from the approach adopted by most NLP modeling languages, in which the user-specified problem is not transformed, and automatic differentiation is used to provide derivative oracles for the objective and constraint functions. In contrast, in DCP-based modeling systems for convex optimization, the core idea is to perform extensive transformations of the original problem formulation into a standard *conic form* [4, 59, 14], which obviates the need for derivative oracles based on automatic differentiation. This approach gracefully also handles functions that are nondifferentiable or defined only on a restricted domain. We adopt a similar approach to canonicalize problems conforming to DNLP.

4.1 The canonical form

The first step of canonicalization is a *parser* that processes the user-specified problem and constructs one *expression tree* for the objective and two for each constraint, the left-hand and right-hand sides. In an expression tree, each inner node represents an atom, with its children corresponding to the arguments of the atom. This is illustrated in figure 1 for the function $f(x) = |x^T A x + c|$ where $A \in \mathbf{S}_n$ and $c \in \mathbf{R}$

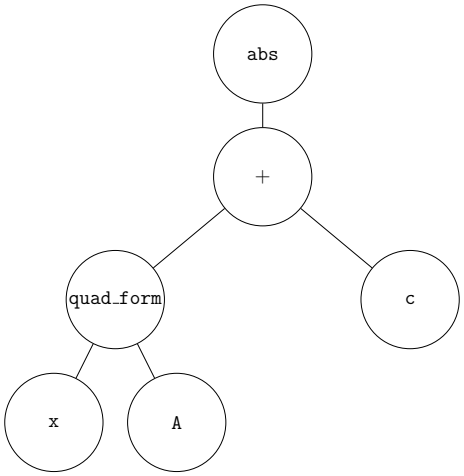

Figure 1: Expression tree for the L-convex expression `abs(quad_form(x, A) + c)`.

are parameters (constants), and $x \in \mathbf{R}^n$ is the variable, represented by the DNLP-compliant expression `abs(quad_form(x, A) + c)`.

Once the expression trees are constructed, the parser traverses them from the leaves to the root to determine the smoothness classification of each expression using the definitions given in §3.2. Finally, it verifies that the objective and constraints conform to the DNLP ruleset described in §3.3.

After the parser has verified that the problem conforms to DNLP, the *canonicalizer* traverses the expression trees from the root to the leaves and transforms the user-specified problem, distinguishing between how smooth and nonsmooth atoms are treated.

Smooth atoms. When a node corresponding to a smooth atom is encountered, we first check whether the atom has full domain. If not, we introduce auxiliary variables for its arguments and add constraints linking these new variables to the original arguments. We also specify bounds on the new variables to explicitly encode the domain of the atom. If a smooth atom has full domain but its arguments are not variables by themselves, we likewise introduce auxiliary variables and constraints to represent those arguments.

A simple example illustrating how smooth atoms are canonicalized is the problem

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^m \log(b_i - a_i^T x) + \|Cx - d\|_2^2 \\ \text{subject to} & \|x\|_2^2 \leq 1, \end{array}$$

with variable x. The corresponding canonicalized problem is

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^{m} \log(t_i) + \|v\|_2^2 \\ \text{subject to} & \|x\|_2^2 \leq 1 \\ & t = b - Ax \\ & v = Cx - d \\ & t \geq 0, \end{array}$$

where the variables are (the original one) x and (the new ones) t and v. Here, t was introduced for the argument of the logarithm since the log-atom has restricted domain, and v was introduced for the argument to the squared Euclidean norm in the objective since the argument was not a variable by itself. No new variable was introduced for the argument to the squared Euclidean norm in the constraints, since the atom has full domain and the argument is already a variable by itself. Also note that we explicitly added the bound $t \geq 0$. (Explicitly communicating function domains via bounds to the solver makes them more robust.)

For a problem that only involves smooth atoms, this procedure for traversing the expression trees results in an equivalent problem formulation similar to a canonical form proposed by Smith [65, 66], known as the *Smith form*, with the minor modification that we always introduce new variables for the arguments of atoms lacking full domain. (In the original definition of the Smith form, a variable is never introduced for the argument of an atom if the argument is a variable by itself [66, table 1].) Another distinction from our approach is that the original Smith form always converts problems into *graph form*, *i.e.*, each nonlinear atom ϕ is replaced by an auxiliary variable t together with the equality constraint $t = \phi(x)$. For example, instead of minimizing $\phi(x)$ directly, one minimizes t subject to the constraint $t = \phi(x)$ over x and t.

Nonsmooth atoms. When a node corresponding to a nonsmooth atom ϕ is encountered, we replace the atom with an auxiliary variable t and add the constraint $t = \phi(x)$. Next, we relax this constraint to $t \geq \phi(x)$ if ϕ is NS-convex, or to $t \leq \phi(x)$ if ϕ is NS-concave. When the original problem is DNLP, this relaxation does not change the optimal value of the problem or the set of optimal x-values, *i.e.*, the relaxation is *lossless*. Finally, we express the relaxed constraint using a smooth reformulation, as we will describe in §4.2.

Table 2: Smooth epigraph and hypograph formulations of nonsmooth atoms.

Atom	Definition	Smoothness	Epigraph / Hypograph Implementation
`abs`	$\phi(x) = \lvert x \rvert$	NS-convex	Epigraph: $-t \leq x \leq t$
`max`	$\phi(x) = \max(x, y)$	NS-convex	Epigraph: $x \leq t$, $y \leq t$
`norm1`	$\phi(x) = \lVert x \rVert_1$	NS-convex	Epigraph: $-v \leq x \leq v$, $\mathbf{1}^T v \leq t$
`norm2`	$\phi(x) = \lVert x \rVert_2$	NS-convex	Epigraph: `quad_over_lin`$(x, t) - t \leq 0$
`norm_inf`	$\phi(x) = \lVert x \rVert_\infty$	NS-convex	Epigraph: $-t\mathbf{1} \leq x \leq t\mathbf{1}$
`sum_largest`	$\phi(x) = \sum_{i=1}^{k} x_{[i]}$	NS-convex	Epigraph: [14, Exercise 5.19]
`min`	$\phi(x) = \min(x, y)$	NS-concave	Hypograph: $x \geq t$, $y \geq t$
`sum_smallest`	$\phi(x) = \sum_{i=n-k+1}^{n} x_{[i]}$	NS-concave	Hypograph: [14, Exercise 5.19]

4.2 Smooth epigraph formulations

As described in the previous section, any problem conforming to DNLP is equivalent to a problem in which any atom that is not smooth appears in a constraint of the form $t \geq \phi(x)$ if ϕ is NS-convex, or $t \leq \phi(x)$ if ϕ is NS-concave. The sets $\{(x, t) \mid t \geq \phi(x)\}$ and $\{(x, t) \mid t \leq \phi(x)\}$ are known as the *epigraph* and *hypograph* of ϕ, respectively. Table 2 describes how these are transformed into smooth formulations that satisfy LICQ. Most of these transformations are standard and covered in introductory linear programming classes. Automating them is nevertheless valuable, as the procedure can be tedious and error-prone, especially when the original problem involves compositions of atoms.

For every atom in table 2, the smooth reformulation is equivalent to the original epigraph or hypograph constraint, with one exception. Specifically, for the `norm2` atom, the point $(x, t) = (0, 0)$ belongs to the epigraph but does not satisfy the smooth reformulation, since the domain of `quad_over_lin` is $t > 0$ (see table 1). Thus, the smooth reformulation excludes this single point from the feasible set.

Conceptually, this exclusion closely parallels the behavior of interior-point methods for conic convex optimization such as MOSEK [5], which represent the epigraph of the `norm2` atom via a second-order cone constraint. These solvers use barrier functions that enforce strict feasibility with respect to the cone, ensuring that iterates remain in the cone interior and thus never reach the origin.

4.3 Two advantages of our canonicalization

Initialization. One advantage of our canonicalization procedure is that it simplifies the task of specifying an initial point when atoms have restricted domains. Our canonical form ensures that the argument to any atom with a restricted domain is a variable t that appears only as an argument to that atom and in a constraint of the form $t = f(x)$, where $f(x)$ is an arbitrary expression. Since solvers require an initial point that lies within the domain of all objective and constraint functions, we can simply initialize each t within the domain of its corresponding atom, without ensuring that the constraint $t = f(x)$ holds initially. This is straightforward to implement by providing an atom-specific oracle that returns a default initial value within the atom's domain. All of this is automated and handled internally, so the user does not need to worry about it. (If a good starting point for the original variables is known, the user should of course specify it manually. In this case, we propagate the starting point to the auxilliary variables introduced during canonicalization by evaluating the expressions defining them at the given starting point for the original variables.)

Without this approach, the task of finding an initial point in the intersection of the domains of the objective and constraint functions falls to the user — a task that can be highly nontrivial. For example, consider computing the analytic center of a polyhedron of the form

$$\{x \in \mathbf{R}^n \mid a_i^T x \leq b_i,\ i = 1, \ldots, m\},$$

which can be done by minimizing the function

$$f(x) = -\sum_{i=1}^{n} \log(b_i - a_i^T x).$$

This function is convex and smooth, so we expect the problem to be readily solved by a solver like Ipopt. For a problem instance where the polyhedron does not contain the origin, Ipopt crashes in its first iteration when we interface it using popular NLP modeling languages such as AMPL, GAMS, JuMP, Pyomo, and CasADi. The reason is that these modeling languages choose the default initial point to be the origin, which lies outside the domain of the objective function. In contrast, when we interface Ipopt using our modeling language, it successfully solves the same problem instance in 14 iterations.

Nonsmoothness. Another advantage of our canonicalization procedure is that it seems more robust for problems involving nonsmooth functions than other NLP

modeling languages that are not based on DNLP. For example, consider the sparse linear regression problem

$$\text{minimize } \|Ax - b\|_2^2 + \lambda\|x\|_1,$$

with variable x, where $\lambda > 0$ is a regularization parameter. For this problem we expect the solution to occur at a point of nondifferentiability. When we interface Ipopt using our modeling language, it gracefully solves a random problem instance in 12 iterations. In contrast, when we specify the same problem instance using AMPL, GAMS, JuMP, Pyomo, and CasADi, Ipopt fails to solve it and terminates after reaching its maximum number of 3000 iterations. The issue is that these modeling languages treat the objective as a black box and supply Ipopt with derivatives via automatic differentiation, even though the second term is nondifferentiable at the solution.

4.4 Our implementation

We have implemented DNLP as an extension to the DCP-based modeling language CVXPY. The implementation is currently available as a standalone package at

https://github.com/cvxgrp/DNLP,

with plans for future integration into the main CVXPY distribution. Problems are expressed using standard CVXPY syntax, augmented with smooth nonconvex and nonconcave atoms including those listed in table 1. (These atoms have previously not been available in CVXPY, since DCP rules only permit atoms that are either convex or concave.) For several common atoms we support simpler syntax as a convenience; for example, squaring all entries of a vector-valued expression `expr` can be done using both `square(expr)` and `expr ** 2`.

Some useful functions and features. The most useful functions and features of the DNLP extension are summarized below.

- `problem.is_dnlp()` returns a Boolean indicating whether the problem is DNLP.
- `problem.solve(nlp=True)` carries out DNLP canonicalization and invokes the default NLP solver on the canonicalized problem (assuming the specified problem is DNLP). The flag `nlp=True` explicitly instructs CVXPY to treat the problem as a nonlinear program. If omitted, CVXPY attempts to canonicalize the problem under DCP rules and raises an error if the problem is not DCP.

- The `solve()` method also accepts the optional flag `best_of=N`, where `N` is a positive integer. When provided, the problem is solved `N` times from different random initializations, and the best solution found is returned. The random starting point for a variable `x` is drawn uniformly from a user-specified box given by the attribute `x.sample_bounds`. If `best_of` is used but `x.sample_bounds` is not provided, no random initialization is done for that variable, unless the variable has finite lower and upper bounds. In that case, the variable is initialized uniformly at random within its bounds.

- As in CVXPY, the `solve()` method accepts the optional flag `solver='solver_name'`, to specify that the NLP solver `solver_name` should be used. Directives and options can be passed to the solver as additional keyword arguments to the `solve()` method.

- The variable attribute `x.value` can be used to manually set the initial value for a variable `x`.

Supported solvers. Currently, we support the open-source interior-point solver Ipopt [72] and the commercial solver Knitro [20]. Knitro implements several algorithms for nonlinear optimization, including an interior-point method and an augmented Lagrangian method. These can be selected by specifying `solver='knitro_ipm'` or `solver='knitro_alm'` in the `solve()` method, respectively. For example, to use Knitro's interior-point method, one would write

```
problem.solve(nlp=True, solver='knitro_ipm').
```

5 Numerical examples

In this section we present several simple examples illustrating our DNLP-based modeling language. Most of these can be implemented in fewer than 10 lines of code, and they are available at https://github.com/cvxgrp/DNLP-examples. The examples were solved using Ipopt, unless otherwise specified.

The code snippets below avoid for-loop constructs where possible, using vectorized operations instead by specifying axis arguments to various atoms. This can have a significant performance benefit, so we encourage users to do so in their own code.

5.1 Path planning with obstacles

Problem. We seek the shortest path connecting points a and b in $\mathbf{R}^d$ that avoids m circles, centered at p_j with radius r_j, $j = 1, \ldots, m$ [51, 64]. After discretizing the arc-length-parametrized path into a sequence of points $x_0, \ldots, x_n$, the problem can be written as

$$\begin{array}{ll} \text{minimize} & L \\ \text{subject to} & x_0 = a, \quad x_n = b \\ & \|x_{i+1} - x_i\|_2^2 \leq (L/n)^2, \quad i = 0, \ldots, n-1 \\ & \|x_i - p_j\|_2^2 \geq r_j^2, \quad i = 1, \ldots, n-1, \quad j = 1, \ldots, m, \end{array}$$

where L and x_i are variables, and a, b, p_j, and r_j are given.

DNLP specification. The code specifying this problem is given below.

```
x, L = Variable((d, n + 1)), Variable()
constr = [x[:, 0] == a, x[:, n] == b,
          sum((x[:, 1:] - x[:, :-1]) ** 2, axis=0) <= (L / n) ** 2]
for i in range(n):
    constr += [sum((x[:, i] - p) ** 2, axis=1) >= r ** 2]
prob = Problem(cp.Minimize(L), constr)
x.value = ...  # initialize to straight line path
prob.solve(nlp=True)
```

Alternative DNLP-compliant formulations. The constraint $\|x_{i+1} - x_i\|_2^2 \leq (L/n)^2$ can also be expressed as $\|x_{i+1} - x_i\|_2 \leq L/n$, which is DNLP-compliant because the left-hand side is L-convex. Since the objective is decreasing in L, these constraints are tight at optimality, so we can also replace them by equalities of the form $\|x_{i+1} - x_i\|_2^2 = (L/n)^2$. A constraint of this form is DNLP-compliant, as its left-hand side is smooth. (Among these three formulations, the first one converges in the fewest iterations in our experiments.)

Results. We consider a problem instance with dimension $d = 2$, $n = 50$ path segments, and $m = 5$ obstacles. Figure 2 shows the solution to this problem instance, when initialized as the straight line path from a to b. For other initializations, the final path is different.

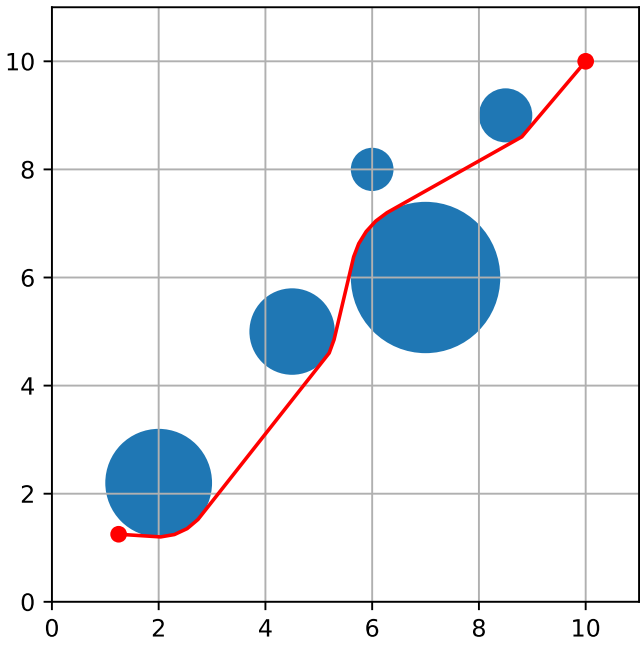

Figure 2: Shortest path connecting two points while avoiding circular obstacles.

5.2 Circle packing

Problem. The goal is to arrange n circles in $\mathbf{R}^2$ with given radii r_i for $i = 1, \ldots, n$, so that they do not overlap and are contained in the smallest possible square [68, 47]. The optimization problem can be formulated as

$$\begin{array}{ll} \text{minimize} & \max_{i=1,\ldots,n} \left(\|c_i\|_\infty + r_i\right) \\ \text{subject to} & \|c_i - c_j\|_2^2 \geq (r_i + r_j)^2, \quad 1 \leq i < j \leq n, \end{array}$$

where the variables are the centers of the circles $c_i \in \mathbf{R}^2$, $i = 1, \ldots, n$, and the radii r_i are given. If L is the value of the objective function, the circles are contained in the square $[-L, L]^2$.

DNLP specification. The code specifying this problem is given below.

```
c, constr = Variable((n, 2)), []
for i in range(n - 1):
    constr += [sum((c[i, :] - c[i+1:, :]) ** 2, axis=1) >=
                   (r[i] + r[i+1:]) ** 2]
cost = max(norm_inf(c, axis=1) + r)
prob = Problem(Minimize(cost), constr)
c.value = uniform(-5.0, 5.0, (n, 2)) # random initial point
prob.solve(nlp=True)
```

Results. We consider a problem instance with $n = 10$ circles, with each radius sampled from a uniform distribution over the interval $[1, 3]$. Figure 3 shows one solution to this problem instance, when initialized with random center locations. The fraction of the square covered by the circles is 0.72.

To solve the problem multiple times with different random initializations, we can replace the line `prob.solve(nlp=True)` in the code snippet above with

```
c.sample_bounds = [-5.0, 5.0]
prob.solve(nlp=True, best_of=500).
```

This solves the problem instance 500 times with different random initializations for the circle centers, each drawn uniformly from the square $[-5, 5]^2$. With this approach, the fraction of the square covered by the circles is 0.77 for the best solution found. Figure 4 shows the best solution, along with a histogram of the coverages obtained across all initializations.

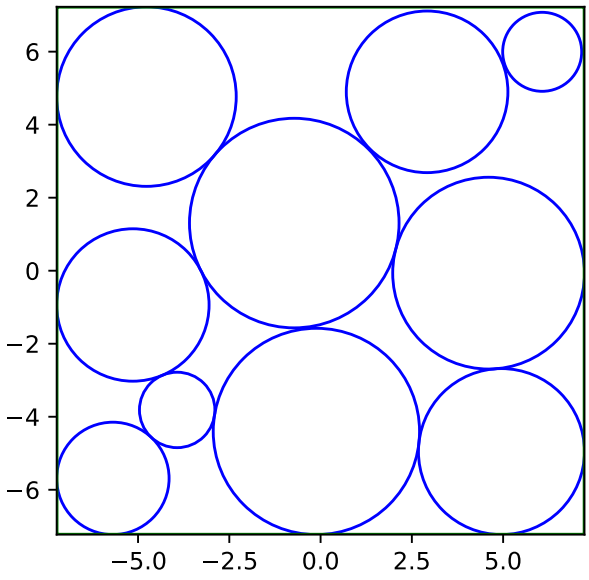

Figure 3: Circle packing.

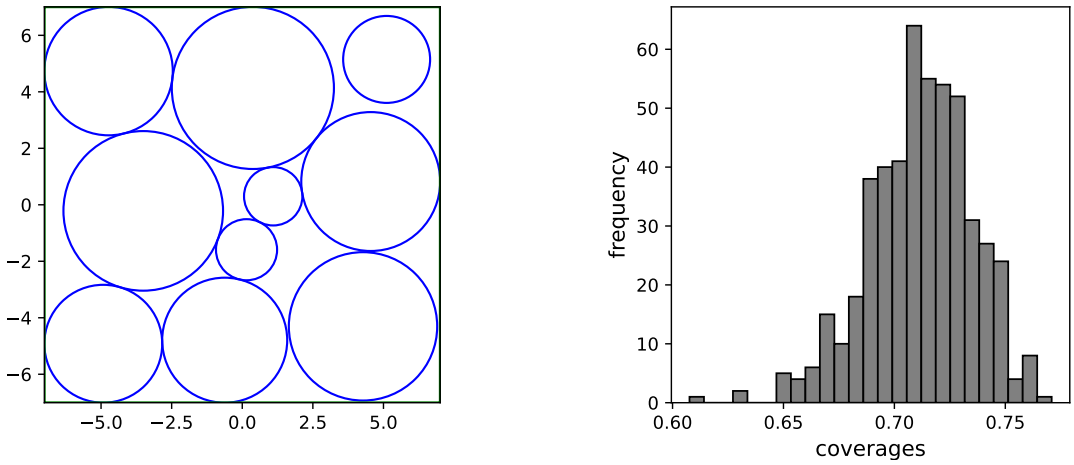

Figure 4: The best circle packing found over 500 random initializations (left), and a histogram of the coverages obtained across all initializations (right).

5.3 Location from range measurements

Problem. The goal is to estimate the position of an object from noisy range (distance) measurements ρ_i to known anchor points a_i in $\mathbf{R}^2$ for $i = 1, \ldots, m$ [67, 8]. We formulate the problem as

$$\text{minimize} \quad \sum_{i=1}^{m} (\|x - a_i\|_2 - \rho_i)^2, \tag{2}$$

where the variable is the object position $x \in \mathbf{R}^2$, and the anchor points a_i and range measurements ρ_i are given.

DNLP specification. The code specifying this problem is given below. To get a DNLP-compliant formulation, we express $\|x - a_i\|_2$ as $\sqrt{\|x - a_i\|_2^2}$ (see §3.4).

```
x = Variable(2)
cost = sum_squares(sqrt(sum((x - a) ** 2, axis=1)) - rho)
problem = Problem(Minimize(cost))
problem.solve(nlp=True)
```

Results. We consider a problem instance with $m = 10$ anchor points, each sampled from a uniform distribution over the square $[-5, 5]^2$. We added zero-mean Gaussian noise with unit standard deviation to the true range measurements. Figure 5 shows the solution to this problem instance, with the initial point set to the origin. The dashed circle around each anchor represents the range measurement from that anchor. The left figure shows the anchors and range measurements without any noise, and the true location of the object is at the intersection of the circles. The right figure shows the noisy range measurements and the estimated location.

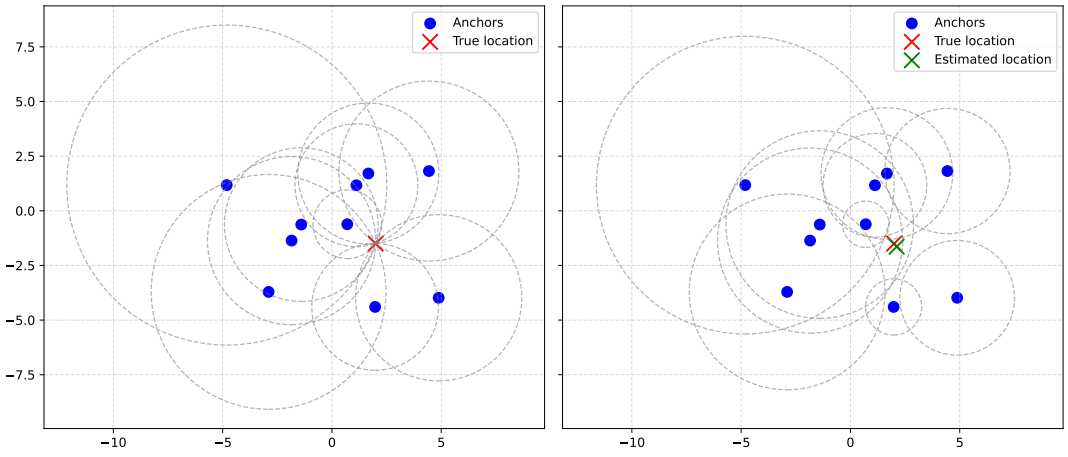

Figure 5: Location estimation from range measurements. *Left.* Range measurements without noise. *Right.* Range measurements with noise.

5.4 Nonnegative matrix factorization

Problem. The goal is to approximate a given nonnegative matrix $A \in \mathbf{R}^{m\times n}$ as the product of two nonnegative matrices $X \in \mathbf{R}^{m\times k}$ and $Y \in \mathbf{R}^{k\times n}$, where k is a given positive integer [54, 38]. One formulation of the problem is

$$\begin{array}{ll} \text{minimize} & \|A - XY\|_F^2 \\ \text{subject to} & X \geq 0, \quad Y \geq 0, \end{array} \tag{3}$$

where the variables are the matrices X and Y, and $\|\cdot\|_F$ denotes the Frobenius norm.

DNLP specification. The code specifying this problem is given below.

```
X = Variable((m, k), bounds=[0, None])
Y = Variable((k, n), bounds=[0, None])
X.value, Y.value = rand(m, k), rand(k, n)  # random initialization
cost = sum_squares(A - X @ Y)
prob = Problem(cp.Minimize(cost))
prob.solve(nlp=True)
```

Results. We use nonnegative matrix factorization to decompose images into basis images [53]. First, we generate 100 images of size 20×20 as random nonnegative combinations of three geometric shapes (a circle, a square, and a triangle), and then we add noise. After stacking the vectorized noisy images as columns of a matrix $A \in \mathbf{R}^{400\times 100}$, we solve (3) with $k = 3$ to recover the underlying shapes. Figure 6 shows the true basis images followed by the recovered ones (first row), six of the original images (second row), the same six images after adding noise (third row), and the denoised images (fourth row) which are given as columns of $X^\star Y^\star$, where $(X^\star, Y^\star)$ is the solution to (3).

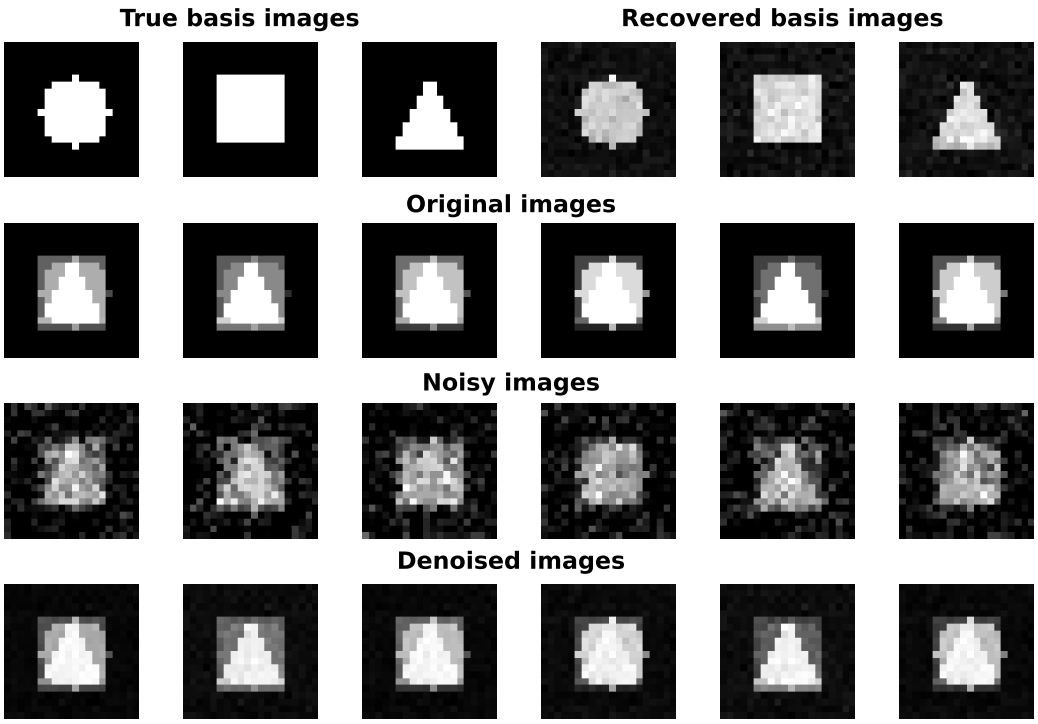

Figure 6: Nonnegative matrix factorization for decomposing images into parts.

5.5 Phase retrieval

Problem. The goal is to recover a signal $x \in \mathbf{C}^n$ from the magnitudes of the complex inner products $a_k^H x$, $k = 1, \ldots, m$, where $a_k \in \mathbf{C}^n$ are given measurement vectors [30, 21]. One version of the recovery problem can be formulated as

$$\text{minimize} \quad \| |Ax|^2 - y^2 \|_1,$$

with variable $x \in \mathbf{C}^n$. Here, $A \in \mathbf{C}^{m \times n}$ has rows a_k^H, and the absolute value and square operations are applied elementwise. Since $|Ax|$ is the same if all entries of x are multiplied by a complex number with unit magnitude, we can only recover x up to some constant phase shift.

Our current DNLP extension of CVXPY does not support complex variables, but we can manually reformulate the problem in terms of the real variable $\tilde{x} = (\Re(x), \Im(x)) \in \mathbf{R}^{2n}$ as

$$\text{minimize} \quad \|(B\tilde{x})^2 + (C\tilde{x})^2 - y^2\|_1,$$

where the problem data are

$$B = \begin{bmatrix} \Re(A) & -\Im(A) \end{bmatrix} \in \mathbf{R}^{m \times 2n}, \qquad C = \begin{bmatrix} \Im(A) & \Re(A) \end{bmatrix} \in \mathbf{R}^{m \times 2n}.$$

(Here $\Re(\cdot)$ and $\Im(\cdot)$ denote the real and imaginary parts, respectively.)

DNLP specification. The code specifying this problem is given below.

```
x_tilde = Variable(2 * n)
cost = norm1((B @ x_tilde) ** 2 + (C @ x_tilde) ** 2 - y ** 2)
prob = Problem(Minimize(cost))
prob.solve(nlp=True)
```

Results. We consider a problem instance with $n = 64$ and $m = 3n$. The real and imaginary part of each entry of the true signal and the measurement vectors are sampled uniformly from the unit interval. Figure 7 shows the original and recovered signals. We see that the signal is accurately recovered (up to a phase shift).

Figure 7: Phase retrieval.

5.6 Sparse signal recovery

Problem. The goal is to recover a sparse signal $x_0 \in \mathbf{R}^n$ from a given measurement vector $y = Ax_0$, where $A \in \mathbf{R}^{m \times n}$ (with $m < n$) is a known sensing matrix [22]. A common heuristic based on convex optimization is to minimize the ℓ_1 norm of x subject to $Ax = y$. An alternative approach based on nonconvex optimization is to minimize the sum of the square roots of the absolute values of the entries of x, which tends to promote sparsity more aggressively [23]. This leads to the problem

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^n \sqrt{|x_i|} \\ \text{subject to} & Ax = y, \end{array}$$

with variable x. This problem is DNLP-compliant since the objective is L-convex.

DNLP specification. The code specifying this problem is given below. For this example, we use Knitro's interior-point method as the solver, because Ipopt failed to solve this problem reliably. The issue likely arises from the fact that the objective function gradient becomes infinite as any entry of x approaches zero, so no KKT point exists for the canonicalized problem.

```
x = Variable(n)
cost, constr = sum(sqrt(abs(x))), [A @ x == y]
prob = Problem(Minimize(cost), constr)
prob.solve(nlp=True, solver='knitro_ipm')
```

Problem instances. We consider a simulation with signal dimension $n = 100$, where we vary the number of measurements m from 60 to 80, and the cardinality of the true signal x_0 from 30 to 50. The positions of the nonzero entries of x_0 are sampled from a uniform distribution, with the nonzero values chosen as $\mathcal{N}(0, 25)$ random variables. The entries of A are sampled from a standard normal distribution. We say that the recovery is successful if the relative error $\|\hat{x} - x_0\|_2 / \|x_0\|_2$ is less than 10^{-2}, where $\hat{x}$ is the recovered signal. To estimate the probability of successful recovery for each pair of number of measurements and signal cardinality, we repeat the simulation 100 times and compute the fraction of successful recoveries.

Results. Figure 8 shows a heatmap of the estimated probability of successful signal recovery. We see that the nonconvex approach is more effective than the convex approach.

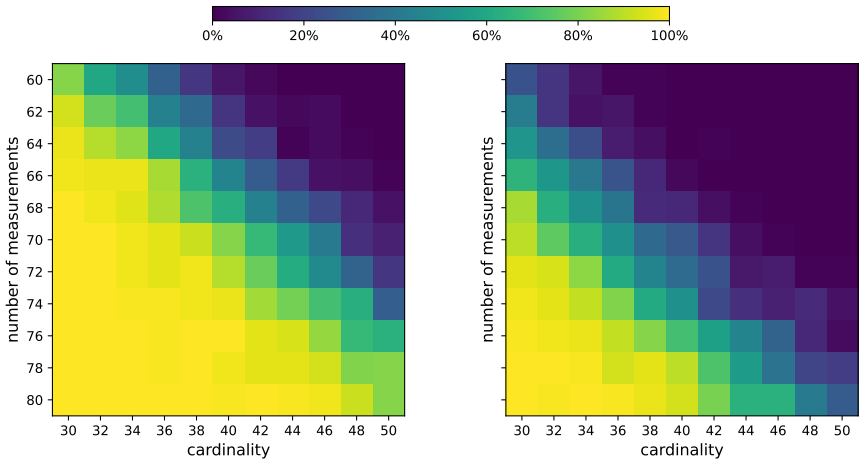

Figure 8: Probability of successful signal recovery. *Left.* Approach based on nonconvex optimization. *Right.* Approach based on convex optimization.

5.7 Nonlinear optimal control

Problem. We consider a simple model of a car in $\mathbf{R}^2$ as described in [15, §19.4]. After time discretization with step size $h > 0$, the state is $x_k \in \mathbf{R}^3$, with $((x_k)_1, (x_k)_2)$ denoting its position at time $t = kh$, and $(x_k)_3$ denoting its angle or orientation. The control input, which we choose, is $u_k \in \mathbf{R}^2$, where $(u_k)_1$ is the speed and $(u_k)_2$ is the steering angle over the time interval $t \in [kh, (k+1)h]$. The goal is to choose inputs u_k for $k = 0, \ldots, N-1$ to move the car from a given initial state x^{init} to a given final state x^{final}.

The car dynamics are given by $x_{k+1} = f(x_k, u_k)$, where

$$f(x_k, u_k) = x_k + (u_k)_1 h \begin{bmatrix} \cos(x_k)_3 \\ \sin(x_k)_3 \\ (\tan(u_k)_2)/L \end{bmatrix}$$

and $L > 0$ is the wheelbase length of the car. We are given limits $a_{\max}$ and $\omega_{\max}$ on the acceleration and steering angle rate, expressed as $|(u_{k+1})_1 - (u_k)_1| \leq a_{\max} h$ and $|(u_{k+1})_2 - (u_k)_2| \leq \omega_{\max} h$. We also have lower and upper limits $s_{\min} \leq (u_k)_1 \leq s_{\max}$ and $\phi_{\min} \leq (u_k)_2 \leq \phi_{\max}$ on the speed and steering angle. We want the control input to be small and smooth, so as objective we take the sum of the squared Euclidean norms of the control input over all time steps plus a term that penalizes rapid changes, weighted by $\gamma > 0$. This gives us the problem

$$\begin{array}{lll} \text{minimize} & \sum_{k=0}^{N-1} \|u_k\|_2^2 + \gamma \sum_{k=0}^{N-2} \|u_{k+1} - u_k\|_2^2 & \\ \text{subject to} & x_{k+1} = f(x_k, u_k), & k = 0, \ldots, N-1 \\ & x_0 = x^{\text{init}}, \quad x_N = x^{\text{final}} & \\ & |(u_{k+1})_1 - (u_k)_1| \leq a_{\max} h, & k = 0, \ldots, N-2 \\ & |(u_{k+1})_2 - (u_k)_2| \leq \omega_{\max} h, & k = 0, \ldots, N-2 \\ & s_{\min} \leq (u_k)_1 \leq s_{\max}, & k = 0, \ldots, N-1 \\ & \phi_{\min} \leq (u_k)_2 \leq \phi_{\max}, & k = 0, \ldots, N-1, \end{array}$$

with variables $x_0, \ldots, x_N$ and $u_0, \ldots, u_{N-1}$. The problem data are h, L, $a_{\max}$, $\omega_{\max}$, $s_{\min}$, $s_{\max}$, $\phi_{\min}$, $\phi_{\max}$, γ, and the initial and final states x^{init}, x^{final}.

DNLP specification. The code specifying this problem is given below.

```
x, u = Variable((N+1, 3)), Variable((N, 2))
cost = sum_squares(u) + gamma * sum_squares(u[1:, :] - u[:-1, :])
constr = [x[0, :] == x_init, x[N, :] == x_final]
constr += [x[1:, :] == x[:-1, :] + h * hstack([
```

```
                multiply(u[:, 0], cos(x[:-1, 2])),
                multiply(u[:, 0], sin(x[:-1, 2])),
                multiply(u[:, 0], tan(u[:, 1]) / L)])]
constr += [abs(u[1:, 0] - u[:-1, 0]) <= a_max * h,
           abs(u[1:, 1] - u[:-1, 1]) <= omega_max * h]
constr += [s_min <= u[:, 0], u[:, 0] <= s_max,
           phi_min <= u[:, 1], u[:, 1] <= phi_max]
prob = Problem(Minimize(cost), constr)
prob.solve(nlp=True)
```

Problem instance. We consider a problem instance where the car starts at the origin with zero orientation, meaning that it is facing right. The final state is $(0.5, 0.5, -\pi/2)$, *i.e.*, the car should end up half a unit above and to the right of its starting position, facing down. We use the parameters $L = 0.1$, $N = 50$, $h = 0.1$, and $\gamma = 10$. The acceleration and steering rate limits are given as $a_{\max} = 0.35$ and $\omega_{\max} = \pi/10$, and the speed and steering angle limits are $s_{\min} = -0.15$, $s_{\max} = 0.6$, $\phi_{\min} = -\pi/8$, and $\phi_{\max} = \pi/8$.

Results. Figure 9 shows the trajectory of the car, together with the speed, steering angle, and their rates of change. We see that the steering angle is initially positive, causing the car to turn left, and then negative, causing it to turn right, before finally straightening out to reach the target position facing down.

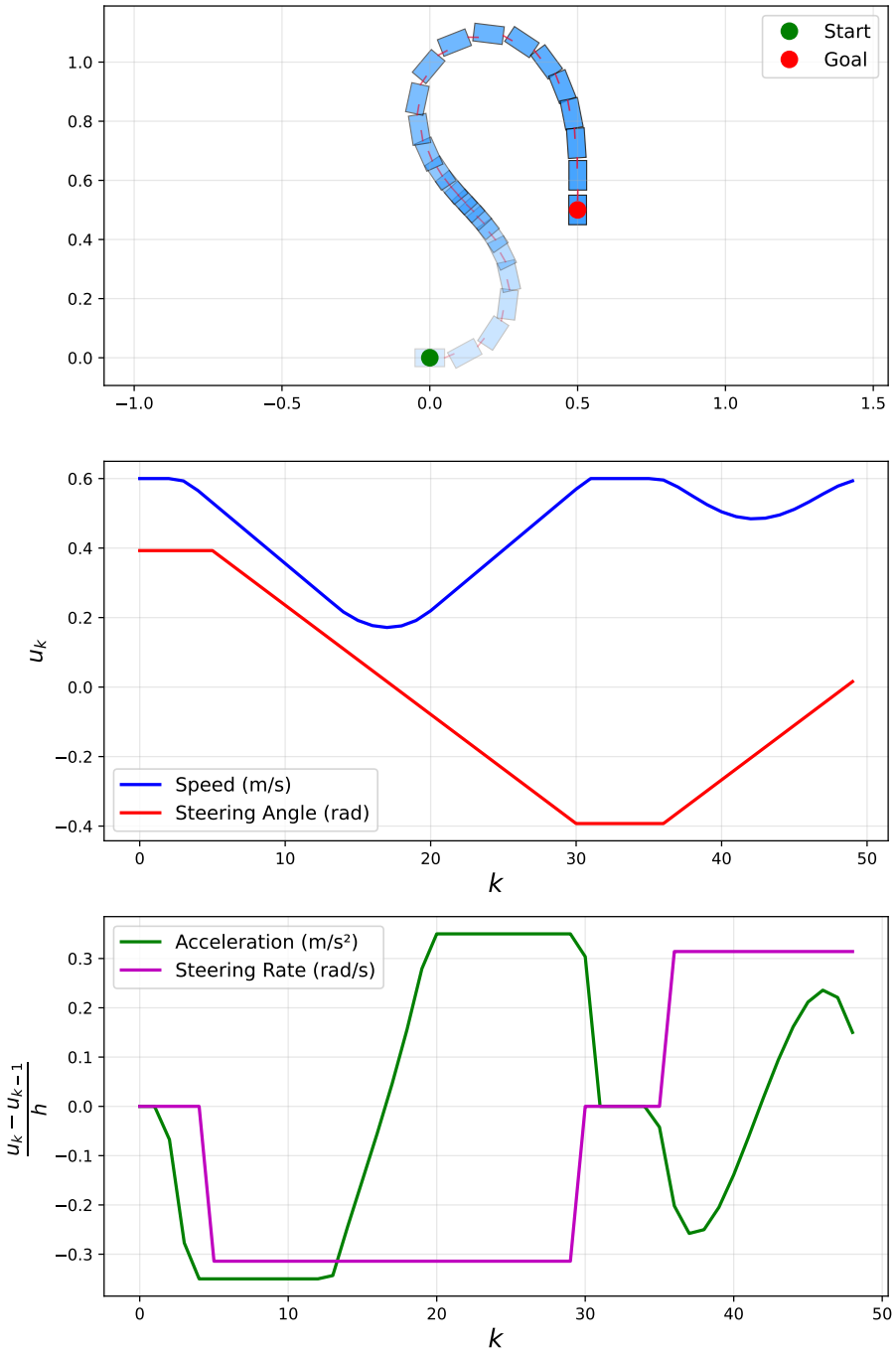

Figure 9: Car trajectory. *Top.* Position and orientation of the car. *Middle.* Speed and steering angle. *Bottom.* Acceleration and steering rate.

5.8 Trimmed logistic regression

Problem. We are given a data set $x_i \in \mathbf{R}^d$, $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$. We seek a prediction model $\hat{y} = \mathbf{sign}(\theta^T x)$, where $\theta \in \mathbf{R}^d$ is the model parameter. In logistic regression, we choose θ to minimize the logistic loss

$$\sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^T x_i)).$$

In trimmed logistic regression [45, 6], we introduce an auxiliary weight $w_i \in [0, 1]$ for each data point, allowing the predictor to downweight outliers and potentially corrupted data points. The parameter θ is found by solving

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} w_i \log(1 + \exp(-y_i \theta^T x_i)) \\ \text{subject to} & \mathbf{1}^T w = k, \\ & 0 \leq w_i \leq 1, \quad i = 1, \ldots, n, \end{array}$$

with variables $\theta \in \mathbf{R}^d$ and $w \in \mathbf{R}^n$. Here, $k \in (0, n)$ is a given parameter that specifies the effective number of samples retained in the fit.

DNLP specification. The code specifying the trimmed logistic regression problem is given below.

```
theta = Variable(d)
w = Variable(n, bounds=[0, 1])
loss = sum(multiply(w, logistic(-multiply(y, X @ theta))))
constr = [sum(w) == k]
prob = Problem(Minimize(loss), constr)
prob.solve(nlp=True)
```

Problem instance. We consider the task of classifying handwritten digits 0 and 1 from the MNIST data set [52]. From the full data set, we randomly select $n = 2000$ images of digits 0 and 1 for training, where each image is represented by $d = 785$ features (the 784 pixel intensities together with an additional bias term). First, we fit a standard logistic regression model on the clean training data. We then adversarially corrupt 1% of the samples by flipping their labels and refit the standard logistic regression model on this corrupted data. Finally, we fit a trimmed logistic regression model on the corrupted data using $k = 0.95n$. To compare the peformance of the different models, we evaluate their accuracy on a separate test set of 2000 images of digits 0 and 1.

Results. The standard logistic regression model achieves a test accuracy of 99.1% when fitted on the clean training data and 89.3% when fitted on the corrupted data. In contrast, the trimmed logistic regression model achieves a test accuracy of 98.4% when fitted on the corrupted data. The weights assigned to the corrupted training samples are zero, indicating that the trimmed logistic regression model successfully identified and ignored the corrupted samples.

5.9 Risk-budgeted portfolio construction

Problem. Risk-budgeted portfolio construction aims to build a portfolio in which different sectors contribute specified proportions to the total portfolio risk [63, 28]. We consider a portfolio of n assets grouped into K sectors, where $\mathcal{G}_k$ is the set of asset indices in sector k. We let $w_i \geq 0$ denote the fraction of the total portfolio value (assumed positive) invested in asset i. The total portfolio risk is the standard deviation of the portfolio return $\sigma = (w^T \Sigma w)^{1/2}$, where $\Sigma \in \mathbf{S}^n_{++}$ is the asset return covariance matrix. We can decompose the risk σ into components σ_k attributable to the sectors as

$$\sigma = \frac{w^T \Sigma w}{(w^T \Sigma w)^{1/2}} = \sum_{k=1}^{K} \sum_{i \in \mathcal{G}_k} \frac{w_i (\Sigma w)_i}{(w^T \Sigma w)^{1/2}} = \sum_{k=1}^{K} \sigma_k,$$

with

$$\sigma_k = \sum_{i \in \mathcal{G}_k} \frac{w_i (\Sigma w)_i}{(w^T \Sigma w)^{1/2}}.$$

The risk-adjusted return of the portfolio is given by $\mu^T w - \lambda w^T \Sigma w$, where μ is the asset return mean, and $\lambda > 0$ is a given risk aversion parameter.

In risk-budgeted portfolio construction, we seek portfolio weights w that maximize risk-adjusted return subject to sector risks being close to given proportions $b_k \in (0, 1)$ of the total portfolio risk, *i.e.*, $\sigma_k \approx b_k \sigma$ for $k = 1, \ldots, K$. With a 10% tolerance for sector risk targets, this can be written as the problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \lambda w^T \Sigma w \\ \text{subject to} & |\sum_{i \in \mathcal{G}_k} w_i (\Sigma w)_i - b_k w^T \Sigma w| \leq 0.1 b_k w^T \Sigma w, \quad k = 1, \ldots, K \\ & \mathbf{1}^T w = 1, \quad w \geq 0, \end{array}$$

with variable $w \in \mathbf{R}^n$.

DNLP specification. The code specifying this problem is given below. For further efficiency we have introduced two auxiliary variables t_1 and t_2 to represent the subexpressions Σw and $w^T \Sigma w$ that appear multiple times in the formulation.

```
w, t1, t2 = Variable((n, ), nonneg=True), Variable((n, )), Variable()
obj = mu.T @ w - lmbda * t2
constr = [sum(w) == 1, t1 == Sigma @ w, t2 == quad_form(w, Sigma)]
for k, g in enumerate(groups):
    constr += [abs(sum(multiply(w[g], t1[g])) - b[k] * t2)
                                          <= 0.1 * b[k] * t2]
```

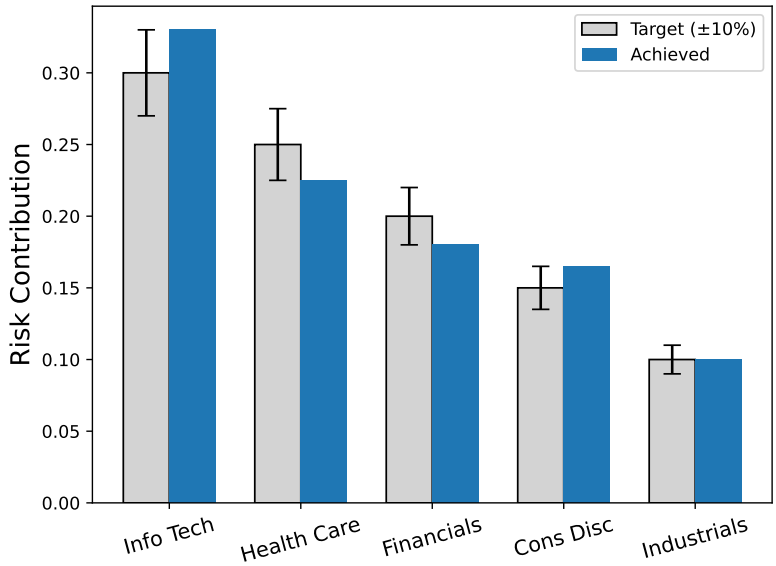

Figure 10: Sector risk contributions of the risk-budgeted portfolio.

```
w.value = np.ones(n) / n # uniform initial guess
prob = Problem(Maximize(obj), constr)
prob.solve(nlp=True)
```

Problem instance. We consider a problem instance with $n = 319$ assets from S&P 500 grouped into the $K = 5$ largest sectors according to the Global Industry Classification Standard (GICS), which are Information Technology, Health Care, Financials, Consumer Discretionary, and Communication Services. The risk budgets are set to $b = (0.3, 0.25, 0.20, 0.15, 0.10)$, allocating approximately 30% of portfolio risk to Information Technology, with the remaining sectors contributing approximately 25%, 20%, 15%, and 10%, respectively. We set the covariance matrix and asset return mean to the sample covariance and empirical mean of the asset returns, respectively, over the period from January 1, 2020 to January 1, 2025. (Of course, in practice one would use sophisticated methods to estimate these.)

Results. Figure 10 shows the sector risk contributions of the optimized portfolio. Two of them take on the highest allowed risk, two take on the smallest allowed risk, and one is in between the sector risk limits.

5.10 Optimal power flow

Problem. The optimal power flow problem seeks to optimize the operation of an electric power system subject to network power flow constraints and system operating limits [34]. We use a standard model, described by a graph with n buses (nodes), where each bus i is characterized by a voltage magnitude v_i and a phase angle θ_i. The real and reactive power injected at the buses are denoted by $p \in \mathbf{R}^n$ and $q \in \mathbf{R}^n$, respectively. These are related to the voltage magnitudes and phase angles via the equations $p = P\mathbf{1}$ and $q = Q\mathbf{1}$, where the bus injection matrices $P \in \mathbf{R}^{n\times n}$ and $Q \in \mathbf{R}^{n\times n}$ are given by

$$\begin{aligned} P &= (vv^T) \circ (G \circ C(\theta) + B \circ S(\theta)) \\ Q &= (vv^T) \circ (G \circ S(\theta) - B \circ C(\theta)). \end{aligned} \tag{4}$$

Here, $G \in \mathbf{S}^n$ and $B \in \mathbf{S}^n$ are the (given) real and imaginary parts of the admittance matrix of the network, $C(\theta) \in \mathbf{S}^n$ and $S(\theta) \in \mathbf{R}^{n\times n}$ are defined as

$$C_{ij}(\theta) = \cos(\theta_i - \theta_j), \qquad S_{ij}(\theta) = \sin(\theta_i - \theta_j),$$

and $\circ$ denotes the elementwise (Hadamard) product. Physical limitations of the network components requires that the power flows and voltages satisfy certain operational constraints, such as bounds

$$v^{\min} \leq v \leq v^{\max}, \quad p^{\min} \leq p \leq p^{\max}, \quad q^{\min} \leq q \leq q^{\max}. \tag{5}$$

(The bounds on p and q can be used to model generation limits at generator buses and load demands at load buses.) To fix the reference angle of the network, we force the phase angle at the first bus to be zero. The total generation cost is typically a convex quadratic function $f(p)$ of the real power generated at each bus. The optimal power flow problem can thus be formulated as

$$\begin{array}{ll} \text{minimize} & f(p) \\ \text{subject to} & P = (vv^T) \circ (G \circ C(\theta) + B \circ S(\theta)) \\ & Q = (vv^T) \circ (G \circ S(\theta) - B \circ C(\theta)) \\ & p = P\mathbf{1}, \quad q = Q\mathbf{1}, \quad \theta_1 = 0 \\ & v^{\min} \leq v \leq v^{\max}, \quad p^{\min} \leq p \leq p^{\max}, \quad q^{\min} \leq q \leq q^{\max}, \end{array}$$

with variables v, θ, P, Q, p, and q.

DNLP specification. The code specifying this problem is given below.

```
theta, P, Q = Variable((N, 1)), Variable((N, N)), Variable((N, N))
v = Variable((N, 1), bounds=[v_min, v_max])
p = Variable(N, bounds=[p_min, p_max])
q = Variable(N, bounds=[q_min, q_max])
C, S = cos(theta - theta.T), sin(theta - theta.T)
constr = [theta[0] == 0, p == sum(P, axis=1), q == sum(Q, axis=1),
          P == multiply(v @ v.T, multiply(G, C) + multiply(B, S)),
          Q == multiply(v @ v.T, multiply(G, S) - multiply(B, C))]
cost = ... # some cost function
prob = Problem(Minimize(cost), constr)
prob.solve(nlp=True)
```

Alternative DNLP specification. The code above declares the bus injection matrices P and Q as dense matrices, and uses the power flow equations (4) to incorporate the sparsity pattern of the network only via the admittance matrices G and B. We can also use the variable attribute `sparsity` to explicitly define P and Q as sparse matrices. If E is the set of edges in the network, we do this by declaring P and Q as

```
P = Variable((N, N), sparsity=E)
Q = Variable((N, N), sparsity=E).
```

This alternative approach is more efficient for large networks.

Results. We consider a 9-node network from [17] with 3 generator buses (green squares), 3 transmission buses (blue circles), and 3 load buses (orange diamonds). Figure 11 shows the optimized real power flow. Each directed edge is annotated with the real power flowing into the bus at the arrowhead, and, in parentheses, the corresponding real-power loss on that line. (The real power flow on each line (i, j) is given by $P_{ij}^{\text{flow}} = v_i^2 G_{ij} - P_{ij}$, with the convention that positive flow is toward bus j. The loss of real power on line (i, j) is given by $L_{ij} = P_{ij}^{\text{flow}} + P_{ji}^{\text{flow}}$.) The total generation cost for the computed flow is \$3087.4, which is known to be the global solution [50, table 15.2].

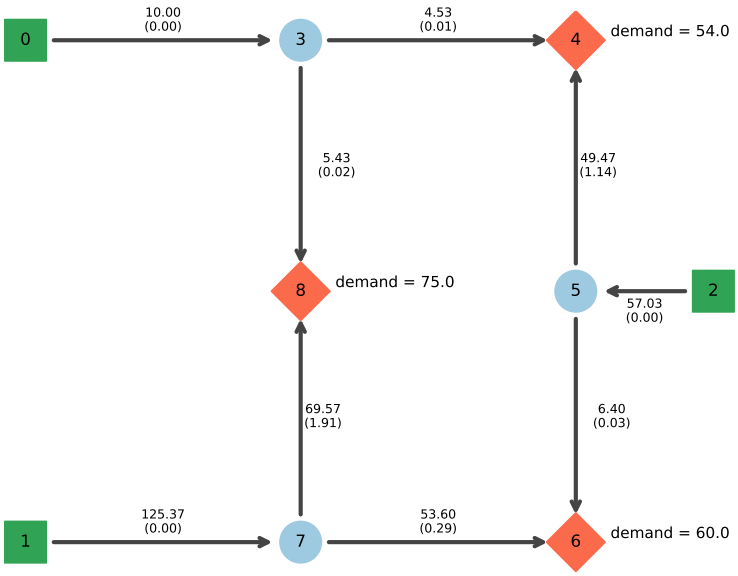

Figure 11: Optimal power flow.

Acknowledgments

We acknowledge many helpful discussions with several colleagues, including Bennet Meyers, Antoine Lesage-Landry, Bartolomeo Stellato, Clara Baynham, Anthony Degleris, Maximilian Schaller, Art Owen, Michael Salerno, and Aleksandr Aravkin. We want to thank Amit Solomon for developing cvxtorch, which was used in an early prototype of this work. We also want to thank Steven Diamond for helpful comments and suggestions on the NLP interface and Youssouf Emine for guidance in implementing the Knitro interface.

William Zhang was supported by Maxime Fortin through the Polytechnique Montreal Bourse Prestige scholarship. William Zhang est financé par Maxime Fortin grâce à la Bourse Prestige de Polytechnique Montréal.

References

[1] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[2] J. Andersson, J. Gillis, G. Horn, J. Rawlings, and M. Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[3] R. Andreani, E. Birgin, J. Martínez, and M. Schuverdt. On Augmented Lagrangian Methods with General Lower-Level Constraints. *SIAM Journal on Optimization*, 18(4):1286–1309, 2008.

[4] M. ApS. MOSEK modeling cookbook, 2025.

[5] M. ApS. *MOSEK optimization suite 11.0*, 2025.

[6] A. Aravkin and D. Davis. Trimmed Statistical Estimation via Variance Reduction. *Mathematics of Operations Research*, 45(1):292–322, 2020.

[7] Artelys. *Release Notes for KNITRO 15.0*. Artelys, 2024. Knitro 15.0 offers a new Augmented Lagrangian (AL) algorithm for nonlinear programs.

[8] A. Beck, P. Stoica, and J. Li. Exact and Approximate Solutions of Source Localization Problems. *IEEE Transactions on signal processing*, 56(5):1770–1778, 2008.

[9] M. Bendsoe and S. Sigmund. *Topology Optimization: Theory, Methods and Applications*. Springer, 2004.

[10] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 3rd edition, 2016.

[11] J. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM, 2010.

[12] L. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization. SIAM, Philadelphia, PA, 2010.

[13] J. Bisschop. *AIMMS optimization modeling*. 2006.

[14] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[15] S. Boyd and L. Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge university press, 2018.

[16] A. Brook, D. Kendrick, and A. Meeraus. GAMS, a user's guide. *SIGNUM Newsletter*, 23(3–4):10–11, 1988.

[17] W. Bukhsh, A. Grothey, K. McKinnon, and P. Trodden. Local solutions of the optimal power flow problem. *IEEE Transactions on Power Systems*, 28(4):4780–4788, 2013.

[18] C. Büskens and D. Wassel. The ESA NLP Solver WORHP. In *Modeling and Optimization in Space Engineering*, pages 85–110. Springer, 2012.

[19] M. Bynum, G. Hackebeil, W. Hart, C. Laird, B. Nicholson, J. Siirola, J.-P. Watson, and D. Woodruff. *Pyomo—Optimization Modeling in Python*, volume 67 of *Springer Optimization and Its Applications*. Springer, Cham, 3 edition, 2021.

[20] R. Byrd, M. Hribar, and J. Nocedal. An Interior Point Algorithm for Large-Scale Nonlinear Programming. *SIAM Journal on Optimization*, 16(5):1190–1208, 2006.

[21] E. Candès, X. Li, and M. Soltanolkotabi. Phase retrieval via Wirtinger flow: Theory and algorithms. *IEEE Transactions on Information Theory*, 61(4):1985–2007, 2015.

[22] E. Candès and M. Wakin. An Introduction to Compressive Sampling. *IEEE signal processing magazine*, 25(2):21–30, 2008.

[23] R. Chartrand. Exact Reconstruction of Sparse Signals via Nonconvex Minimization. *IEEE Signal Processing Letters*, 14(10):707–710, 2007.

[24] A. Conn, N. Gould, and P. Toint. *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, volume 17 of *Springer Series in Computational Mathematics*. Springer-Verlag, Heidelberg, Berlin, New York, 1992.

[25] F. Curtis, T. Mitchell, and M. Overton. A BFGS-SQP method for nonsmooth, nonconvex, constrained optimization and its evaluation using relative minimization profiles. *Optimization Methods and Software*, 32(1):148–181, 2017.

[26] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[27] I. Dunning, J. Huchette, and M. Lubin. JuMP: A Modeling Language for Mathematical Optimization. *SIAM review*, 59(2):295–320, 2017.

[28] Y. Feng and D. Palomar. SCRIP: Successive Convex Optimization Methods for Risk Parity Portfolio Design. *IEEE Transactions on Signal Processing*, 63(19):5285–5300, 2015.

[29] A. Fiacco and G. McCormick. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. Classics in Applied Mathematics. John Wiley & Sons, New York, 1968. Reprinted by SIAM, 1990.

[30] J. Fienup. Phase Retrieval Algorithms: A Comparison. *Applied optics*, 21(15):2758–2769, 1982.

[31] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2000.

[32] A. Forsgren, P. Gill, and M. Wright. Interior methods for nonlinear optimization. *SIAM review*, 44(4):525–597, 2002.

[33] R. Fourer, D. Gay, and B. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36(5):519–554, 1990.

[34] S. Frank and S. Rebennack. An Introduction to Optimal Power Flow: Theory, Formulation, and Examples. *IIE Transactions*, 48(12):1172–1197, 2016.

[35] P. Gill, W. Murray, and M. Saunders. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Journal on Optimization*, 12(4):979–1006, 2005.

[36] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. SIAM, 2019.

[37] P. Gill, M. Saunders, and E. Wong. On the Performance of SQP Methods for Nonlinear Optimization. *Modeling and optimization: theory and applications*, pages 95–123, 2015.

[38] N. Gillis. *Nonnegative Matrix Factorization*. SIAM, 2020.

[39] G. Giorgi and T. Kjeldsen. *Traces and Emergence of Nonlinear Programming*. Springer Science & Business Media, 2013.

[40] M. Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, Dec. 2004. Ph.D. dissertation.

[41] M. Grant and S. Boyd. Graph Implementations for Nonsmooth Convex Programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, volume 371 of *Lecture Notes in Control and Information Sciences*. Springer, London, 2008.

[42] M. Grant, S. Boyd, and Y. Ye. Disciplined Convex Programming. In *Global optimization: From theory to implementation*, pages 155–210. Springer, 2006.

[43] A. Griewank and A. Walther. *Evaluating Derivatives*. SIAM, 2008.

[44] Gurobi Optimization, LLC. Gurobi Optimizer Release Notes, version 13, 2025.

[45] A. Hadi and A. Luceño. Maximum Trimmed Likelihood Estimators: A Unified Approach, Examples, and Algorithms. *Computational Statistics & Data Analysis*, 25(3):251–272, 1997.

[46] W. Hart, J. Watson, and D. Woodruff. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[47] M. Hifi and R. Mallah. A Literature Review on Circle and Sphere Packing Problems: Models and Methodologies. *Advances in Operations Research*, 2009(1):150624, 2009.

[48] A. Izmailov, M. Solodov, and E. Uskov. Global Convergence of Augmented Lagrangian Methods Applied to Optimization Problems With Degenerate Constraints, Including Problems With Complementarity Constraints. *SIAM Journal on Optimization*, 22(4):1579–1606, 2012.

[49] A. Keane and P. Nair. *Computational Approaches for Aerospace Design: the Pursuit of Excellence*. John Wiley & Sons, 2005.

[50] V. Krasko and S. Rebennack. Chapter 15: Global optimization: Optimal power flow problem. In *Advances and trends in optimization with engineering applications*, pages 187–205. SIAM, 2017.

[51] J. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[52] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[53] D. Lee and H. Seung. Learning the Parts of Objects by Non-Negative Matrix Factorization. *Nature*, 401(6755):788–791, 1999.

[54] D. Lee and H. Seung. Algorithms for Non-Negative Matrix Factorization. *Advances in Neural Information Processing Systems*, 2000.

[55] B. Liang, T. Mitchell, and J. Sun. NCVX: A General-Purpose Optimization Solver for Constrained Machine and Deep Learning. *arXiv preprint arXiv:2210.00973*, 2022.

[56] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *2004 IEEE international conference on robotics and automation (IEEE Cat. No. 04CH37508)*, pages 284–289. IEEE, 2004.

[57] W. Murray. Analytical Expressions for The Eigenvalues and Eigenvectors of the Hessian Matrices of Barrier and Penalty Functions. *Journal of Optimization Theory and Applications*, 7(3):189–196, 1971.

[58] B. Murtagh and M. Saunders. *MINOS 5.5 User's Guide*. Technical Report SOL 83-20R. 1998 (revised).

[59] Y. Nesterov and A. Nemirovskii. *Interior-point Polynomial Algorithms in Convex Programming*. SIAM, 1994.

[60] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 2006.

[61] P. Papalambros and D. Wilde. *Principles of Optimal Design: Modeling and Computation*. Cambridge University Press, Cambridge, 3rd edition, 2017.

[62] H. Reiner and T. Hoang. *Global Optimization: Deterministic Approaches*. Springer, Berlin, Heidelberg, 1996.

[63] T. Roncalli. *Introduction to Risk Parity and Budgeting*. CRC press, 2013.

[64] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel. Motion Planning with Sequential Convex Optimization and Convex Collision Checking. *The International Journal of Robotics Research*, 33(9):1251–1270, 2014.

[65] E. Smith. *On the Optimal Design of Continuous Processes*. PhD thesis, Imperial College London (University of London), London, 1996. Unpublished doctoral dissertation.

[66] E. Smith and C. Pantelides. *Global Optimisation of General Process Models*, pages 355–386. Springer US, Boston, MA, 1996.

[67] J. Smith and J. Abel. Closed-Form Least-Squares Source Location Estimation from Range-Difference Measurements. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(12):1661–1669, 1987.

[68] E. Specht. Packomania. http://www.packomania.com/, 2013.

[69] D. Thierry and L. Biegler. The ℓ_1—Exact Penalty-Barrier Phase for Degenerate Nonlinear Programming Problems in Ipopt. *IFAC-PapersOnLine*, 53(2):6496–6501, 2020. 21st IFAC World Congress.

[70] C. Vanaret and S. Leyffer. Implementing a Unified Solver for Nonlinearly Constrained Optimization. *Preprint, available at https://www.researchgate.net/profile/Charlie-Vanaret/research*, 2025.

[71] R. Vanderbei and D. Shanno. An Interior-Point Algorithm for Nonconvex Nonlinear Programming. *Computational Optimization and Applications*, 13(1):231–252, 1999.

[72] A. Wächter and L. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[73] R. Wilson. *A Simplicial Algorithm for Concave Programming*. PhD thesis, Graduate School of Business Administration, Harvard University, Cambridge, MA, 1963. Ph.D. thesis.

[74] M. Wright. Ill-Conditioning and Computational Error in Interior Methods for Nonlinear Programming. *SIAM Journal on Optimization*, 9(1):84–111, 1998.