Disciplined Convex-Concave Programming

Xinyue Shen1  Steven Diamond2  Yuantao Gu1  Stephen Boyd2

Abstract—In this paper we introduce disciplined convex-concave programming (DCCP), which combines the ideas of disciplined convex programming (DCP) with convex-concave programming (CCP). Convex-concave programming is an organized heuristic for solving nonconvex problems that involve objective and constraint functions that are a sum of a convex and a concave term. DCP is a structured way to define convex optimization problems, based on a family of basic convex and concave functions and a few rules for combining them. Problems expressed using DCP can be automatically converted to standard form and solved by a generic solver; widely used implementations include CVXPY, CVX, CVXPy, and Convex.jl. In this paper we propose a framework that combines the two ideas, and includes two improvements over previously published work on convex-concave programming, specifically the handling of domains of the functions, and the issue of subdifferentiability on the boundary of the domains. We describe a Python implementation called DCCP, which extends CVXPy, and give examples.

I. DISCIPLINED CONVEX-CONCAVE PROGRAMMING

A. Difference of convex programming

Difference of convex (DC) problems have the form

\[
\begin{align*}
\text{minimize} & \quad f_0(x) - g_0(x) \\
\text{subject to} & \quad f_i(x) - g_i(x) \leq 0, \quad i = 1, \ldots, m,
\end{align*}
\]

where \( x \in \mathbb{R}^n \) is the optimization variable, and the functions \( f_i : \mathbb{R}^n \to \mathbb{R} \) and \( g_i : \mathbb{R}^n \to \mathbb{R} \) for \( i = 0, \ldots, m \) are convex. The DC problem (1) can also include equality constraints of the form \( p_i(x) = q_i(x) \), where \( p_i \) and \( q_i \) are convex; we simply express these as the pair of inequality constraints

\[
p_i(x) - q_i(x) \leq 0, \quad q_i(x) - p_i(x) \leq 0,
\]

which have the difference of convex form in (1). When the functions \( g_i \) are all affine, the problem (1) is a convex optimization problem, and easily solved [1].

The broad class of DC functions includes all \( C^2 \) functions [2], so the DC problem (1) is very general. A special case is Boolean linear programs, which can represent many problems, such as the traveling salesman problem, that are widely believed to be hard to solve [3]. DC programs arise in many applications in fields such as image processing [4], machine learning [5], [6], [7], [8], computer vision [9], control [10], [11], discrete tomography [12], subgraph matching [13], and statistics [14]. As a more specific example, optimizing over the efficient set of a multiple objective problem can be formulated as a DC program [15].

DC problems can be solved globally by methods such as branch and bound [16], [17], [18], which can be slow in practice. Good overviews of solving DC programs globally can be found in [19], [20] and the references therein. A locally optimal (approximate) solution can be found instead through the many techniques of general nonlinear optimization [21].

The convex-concave procedure (CCP) [22], [23] is another heuristic algorithm for finding a local optimum of (1), which leverages our ability to efficiently solve convex optimization problems. In its basic form, it replaces concave terms with a convex upper bound, and then solves the resulting convex problem, which is a restriction of the original DC problem. Basic CCP can thus be viewed as an instance of majorization minimization (MM) algorithms [24], in which a minimization problem is approximated by an easier to solve upper bound created around the current point (a step called majorization) and then minimized. Many MM extensions have been developed over the years and more can be found in [25], [26], [27]. CCP can also be viewed as a version of DCA [28] which instead of explicitly linearizing the concave terms, solves a (convex) dual problem; see, e.g., [29] and the references therein.

A recent overview of CCP, with some extensions, can be found in [30], where the issue of infeasibility is handled (heuristically) by an increasing penalty on constraint violations. The method we present in this paper is an extension of the penalty CCP method introduced in [30], given as algorithm I.1 below.

Algorithm I.1 Penalty CCP.

given an initial point \( x_0, \tau_0 > 0, \tau_{\text{max}} > 0, \) and \( \mu > 1 \).

\( k := 0 \).

repeat

1. Convexify. For \( i = 0, \ldots, m \), form

\[
\tilde{g}_i(x; x_k) = g_i(x_k) + \nabla g_i(x_k)^T(x - x_k).
\]

2. Solve. Set the value of \( x_{k+1} \) to a solution of

\[
\begin{align*}
\text{minimize} & \quad f_0(x) - \tilde{g}_0(x; x_k) + \tau_k \sum_{i=1}^m s_i \\
\text{subject to} & \quad f_i(x) - \tilde{g}_i(x; x_k) \leq s_i, \quad i = 1, \ldots, m \\
& \quad s_i \geq 0, \quad i = 1, \ldots, m.
\end{align*}
\]

3. Update \( \tau \). \( \tau_{k+1} := \min(\mu \tau_k, \tau_{\text{max}}) \).

4. Update iteration. \( k := k + 1 \).

until stopping criterion is satisfied.

See [30] for discussion of a few variations on the penalty CCP algorithm, such as not using slack variables for constraints that are convex, i.e., the case when \( g_i \) is affine. Here it is assumed that \( g_i \) are differentiable, and have full domain (i.e., \( \mathbb{R}^n \)). The first condition is not critical; we can replace \( \nabla g_i(x_k) \) with a subgradient of \( g_i \) at \( x_k \), if it is not differentiable. The linearization with a subgradient instead of the gradient is still a lower bound on \( g_i \).

1Department of Electronic Engineering, Tsinghua University, Beijing 100084, China
2Department of Electrical Engineering, Stanford University, CA 94305, USA
In some practical applications, the second assumption, that \( g_i \) have full domain, does not hold, in which case the penalty CCP algorithm can fail, by arriving at a point \( x_k \) not in the domain of \( g_i \), so the convexification step fails. This is one of the issues we will address in this paper.

### B. Disciplined convex programming

Disciplined convex programming (DCP) is a methodology introduced by Grant et al. [31] that imposes a set of conventions that must be followed when constructing (or specifying or defining) convex programs. Conforming problems are called disciplined convex programs.

The conventions of DCP restrict the set of functions that can appear in a problem and the way functions can be composed. Every function in a disciplined convex program must come from a set of atomic functions with known curvature and graph implementation, or representation as partial optimization over a cone program [32], [33]. Every composition of functions \( f(g_1(x), \ldots, g_p(x)) \), where \( f : \mathbb{R}^p \to \mathbb{R} \) is convex and \( g_1, \ldots, g_p : \mathbb{R}^n \to \mathbb{R} \), must satisfy the following composition rule, which ensures the composition is convex. Let \( f' : \mathbb{R}^p \to \mathbb{R} \to \mathbb{R} \cup \{\infty\} \) be the extended-value extension of \( f \) [1, Chap. 3]. One of the following conditions must hold for each \( i = 1, \ldots, p \):

- \( g_i \) is convex and \( f' \) is nondecreasing in argument \( i \).
- \( g_i \) is concave and \( f' \) is nonincreasing in argument \( i \).
- \( g_i \) is affine.

The composition rule for concave functions is analogous.

A disciplined convex program has the specific form

\[
\begin{align*}
\text{minimize/maximize} & \quad o(x) \\
\text{subject to} & \quad l_i(x) \sim r_i(x), \quad i = 1, \ldots, m,
\end{align*}
\]

where \( o \) (the objective), \( l_i \) (lefthand sides), and \( r_i \) (righthand sides) are expressions (functions of the variable \( x \)) with curvature known from the DCP rules, and \( \sim \) denotes one of the relational operators:\( =, \leq, \text{ or } \geq \). In DCP this problem must be convex, which imposes conditions on the curvature of the expressions, listed below.

- For a minimization problem, \( o \) must be convex; for a maximization problem, \( o \) must be concave.
- When the relational operator is \( = \), \( l_i \) and \( r_i \) must both be affine.
- When the relational operator is \( \leq \), \( l_i \) must be convex, and \( r_i \) must be concave.
- When the relational operator is \( \geq \), \( l_i \) must be concave, and \( r_i \) must be convex.

Functions that are affine (i.e., are both convex and concave) can match either curvature requirement; for example, we can minimize or maximize an affine expression.

A disciplined convex program can be transformed into an equivalent cone program by replacing each function with its graph implementation. The convex optimization modeling systems YALMIP [34], CVX [35], CVXPY [36], and Convex.jl [37] use DCP to verify problem convexity and automatically convert convex programs into cone programs, which can then be solved using generic solvers.

### C. Disciplined convex-concave programming

We refer to a problem as a disciplined convex-concave program (DCCP) if it has the form (2), with \( o, l_i, \text{ and } r_i \) all having known DCP-verified curvature, but the DCP curvature conditions for the objective and constraints need not hold. Such problems include DCP as a special case, but it includes many other nonconvex problems as well. In DCCP we can, for example, maximize a convex function, subject to nonaffine equality constraints, and nonconvex inequality constraints between convex and concave expressions.

The general DC program (1) and the DCCP standard form (2) are equivalent. To express (1) as (2), we express it as

\[
\begin{align*}
\text{minimize} & \quad f_0(x) - t \\
\text{subject to} & \quad t = g_0(x), \quad g_i(x) \leq g_i(x), \quad i = 1, \ldots, m,
\end{align*}
\]

where \( x \) is the original optimization variable, and \( t \) is a new optimization variable. The objective here is convex. We have one (nonconvex) equality constraint, and the inequality constraints are all nonconvex (except for some special cases when \( g_i \) is affine). It is straightforward to express the DCCP form (2) in the form (1), by identifying the functions \( o, l_i, \text{ and } r_i \) as \( \pm f_i \) or \( \pm g_i \) depending on their curvatures.

DCCP is a convenient and simple standard form for DC programming, because the linearized problem in algorithm L1 is DCP whenever the original problem is DCCP. The linearized problem can thus be automatically converted into a cone program and solved using generic solvers.

### II. Domain and Subdifferentiability

In this section we delve deeper into an issue that is ‘assumed away’ in the standard treatments and discussions of DC programming, specifically, how to handle the case when the functions \( g_i \) do not have full domain. (The functions \( f_i \) can have non-full domains, but this is handled automatically by the conversion into a cone program.)

#### a) An example: Suppose the domain of \( g_i \) is \( D_i \), for \( i = 0, \ldots, m \). If \( D_i \neq \mathbb{R}^n \), simply defining the linearization \( \bar{g}_i(x; z) \) as the first order Taylor expansion of \( g_i \) at the point \( z \) can lead to failure. The following simple problem gives an example:

\[
\begin{align*}
\text{minimize} & \quad \sqrt{x} \\
\text{subject to} & \quad x \geq -1,
\end{align*}
\]

where \( x \in \mathbb{R} \) is the optimization variable. The objective has domain \( \mathbb{R}_+ \), and the solution is evidently \( x^* = 0 \). The linearized problem in the first iteration of CCP is

\[
\begin{align*}
\text{minimize} & \quad x_0 + \frac{1}{2\sqrt{x_0}}(x - x_0) \\
\text{subject to} & \quad x \geq -1,
\end{align*}
\]

which has solution \( x_1 = -1 \). The CCP algorithm will fail in the first step of the next iteration, since the original objective function is not defined at \( x_1 = -1 \).

If we add the domain constraint directly into the linearized problem, we obtain \( x_1 = 0 \), but the first step of the next iteration also fails here, in a different way. While \( x_1 \) is in the
domain of the objective function, the objective is not subdif-
ferentiable (or superdifferentiable) at \( x_1 \), so the linearization
does not exist. This phenomenon of non-subdifferentiability or
non-superdifferentiability can only occur at a point on the
boundary of the domain.

A. Linearization with domain

Suppose that the intersection of domains of all \( g_i \) in
problem (1) is \( D = \cap_{i=0}^m D_i \). The correct way to handle
the domain is to define the linearization of \( g_i \) at point \( z \) to
be
\[
\hat{g}_i(x; z) = g_i(z) + \nabla g_i(z)^T(x - z) - I_i(x),
\]
where the indicator function is
\[
I_i(x) = \begin{cases} 
0 & x \in D_i \\
\infty & x \notin D_i,
\end{cases}
\]
so any feasible point for the linearized problem is in the
domain \( D \).

Since \( g_i \) is convex, \( D_j \) is a convex set and \( I_i \) is a convex
function. Therefore the ‘linearization’ (3) is a concave func-
tion; it follows that if we replace the standard linearization
in algorithm 1.1 with the domain-restricted linearization (3), the
linearized problem is still convex.

B. Domain in DCCP

Recall that we defined DCCP to ensure that the linearized
problem in algorithm 1.1 is a DCP problem. It is not obvious
that if we replace the standard linearization with the domain
modified version (3), the linearized problem is still DCP. In
this section we prove that the linearized problem still satisfies
the rules of DCP, or equivalently that each \( I_i(x) \) has a known
graph implementation or satisfies the DCP composition rule.

If \( g_i \) is an atomic function, then we assume that
\[
D_i = \cap_{j=1}^p \{ x \mid A_j x + b_j \in K_j \},
\]
for some cone constraints \( K_1, \ldots, K_p \). The assumption is
reasonable since \( g_i \) itself can be represented as partial
optimization over a cone program. The graph implementation
of \( I_i(x) \) is simply
\[
\begin{align*}
\text{minimize} & \quad 0 \\
\text{subject to} & \quad A_j x + b_j \in K_j, \quad j = 1, \ldots, p.
\end{align*}
\]

The other possibility is that \( g_i \) is a composition of atomic
functions. Since the original problem is DCCP, we may
assume that \( g_i(x) = f(h_1(x), \ldots, h_p(x)) \) for some convex
atomic function \( f : R^n \to R \) and DCP compliant \( h_1, \ldots, h_p : R^n \to R \) such that \( f(h_1(x), \ldots, h_p(x)) \) satisfies the DCP
composition rule. Then we have
\[
I_i(x) = I_f(h_1(x), \ldots, h_p(x)) + \sum_{j=1}^p I_{h_j}(x),
\]
where \( I_f \) is the indicator function for the domain of \( f \) and
\( I_{h_1}, \ldots, I_{h_p} \) are defined similarly.

Since \( f \) is convex, \( I_f \) is convex. Moreover,
\( I_f(h_1(x), \ldots, h_p(x)) \) satisfies the DCP composition
rule. To see why, observe that for \( j = 1, \ldots, p \), if \( h_j \) is
convex then by assumption the extended-value extension
\( f \) is nondecreasing in argument \( j \). It follows that \( I_f \) is
nondecreasing in argument \( j \). Similarly, if \( h_j \) is concave
then \( I_f \) is nonincreasing in argument \( j \).

An inductive argument shows that \( I_{h_1}, \ldots, I_{h_p} \) are convex
and satisfy the DCP rules. We conclude that \( I_i \) satisfies the
DCP composition rule.

C. Chain rule for a subgradient

When linearizing a nondifferentiable convex function, we
need to calculate a subgradient at a certain point. (We
say ‘a’ subgradient since it is not in general unique.) The
gradient of a differentiable expression is readily computed
recursively by the chain rule. The chain rule for compositions
of functions does not generally apply for the calculation
of the subdifferential, but it does apply when we seek a
subgradient when the function conforms to the DCP rules.

Suppose that \( h(x) = f(g_1(x), \ldots, g_p(x)) \), where \( f \) is
convex, and the DCP rules hold, i.e., for each \( i \), \( g_i \) is
affine, or \( g_i \) is convex and \( f \) is nondecreasing in argument
\( i \), or \( g_i \) is concave and \( f \) is nonincreasing in argument
\( i \). We can compute a subgradient of \( h \) at \( x \) as follows.
Choose any \( q \in \partial f(g_1(x), \ldots, g_p(x)) \). By the monotonicity
assumption \( g_i \geq 0 \) when \( f_i \) is nondecreasing in argument
\( i \) and \( g_i \leq 0 \) when \( f_i \) is nonincreasing in argument \( i \).
Choose any \( s_i \in \partial g_i(x) \), \( i = 1, \ldots, p \). Then we have
\[
r = q_1 s_1 + \cdots + q_p s_p \in \partial h(x).
\]

This can be directly proved as follows. For any \( x, y \in R^n \),
\[
h(x) = f(g_1(x), \ldots, g_p(x)) \geq f(g_1(y), \ldots, g_p(y)) + \sum_{i=1}^p q_i (g_i(x) - g_i(y)) \geq f(g_1(y), \ldots, g_p(y)) + \sum_{i=1}^p q_i s_i^T (x - y) = h(y) + v^T (x - y).
\]

D. Sub-differentiability on boundary

When \( D \neq R^n \), a solution to the linearized problem \( \hat{x}_k \)
at iteration \( k \) can be on the boundary of the closure of
\( D \). It is possible (as our simple example above shows) that
the convex function \( g_i \) is not subdifferentiable at \( \hat{x}_k \),
which means the linearization does not exist and the algorithm
fails. This pathology can and does occur in practical problems.

In order to handle this, at each iteration, when the subgra-
dient \( \nabla g_i(\hat{x}_k) \) for any function \( g_i \) does not exist, we simply
take a damped step,
\[
x_k = \alpha \hat{x}_k + (1 - \alpha) x_{k-1},
\]
where \( 0 < \alpha < 1 \). If \( x_0 \) is in the interior of the domain, then
\( x_k \) will be in the interior for all \( k \geq 0 \), and \( \nabla g_i(x_k) \) will
be guaranteed to exist. The algorithm can (and does, for our
simple example) converge to a point on the boundary of the
the domain, but each iterate is in the interior of the domain,
which is enough to guarantee that the linearization exists.

III. IMPLEMENTATION

The proposed methods described above have been imple-
mented as the Python package DCCP, which extends the
package CVXPY. New methods were added to CVXPY to
return the domain of a DCP expression (as a list of constraints), and gradients (or sub/supergradients) were added to the atoms. The linearization, damping, and initialization are handled by the package DCCP. Users can form any problem of the form (2) conforming to the DCCP rules, with each expression composed of functions in the CVXPY library.

When the `solve(method = 'dccp')` method is called on a problem object, DCCP first verifies that the problem satisfies the DCCP rules. Then it splits each non-affine equality constraint \( l_i = r_i \) into \( l_i \leq r_i \) and \( l_i \geq r_i \). The curvature of the objective and the left and right hand sides of each constraint is checked, and if necessary, linearized. In the linearization the function value and gradient are CVXPY parameters, which are constants whose value can change without reconstructing the problem. For each constraint in which the left or right hand side is linearized, a slack variable is introduced, and added to the objective. For any expression that is linearized, the domain of the original expression is added as a constraint.

If a valid initial value for the variable \( x \) is given by the user, it is used. Otherwise several random points are generated from the standard normal distribution (or the uniform distribution on \([0, 1]\) if the variable is declared as nonnegative) and projected onto the domain \( D \) (which, according to the discussion in §II-B, is a DCP problem). The projected points are then averaged to give an initial value for \( x \). This random initialization is a generic heuristic for finding a starting point in the interior of \( D \), and seems to work well in practice.

Algorithm I.1 is then applied to the convexified problem. In each iteration the parameters in the linearizations (which are function and gradient values) are updated based on the current value of the variables. If a gradient (super/subgradient) with respect to any variable does not exist, damping is applied to all variables. The convexified problem at each iteration is solved using CVXPY.

Some useful functions and attributes in the DCCP package are listed below.

- Function `is_dccp(problem)` returns a boolean indicating if an optimization problem is DCCP.
- Attribute `expression.gradient` returns a dictionary of the gradients (or sub/supergradient) of a DCP expression with respect to its variables at the points specified by `variable.value`. (This attribute is also in the core CVXPY package.)
- Function `linearize(expression)` returns the linearization (3) of a DCP expression.
- Attribute `expression.domain` returns a list of constraints describing the domain of a DCP expression. (This attribute is also in the core CVXPY package.)
- Function `convexify(constraint)` returns the transformed constraint (without slack variables) satisfying DCP of a DCCP constraint.
- Method `problem.solve(method = 'dccp')` carries out the proposed algorithm, and returns the value of the transformed cost function, the value of the weight \( \mu_k \), and the maximum value of slack variables at each iteration \( k \). An optional parameter `ccp_times` is used to set the number of times to run the CCP based algorithm, using the randomized initialization.

IV. EXAMPLES

In this section we describe some simple examples, showing how they can be expressed using DCCP, and give the results. In each case we run the default solve method, with no tuning or adjustment of algorithm parameters.

A. Path planning

The goal is to find the shortest path connecting points \( a \) and \( b \) in \( \mathbb{R}^d \) that avoids \( m \) circles, centered at \( p_j \) with radius \( r_j \), \( j = 1, \ldots, m \) [38]. After discretizing the arc length parametrized path into points \( x_0, \ldots, x_n \), the problem is posed as

\[
\begin{align*}
\text{minimize} & \quad L \\
\text{subject to} & \quad x_0 = a, \quad x_n = b \\
& \quad \|x_i - x_{i-1}\|_2 \leq L/n, \quad i = 1, \ldots, n \\
& \quad \|x_i - p_j\|_2 \geq r_j, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m,
\end{align*}
\]

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

The code is given below.

```python
x = Variable(d,n+1)
L = Variable()
constr = [x[:,0] == a, x[:,n] == b]
for i in range(1,n+1):
    constr += [norm(x[:,i]-x[:,i-1]) <= L/n]
for j in range(m):
    constr += [norm(x[:,i] - center[:,j]) >= r[j]]
prob = Problem(Minimize(L), constr)
prob.solve(method = 'dccp')
```

An example with \( d = 2 \) and \( n = 50 \) is shown in figure 1.

B. Control with collision avoidance

We have \( n \) linear dynamic systems, given by

\[
x_{i+1} = A^i x_i + B^i u_i, \quad y_i = C^i x_i, \quad i = 1, \ldots, n,
\]

where \( t = 0, 1, \ldots, T \) denotes (discrete) time, \( x_i \) are the states, and \( y_i \) are the outputs. At each time \( t \) for \( t = 0, \ldots, T \) the \( n \) outputs \( y_i \) are required to keep a distance of at least
d_{\text{min}}$ from each other [39]. The initial states $x_i^0$ and ending states $x_i^n$ are given by $x_{\text{init}}^i$ and $x_{\text{end}}^i$, and the inputs are limited by $\|u_i\|_\infty \leq f_{\text{max}}$. We will minimize a sum of the $\ell_1$ norms of the inputs, an approximation of fuel use. (Of course we can have any convex state and input constraints, and any convex objective.) This gives the problem

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^n \sum_{t=0}^{T-1} \|u_t^i\|_1 \\
\text{subject to} & \quad x_0^i = x_{\text{init}}^i, \quad x_T^i = x_{\text{end}}^i, \quad i = 1, \ldots, n \\
& \quad x_{t+1}^i = A^i x_t^i + B^i u_t^i, \quad t = 0, \ldots, T-1, \quad i = 1, \ldots, n \\
& \quad \|y_t^i - y_{t-1}^i\|_2 \geq d_{\text{min}}, \quad t = 0, \ldots, T, \quad 1 \leq i < j \leq n \\
& \quad y_t^i = C^i x_t^i, \quad \|u_t^i\|_\infty \leq f_{\text{max}}, \quad t = 0, \ldots, T-1, \quad i = 1, \ldots, n,
\end{align*}
\]

where $x_t^i$, $y_t^i$, and $u_t^i$ are variables.

The code can be written as follows.

```python
constr = []
cost = 0
for i in range(n):
    for t in range(T):
        u[i] += [Variable(d)]
        constr += [norm(u[i][-1], 'inf') <= f_max]
        cost += norm(u[i][-1], 1)
        y[i] += [Variable(d)]
        x[i] += [Variable(2*d)]
        constr += [y[i][-1] == C[i]*x[i][-1]]
    for t in range(T-1):
        constr += [x[i][t+1] == A[i]*x[i][t] + B[i]*u[i][t]]
    for t in range(T):
        for j in range(i+1, n):
            constr += [norm(y[i][t] - y[j][t], 2) >= d_min]
prob = Problem(Minimize(cost), constr)
prob.solve(method = 'dccp')
```

We consider an instance with $n = 2$, with outputs (positions) $y_t^i \in \mathbb{R}^2$, $d_{\text{min}} = 0.6$, $f_{\text{max}} = 0.5$, $T = 100$. The linear dynamic system matrices are

\[
A^i = \begin{bmatrix}
1 & 0 & 0.1 & 0 \\
0 & 1 & 0 & 0.1 \\
0 & 0 & 0.95 & 0 \\
0 & 0 & 0 & 0.95
\end{bmatrix},
B^i = \begin{bmatrix}
0 & 0 \\
0 & 0 \\
0.1 & 0 \\
0 & 0.1
\end{bmatrix},
C^i = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{bmatrix}.
\]

The results are in figure 2, where the black arrows in the first two figures show initial and final states (position and velocity), and the black dashed line in the third figure shows $d_{\text{min}}$. The left plot shows the output trajectory without collision avoidance, the middle shows it with collision avoidance, and the right plot shows the distance between outputs versus time.

C. Sparse recovery using $\ell_{1/2}$ ‘norm’

The aim is to recover a sparse nonnegative signal $x_0 \in \mathbb{R}^n$ from a measurement vector $y = Ax_0$, where $A \in \mathbb{R}^{m \times n}$ (with $m < n$) is a known sensing matrix [40]. A common heuristic based on convex optimization is to minimize the $\ell_1$ norm of $x$ (which reduces here to the sum of entries of $x$) subject to $y = Ax_0$ (and here, $x \geq 0$). It has been proposed to minimize the sum of the $\text{squareroots}$ of the entries of $x$, which since $x \geq 0$ is the same as minimizing the squareroot of the $\ell_{1/2}$ ‘norm’ (which is not convex, and therefore not a norm), to obtain better recovery. The optimization problem is

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^n \sqrt{x_i} \\
\text{subject to} & \quad y = Ax,
\end{align*}
\]

where $x$ is the variable. (The constraint $x \geq 0$ is implicit, since this is the objective domain.) This is a nonconvex problem, directly in DCCP form.

The corresponding code is as follows.

```python
x = Variable(n, 1)
x.value = np.ones((n,1))
obj = Minimize(sum_entries(sqrt(x)))
prob = Problem(obj, [A*x == y])
prob.solve(method = 'dccp')
```

In a numerical simulation, we take $n = 100$, $A_{ij} \sim \mathcal{N}(0,1)$, the positions of the nonzero entries in $x_0$ are from uniform distribution, and the nonzero values are the absolute values of $\mathcal{N}(0,100)$ random variables. To count the probability of recovery, 100 independent instances are tested, and a recovery is successful if the relative error $\|\hat{x} - x_0\|_2/\|x_0\|_2$ is less than 0.01. In each instance, the
cardinality takes 6 values from 30 to 50, according to which \( x_0 \) is generated, and \( A \) is generated for each \( m \) taking one of the 6 values from 50 to 80. The results in figure 3 verify that nonconvex recovery is more effective than convex recovery.

**ACKNOWLEDGMENTS**

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-114747, by the DARPA X-DATA and SIMPLEX programs, and by the CSC State Scholarship Fund.

**REFERENCES**


