Disciplined Nonlinear Programming

Daniel Cederberg

Based on joint work with William Zhang, Parth Nobel, and Stephen Boyd

March 2026 [1]

[1] Versions of this talk have been given at Stanford, KTH, and Amazon.

Outline

Convex optimization

Nonlinear programming

Disciplined nonlinear programming

Examples

Conclusions

Convex optimization

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & Ax = b \end{array}$$

- variable $x \in \mathbf{R}^n$
- equality constraints are linear and $f_0, \ldots, f_m$ are convex
- why care about convexity?

Convex optimization

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & Ax = b \end{array}$$

- variable $x \in \mathbf{R}^n$
- equality constraints are linear and $f_0, \ldots, f_m$ are convex
- why care about convexity?
  - **classic answer**: local optimum = global optimum
  - **modern answer**: very actionable because of reliable software

Disciplined convex programming (DCP)

- **motivation**: difficult to verify convexity of an arbitrarily constructed problem

- a grammar for specifying convex optimization problems [Grant et al., 2006]
- convex and concave functions are constructed using simple composition rules
- a problem that follows the DCP rules is convex-by-construction

Disciplined convex programming (DCP)

- **motivation**: difficult to verify convexity of an arbitrarily constructed problem

- a grammar for specifying convex optimization problems [Grant et al., 2006]
- convex and concave functions are constructed using simple composition rules
- a problem that follows the DCP rules is convex-by-construction
- and can be solved globally and efficiently (in theory and in practice)

Modeling languages implementing DCP

- Yalmip & CVX (MATLAB), CVXPY (Python), CVXR (R), Convex.jl (Julia)
- classic commercial software like AMPL and GAMS do not implement DCP

Modeling languages implementing DCP

- Yalmip & CVX (MATLAB), CVXPY (Python), CVXR (R), Convex.jl (Julia)
- classic commercial software like AMPL and GAMS do not implement DCP

- two main tasks of DCP-based modeling languages:
  - **analysis**: verify that a high level problem description follows the DCP rules
  - **synthesis**: build an equivalent problem on standard conic form:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + s = b \\ & s \in K \end{array}$$

variables x and s, and problem data A, b, c and convex cone K

Modeling languages implementing DCP

- Yalmip & CVX (MATLAB), CVXPY (Python), CVXR (R), Convex.jl (Julia)
- classic commercial software like AMPL and GAMS do not implement DCP

- two main tasks of DCP-based modeling languages:
  - **analysis**: verify that a high level problem description follows the DCP rules
  - **synthesis**: build an equivalent problem on standard conic form:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + s = b \\ & s \in K \end{array}$$

  variables x and s, and problem data $A,\ b,\ c$ and convex cone K

- no communication between modeling language and solver beyond A, b, c, K

Disciplined convex programming in practice: An example

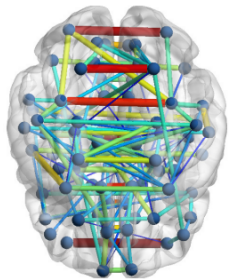
$$\text{minimize} \; -\log\det X + \mathbf{Tr}(SX) + \lambda\|X\|_1$$

- variable $X \in \mathbf{S}^n$ represents an inverse covariance matrix of a Gaussian vector
- the problem data is the sample covariance matrix $S \in \mathbf{S}^n_+$
- called *sparse inverse covariance estimation* (applications in graphical modeling)

Disciplined convex programming in practice: An example

$$\text{minimize} \; -\log\det X + \mathbf{Tr}(SX) + \lambda\|X\|_1$$

- variable $X \in \mathbf{S}^n$ represents an inverse covariance matrix of a Gaussian vector
- the problem data is the sample covariance matrix $S \in \mathbf{S}^n_+$
- called *sparse inverse covariance estimation* (applications in graphical modeling)



Disciplined convex programming in practice: An example

$$\text{minimize} \; -\log\det X + \mathbf{Tr}(SX) + \lambda\|X\|_1$$

What the user sees:

```
import cvxpy as cp
S, lmbda = ...
X = cp.Variable((n, n), symmetric=True)
obj = -cp.logdet(X) + cp.trace(S @ X) +
    lmbda * cp.norm1(X)
prob = cp.Problem(cp.Minimize(obj))
prob.solve()
```

What the solver sees:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + s = b \\ & s \in K \end{array}$$

where A, b, c, K and x are ~~defined on the next slide~~ impossible to derive by hand

What do to when your problem is not convex?

- sometimes possible to reformulate as convex using *e.g.*, a change of variables
  - maximum likelihood estimation for exponential families
  - feedback control design
  - geometric programming
- implement simple heuristics based on convex optimization
  - sequential convex programming
  - convex-concave procedure
  - alternating convex optimization
- use a general-purpose nonlinear programming solver

Outline

Convex optimization

Nonlinear programming

Disciplined nonlinear programming

Examples

Conclusions

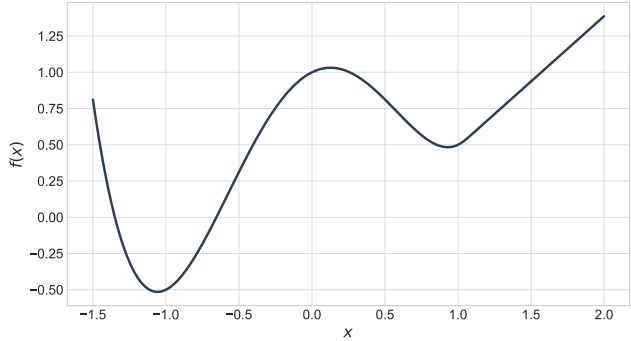
Nonlinear programming (NLP)

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

- $f : \mathbf{R}^n \to \mathbf{R}$, $c : \mathbf{R}^n \to \mathbf{R}^p$ differentiable, not necessarily convex
- we seek a **local solution**

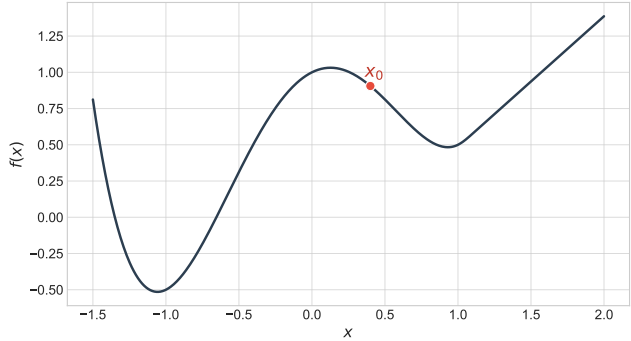
Misconception 1 about NLP

NLP solvers always compute global minimizers



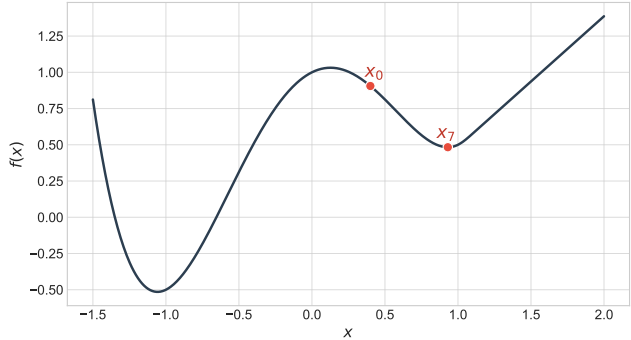
Misconception 1 about NLP

NLP solvers always compute global minimizers



Misconception 1 about NLP

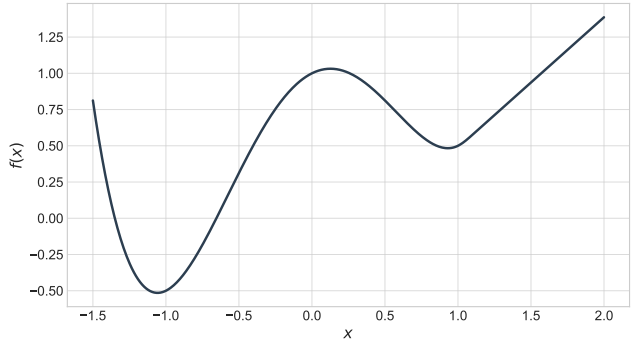
NLP solvers always compute global minimizers



- Ipopt [Wächter and Biegler, 2006] converges to a local (but not global) minimizer

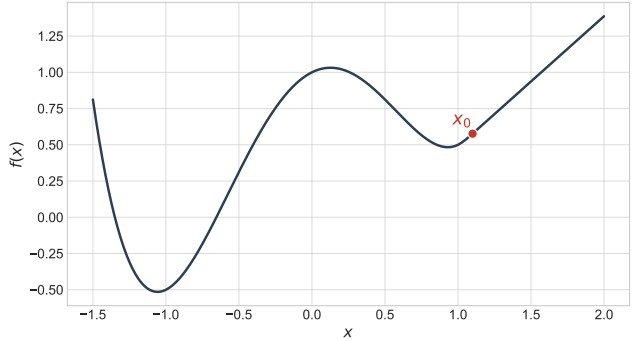
Misconception 2 about NLP

NLP solvers always converge to the local minimizer closest to the initial point



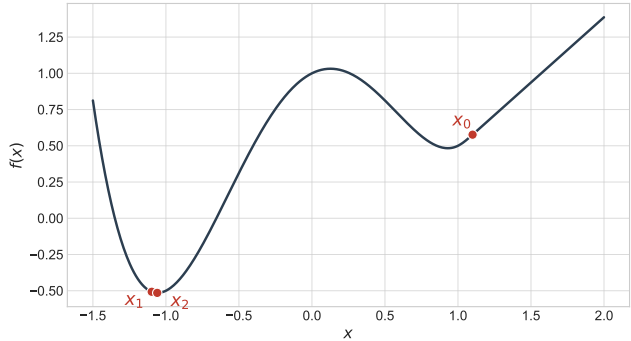
Misconception 2 about NLP

NLP solvers always converge to the local minimizer closest to the initial point



Misconception 2 about NLP

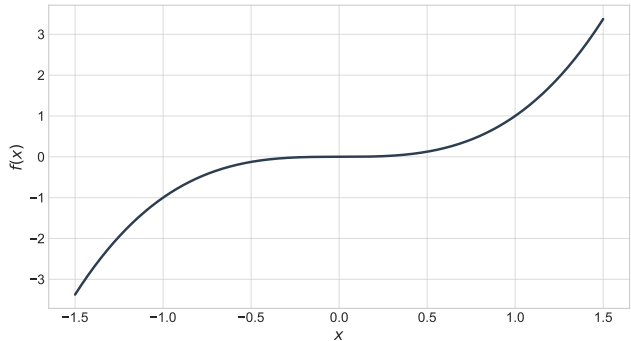
NLP solvers always converge to the local minimizer closest to the initial point



- Ipopt converges to a local minimizer, but not the one closest to the initial point

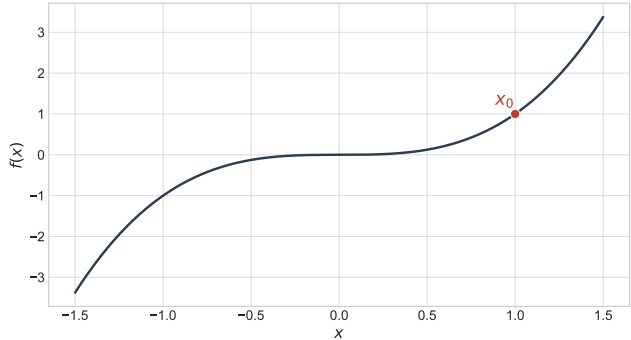
Misconception 3 about NLP

NLP solvers always converge to a local minimizer



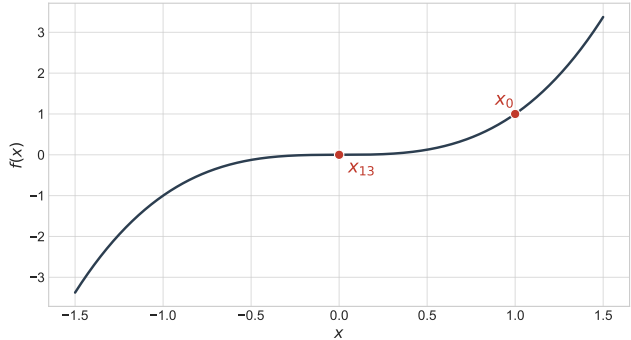
Misconception 3 about NLP

NLP solvers always converge to a local minimizer



Misconception 3 about NLP

NLP solvers always converge to a local minimizer



- Ipopt converges to a saddle point when minimizing $f(x) = x^3$ initialized at $x_0 = 1$

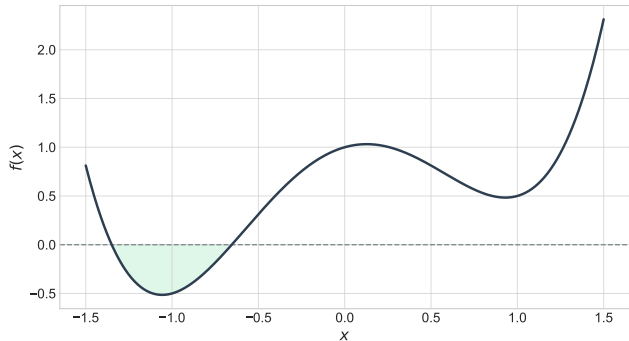
Misconception 4 about NLP

NLP solvers always converge to a 'good' feasible point (KKT point)

Misconception 4 about NLP

minimize $f(x) = x^4 - 2x^2 + 0.5x + 1$
subject to $f(x) \leq 0$
(Also see Wächter & Bowly, Gurobi, 2026)

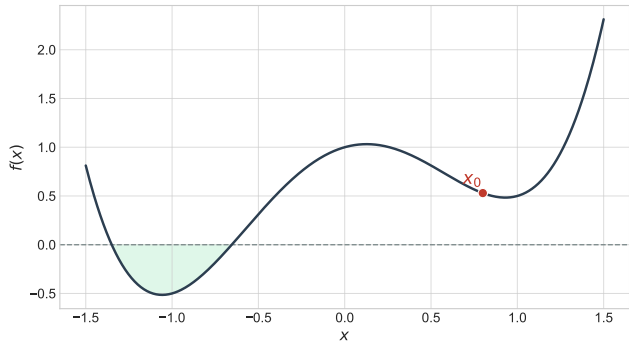
NLP solvers always converge to a 'good' feasible point (KKT point)



Misconception 4 about NLP

minimize $f(x) = x^4 - 2x^2 + 0.5x + 1$
subject to $f(x) \leq 0$
(Also see Wächter & Bowly, Gurobi, 2026)

NLP solvers always converge to a 'good' feasible point (KKT point)

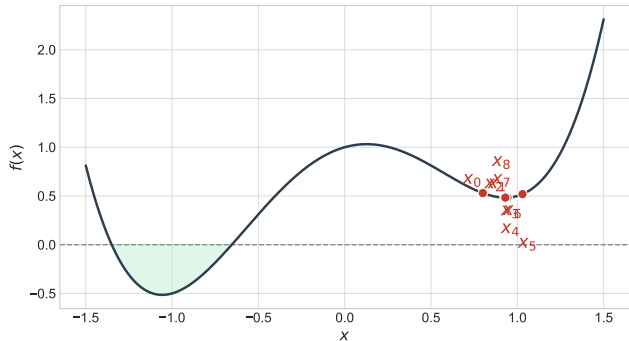


Misconception 4 about NLP

minimize	$f(x) = x^4 - 2x^2 + 0.5x + 1$
subject to	$f(x) \leq 0$

(Also see Wächter & Bowly, Gurobi, 2026)

NLP solvers always converge to a 'good' feasible point (KKT point)

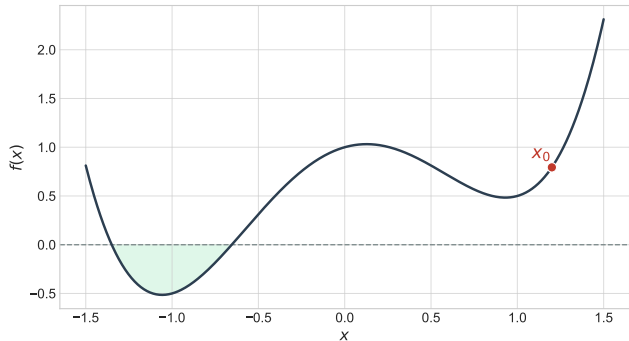


- Ipopt converges to an infeasible point when initialized at $x_0 = 0.8$

Misconception 4 about NLP

minimize $f(x) = x^4 - 2x^2 + 0.5x + 1$
subject to $f(x) \leq 0$
(Also see Wächter & Bowly, Gurobi, 2026)

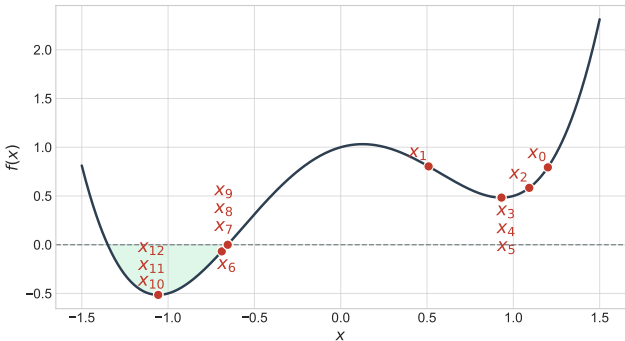
NLP solvers always converge to a 'good' feasible point (KKT point)



Misconception 4 about NLP

minimize $f(x) = x^4 - 2x^2 + 0.5x + 1$
subject to $f(x) \leq 0$
(Also see Wächter & Bowly, Gurobi, 2026)

NLP solvers always converge to a 'good' feasible point (KKT point)



- Ipopt converges to global minimizer when initialized at $x_0 = 1.2$

Summary of misconceptions

NLP solvers are **not** guaranteed to

- compute global minimizers
- compute the local minimizer closest to the initial point
- compute local minimizers at all (might converge to saddle points)
- compute a feasible point at all (might converge to an infeasible point)

Summary of misconceptions

NLP solvers are **not** guaranteed to

- compute global minimizers
- compute the local minimizer closest to the initial point
- compute local minimizers at all (might converge to saddle points)
- compute a feasible point at all (might converge to an infeasible point)

but in practice, NLP solvers often work well (perhaps with decent initialization)

Constraint qualifications (CQs)

- NLP solvers attempt to compute **KKT points**
- KKT conditions rely on a local linear approximation of the feasible set
  - KKT is necessary for local optimality only when this approximation is “good enough”
  - (formally: the tangent cone equals the linearized tangent cone)
- **constraint qualifications** are conditions that guarantee this
  - most common CQ: LICQ (gradients of active constraints are linearly independent)

Constraint qualifications (CQs)

- NLP solvers attempt to compute **KKT points**
- KKT conditions rely on a local linear approximation of the feasible set
  - KKT is necessary for local optimality only when this approximation is “good enough”
  - (formally: the tangent cone equals the linearized tangent cone)
- **constraint qualifications** are conditions that guarantee this
  - most common CQ: LICQ (gradients of active constraints are linearly independent)
  - we found LICQ to be important in practice

Modeling languages for NLP

- YALMIP (MATLAB), JuMP (Julia), Pyomo (Python), and CasADi (C++)
- commercial software like AMPL, GAMS, AIMMS

Modeling languages for NLP

- YALMIP (MATLAB), JuMP (Julia), Pyomo (Python), and CasADi (C++)
- commercial software like AMPL, GAMS, AIMMS

- main task of NLP modeling languages:
  - provide solvers with derivatives
- communication between modeling language and solver in every iteration

When many existing NLP modeling languages do a bad job

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

- variable x, regularization parameter $\lambda > 0$
- convex problem (lasso) so should be solved gracefully
- the solution occurs at a point of nondifferentiability
- objective is treated as a black box and “derivatives” are supplied to the solver
- ignoring nondifferentiability might cause solvers to struggle

When existing NLP modeling languages do a bad job

output from Ipopt when interfacing it using Casadi:

```
This is Ipopt version 3.14.11, running with linear solver MUMPS 5.4.1.

...

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  1.1126107e+01 0.00e+00 1.71e+00  -1.7 1.29e+08  -7.8 1.00e+00 1.32e-23f 77
  11  1.3152054e+10 0.00e+00 1.71e+00  -1.7 4.09e+08  -8.3 1.00e+00 1.00e+00w  1
  12  3.5423855e+10 0.00e+00 1.66e+00  -1.7 1.13e+09  -8.8 1.00e+00 1.00e+00w  1
  13  1.4112259e+11 0.00e+00 1.58e+00  -1.7 3.39e+09  -9.2 1.00e+00 1.00e+00w  1
Cannot call restoration phase at point that is almost feasible (violation 0.000000e+00).
Abort in line search due to no other fall back.

Number of Iterations....: 13

EXIT: Error in step computation!
```

Outline

Disciplined nonlinear programming (DNLP)

- a new grammar for specifying nonlinear programs
- smooth functions can be freely mixed with nonsmooth functions
- rules governing how nonsmooth functions can appear
- smooth = twice continuously differentiable in the interior of their domain

Disciplined nonlinear programming (DNLP)

- a new grammar for specifying nonlinear programs
- smooth functions can be freely mixed with nonsmooth functions
- rules governing how nonsmooth functions can appear
- smooth = twice continuously differentiable in the interior of their domain

a DNLP-compliant problem can be transformed to a form that

- only involves smooth functions
- does not introduce LICQ violations in the transformation

The DNLP grammar

in DNLP, problems are constructed using **atoms** with known attributes

- atoms are divided into three categories:
  - **smooth**
  - **nonsmooth convex**
  - **nonsmooth concave**
- DNLP enforces simple rules on how atoms can be composed

The DNLP grammar

in DNLP, problems are constructed using **atoms** with known attributes

- atoms are divided into three categories:
  - **smooth**
  - **nonsmooth convex**
  - **nonsmooth concave**
- DNLP enforces simple rules on how atoms can be composed
- the DNLP rules are very simple if you know the DCP rules:

a problem is DNLP if it is DCP when smooth atoms are linearized

Some new DNLP atoms

Atom	Definition
`multiply`	$\phi(x, y) = xy$
`quad_form`	$\phi(x) = x^T Q x$ where $Q \in \mathbf{S}^n$
`sin`	$\phi(x) = \sin x$
`sigmoid`	$\phi(x) = \frac{1}{1+e^{-x}}$
`normal_cdf`	$\phi(x) = (1/\sqrt{2\pi}) \int_{-\infty}^{x} e^{-t^2/2} dt$

- these atoms are not convex or concave, but they are smooth

Dealing with nonsmooth atoms

- a DNLP problem is equivalent to a problem where any
  - nonsmooth convex atom ϕ appears in a constraint of the form $\phi(x) \leq t$
  - nonsmooth concave atom ϕ appears in a constraint of the form $\phi(x) \geq t$

Dealing with nonsmooth atoms

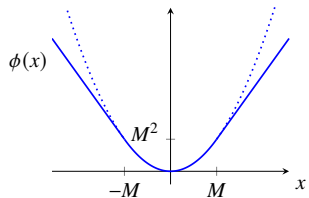
- a DNLP problem is equivalent to a problem where any
  - nonsmooth convex atom ϕ appears in a constraint of the form $\phi(x) \leq t$
  - nonsmooth concave atom ϕ appears in a constraint of the form $\phi(x) \geq t$
- these constraints can be formulated using smooth functions [Grant and Boyd, 2008]
  - the reformulations do not introduce LICQ violations
- **example**: $|x| \leq t$ can be reformulated as $-t \leq x \leq t$

Dealing with nonsmooth atoms

- a DNLP problem is equivalent to a problem where any
  - nonsmooth convex atom ϕ appears in a constraint of the form $\phi(x) \leq t$
  - nonsmooth concave atom ϕ appears in a constraint of the form $\phi(x) \geq t$
- these constraints can be formulated using smooth functions [Grant and Boyd, 2008]
  - the reformulations do not introduce LICQ violations
- **example**: $|x| \leq t$ can be reformulated as $-t \leq x \leq t$
- other nonsmooth atoms, such as the Huber atom

$$\phi(x) = \begin{cases} x^2 & |x| \leq M \\ 2M|x| - M^2 & |x| > M \end{cases}$$

have more complex reformulations



Outline

A DNLP extension of CVXPY

- DNLP is implemented in CVXPY version 1.9
- supported solvers:
  - IPOPT (open-source, interior-point method)
  - Uno [Vanaret and Leyffer, 2025] (open-source, IPM and SQP)
  - KNITRO (commercial, many algorithms)
  - COPT (commercial, interior-point method)
- computes sparse Jacobians and Hessians via C backend

Example: Circle packing

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

- goal is to pack n circles with given radii r_i into smallest square
- variables are the circle centers $c_i \in \mathbf{R}^2$
- becomes DCP when smooth atoms are linearized, so DNLP-compliant

Example: Circle packing

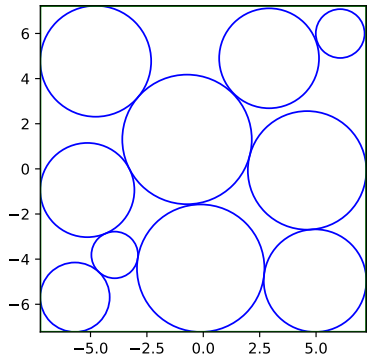
$$\begin{array}{ll} \text{minimize} & \max_{i=1,\ldots,n} \left(\|c_i\|_\infty + r_i\right) \\ \text{subject to} & \|c_i - c_j\|_2^2 \geq (r_i + r_j)^2, \quad 1 \leq i < j \leq n, \end{array}$$

- goal is to pack n circles with given radii r_i into smallest square
- variables are the circle centers $c_i \in \mathbf{R}^2$
- becomes DCP when smooth atoms are linearized, so DNLP-compliant

```
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)                            # or prob.solve(nlp=True, best_of=500)
```

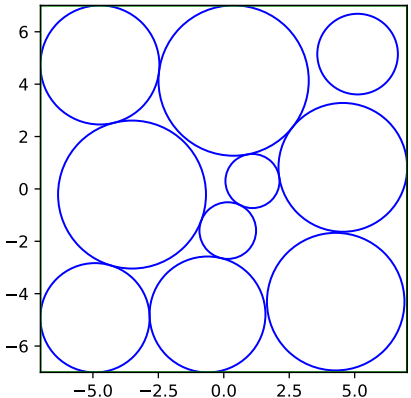
Example: Circle packing

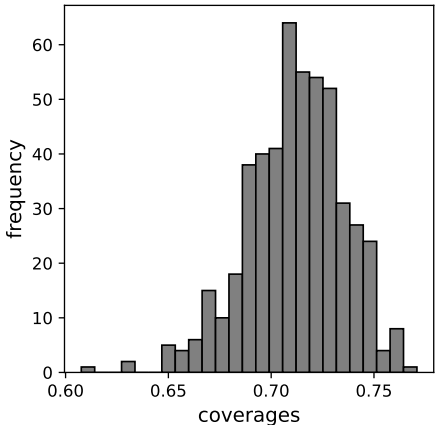
- result for $n = 10$ circles with random radii, with random initial point
- the fraction of the square covered by the circles is 0.72



Example: Circle packing

- result for same problem instance over 500 random initializations using `best_of=500`
- best coverage found is 0.77





Example: Sparse signal recovery

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

- goal is to recover a sparse signal $x_0 \in \mathbf{R}^n$ from m given measurements $y = Ax_0$
- variable is the signal $x \in \mathbf{R}^n$
- the objective is nonconvex but promotes sparsity better than the ℓ_1 norm
- becomes DCP when smooth atoms are linearized, so DNLP-compliant

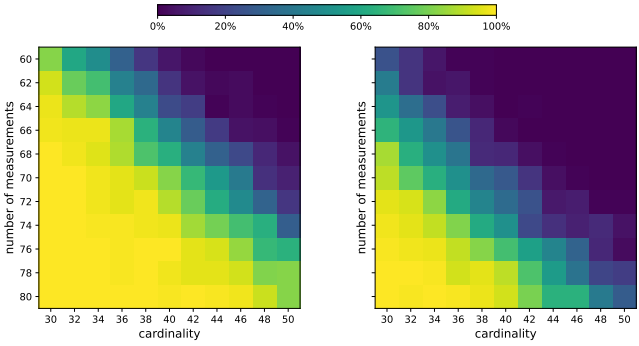
Example: Sparse signal recovery

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

- goal is to recover a sparse signal $x_0 \in \mathbf{R}^n$ from m given measurements $y = Ax_0$
- variable is the signal $x \in \mathbf{R}^n$
- the objective is nonconvex but promotes sparsity better than the ℓ_1 norm
- becomes DCP when smooth atoms are linearized, so DNLP-compliant

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

Example: Sparse signal recovery



- probability of successful recovery for signal dimension $n = 100$, nonconvex approach (left) and convex ℓ_1 approach (right)

Many more examples in the paper

- planning and geometry
- signal processing
- finance
- energy systems
- statistics

Outline

Conclusions

- DNLP is a new grammar for specifying NLPs
  - no rules if all atoms are smooth
  - mitigates solver failures caused by nonsmoothness
- the DNLP-extension of CVXPY makes it easy to use NLP solvers
  - enables high level, human readable problem descriptions
  - automatically generates smooth reformulations of nonsmooth atoms
  - ideal for teaching and research