Code Generation for Embedded Convex Optimization

Jacob Mattingley and **Stephen Boyd**

Stanford University

Lund University, 22/8/2012

Convex optimization

- Problems solvable reliably and efficiently
- Widely used in scheduling, finance, engineering design
- Solve every few minutes or seconds

Code generation for embedded convex optimization

Replace 'minutes' with 'milliseconds' and eliminate failure

Agenda

I. **Introduction** to embedded convex optimization and CVXGEN

II. **Demonstration** of CVXGEN

III. **Techniques** for constructing fast, robust solvers

IV. **Verification** of technical choices

V. **Final notes** and conclusions

Part I: **Introduction**

1. Embedded convex optimization
2. Embedded solvers
3. CVXGEN

Embedded convex optimization: **Requirements**

Embedded solvers must have:

- **Time limit**, sometimes strict, in milliseconds or microseconds
- **Simple footprint** for portability and verification
- **No failures**, even with somewhat poor data

Embedded convex optimization: **Exploitable features**

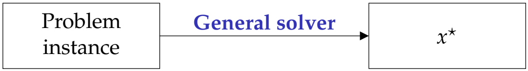
Embedded solvers can exploit:

- Modest accuracy requirements
- Fixed dimensions, sparsity, structure
- Repeated use
- **Custom design** in pre-solve phase

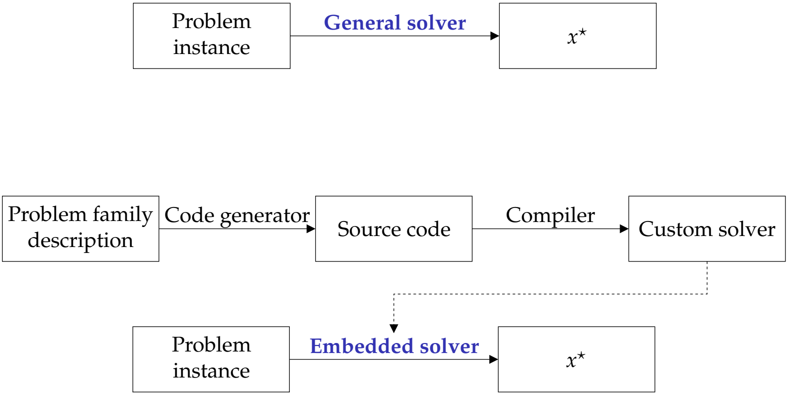
Embedded convex optimization: **Applications**

- Signal processing, model predictive control
- Fast simulations, Monte Carlo
- Low power devices
- Sequential QP, branch-and-bound

Embedded convex optimization: **Pre-solve phase**



Embedded convex optimization: **Pre-solve phase**



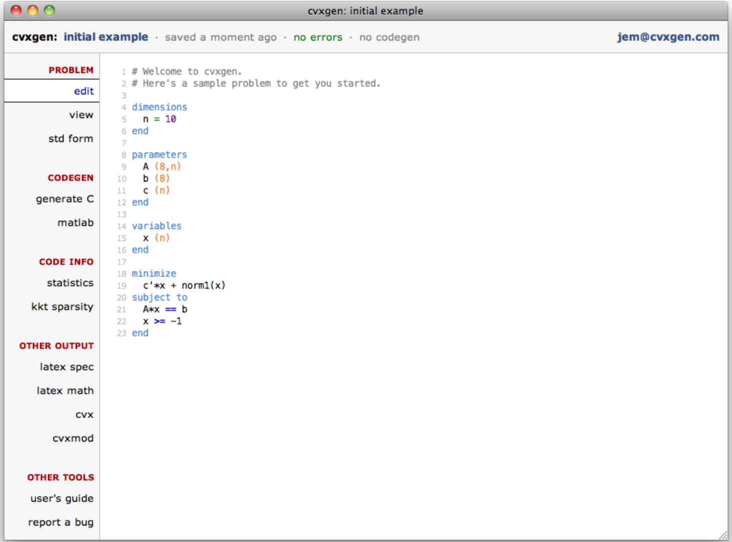
CVXGEN

- Code generator for embedded convex optimization
- Mattingley, Boyd
- Disciplined convex programming input
- Targets small QPs in flat, library-free C

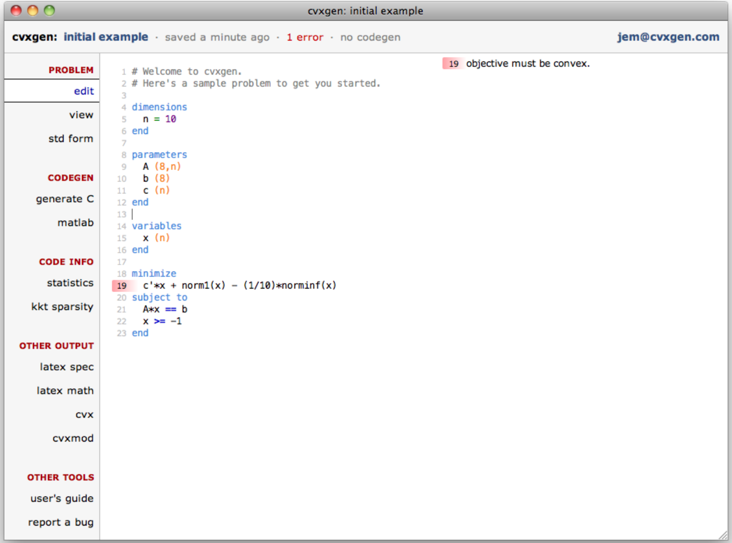
Part II: **Demonstration**

1. Manipulating optimization problems with CVXGEN
2. Generating and using solvers
3. Important hidden details

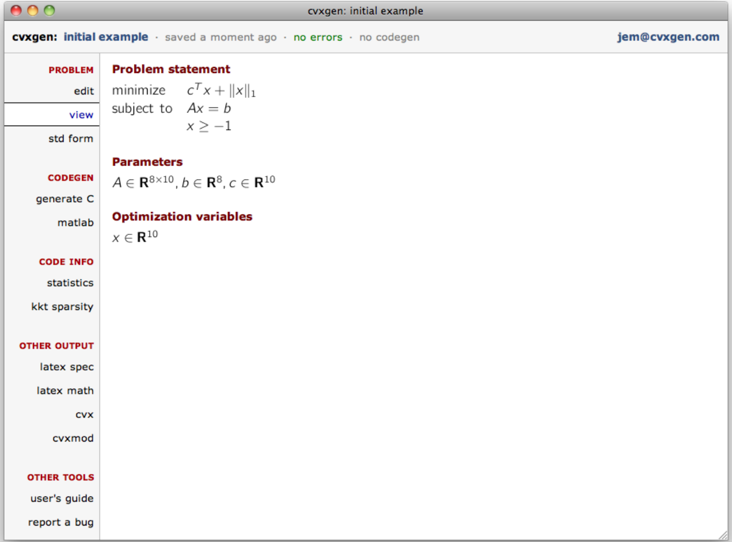
CVXGEN: **Problem specification**



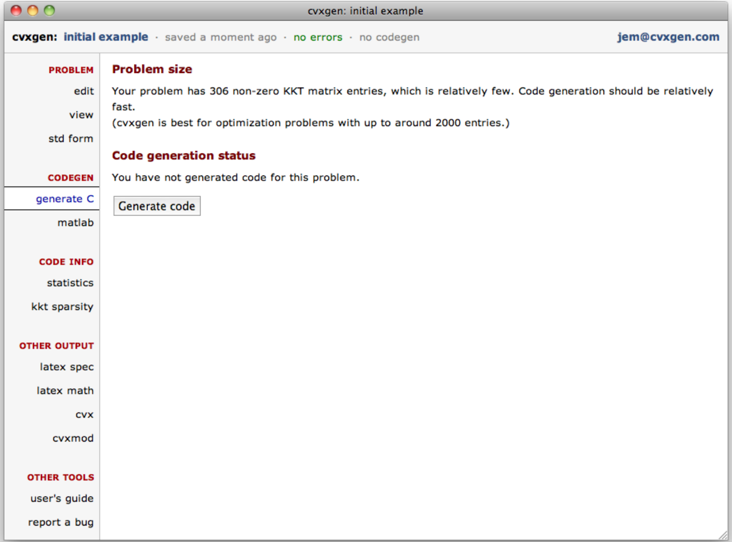
CVXGEN: **Automatic checking**



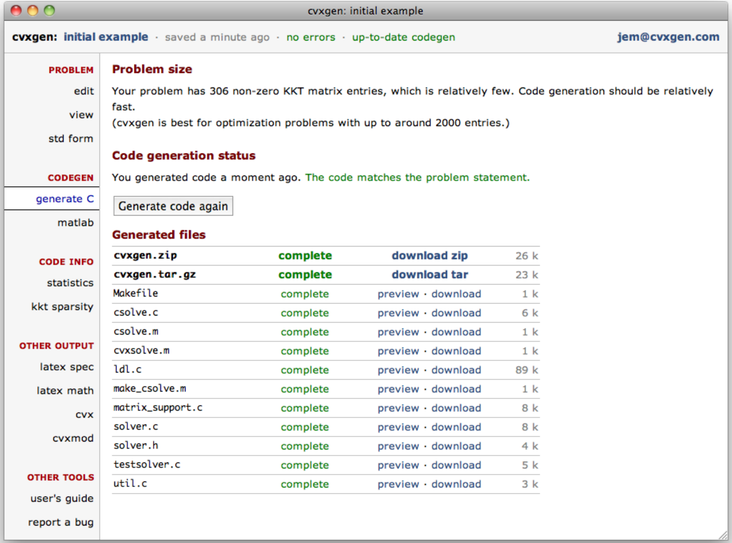
CVXGEN: Formatted problem statement



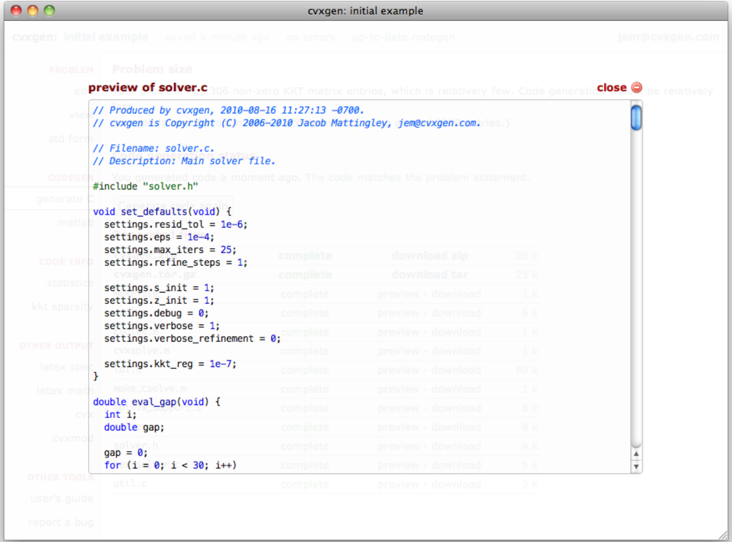
CVXGEN: Single-button code generation



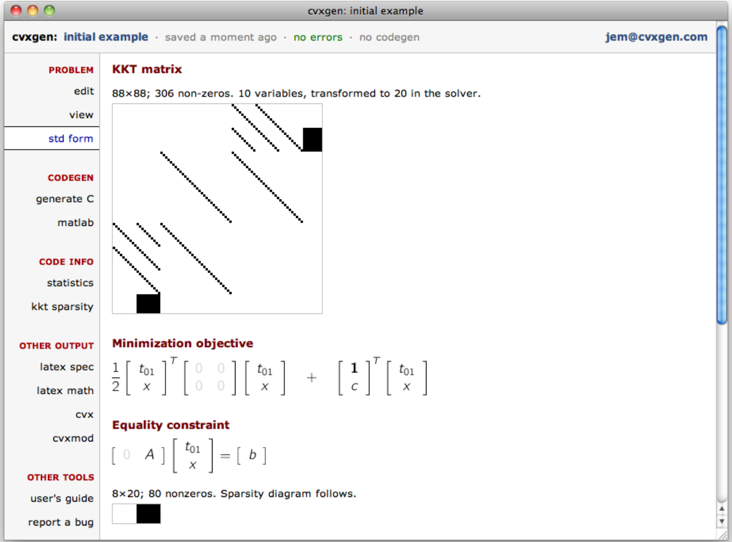
CVXGEN: **Completed code generation**



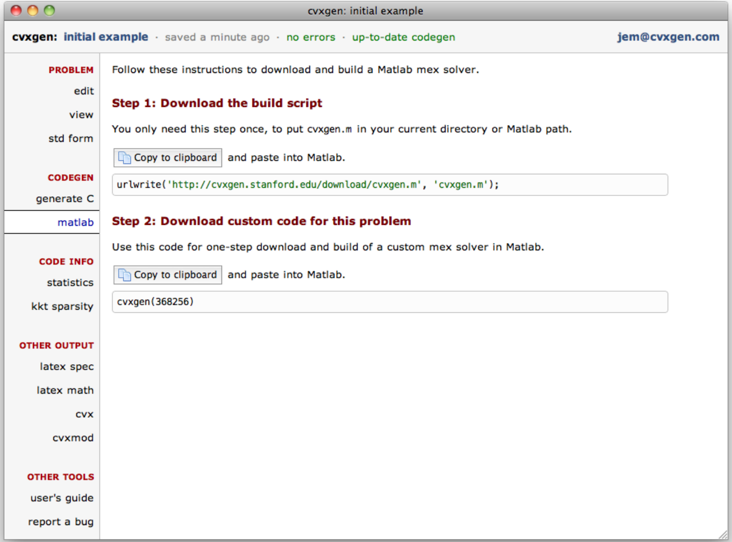
CVXGEN: **Fast, problem-specific code**



CVXGEN: **Automatic problem transformations**



CVXGEN: **Automatically generated Matlab interface**



Important hidden details

Important details not seen in demonstration:

- Extremely high speeds
- Bounded computation time
- Algorithm robustness

Part III: **Techniques**

1. Transformation to canonical form
2. Interior-point algorithm
3. Solving the KKT system
   - Permutation
   - Regularization
   - Factorization
   - Iterative refinement
   - Eliminating failure
4. Code generation

Transformation to canonical form

- Problem description uses high-level langauge
- Solve problems in canonical form: with variable $x \in \mathbf{R}^n$,

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Q x + q^T x \\ \text{subject to} & Gx \leqslant h, \quad Ax = b \end{array}$$

- Transform high-level description to canonical form automatically:
    1. Expand convex functions via epigraphs.
    2. Collect optimization variables into single vector variable.
    3. Shape parameters into coefficient matrices and constants.
    4. Replace certain products with more efficient pre-computations.
- Generate code for forwards, backwards transformations

Transformation to canonical form: **Example**

- Example problem in original form with variables x, y:

$$\begin{array}{ll} \text{minimize} & x^T Q x + c^T x + \alpha \|y\|_1 \\ \text{subject to} & A(x-b) \leqslant 2y \end{array}$$

- After epigraphical expansion, with new variable t:

$$\begin{array}{ll} \text{minimize} & x^T Q x + c^T x + \alpha \mathbf{1}^T t \\ \text{subject to} & A(x-b) \leqslant 2y, \quad -t \leqslant y \leqslant t \end{array}$$

- After reshaping variables and parameters into standard form:

$$\begin{array}{ll} \text{minimize} & \begin{bmatrix} x \\ y \\ t \end{bmatrix}^T \begin{bmatrix} Q & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} + \begin{bmatrix} c \\ \alpha \mathbf{1} \\ 0 \end{bmatrix}^T \begin{bmatrix} x \\ y \\ t \end{bmatrix} \\ \text{subject to} & \begin{bmatrix} A & -2I & 0 \\ 0 & -I & -I \\ 0 & I & I \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \leqslant \begin{bmatrix} Ab \\ 0 \\ 0 \end{bmatrix} \end{array}$$

Solving the standard-form QP

- Standard primal-dual interior-point method with Mehrotra correction
- Reliably solve to high accuracy in 5–25 iterations
- Mehrotra '89, Wright '97, Vandenberghe '09

Algorithm

Initialize via least-squares. Then, repeat:

1. Stop if the residuals and duality gap are sufficiently small.
2. Compute affine scaling direction by solving

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\mathrm{aff}} \\ \Delta s^{\mathrm{aff}} \\ \Delta z^{\mathrm{aff}} \\ \Delta y^{\mathrm{aff}} \end{bmatrix} = \begin{bmatrix} -(A^T y + G^T z + Px + q) \\ -Sz \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix}.$$

3. Compute centering-plus-corrector direction by solving

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\mathrm{cc}} \\ \Delta s^{\mathrm{cc}} \\ \Delta z^{\mathrm{cc}} \\ \Delta y^{\mathrm{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu\mathbf{1} - \mathbf{diag}(\Delta s^{\mathrm{aff}})\Delta z^{\mathrm{aff}} \\ 0 \\ 0 \end{bmatrix},$$

   with

$$\begin{array}{c} \mu = s^T z/p \qquad \sigma = \left((s + \alpha\Delta s^{\mathrm{aff}})^T (z + \alpha\Delta z^{\mathrm{aff}})/(s^T z)\right)^3 \\ \alpha = \sup\{\alpha \in [0,1] \mid s + \alpha\Delta s^{\mathrm{aff}} \geqslant 0,\ z + \alpha\Delta z^{\mathrm{aff}} \geqslant 0\} \end{array}.$$

Algorithm (continued)

4. Combine the updates with

$$\begin{array}{ll} \Delta x = \Delta x^{\text{aff}} + \Delta x^{\text{cc}} & \Delta s = \Delta s^{\text{aff}} + \Delta s^{\text{cc}} \\ \Delta y = \Delta y^{\text{aff}} + \Delta y^{\text{cc}} & \Delta z = \Delta z^{\text{aff}} + \Delta z^{\text{cc}} \end{array}.$$

5. Find

$$\alpha = \min\{1,\ 0.99 \sup\{\alpha \geqslant 0 \mid s + \alpha\Delta s \geqslant 0,\ z + \alpha\Delta z \geqslant 0\}\},$$

   and update

$$\begin{array}{ll} x := x + \alpha\Delta x & s := s + \alpha\Delta s \\ y := y + \alpha\Delta y & z := z + \alpha\Delta z \end{array}.$$

Solving KKT system

- Most computation effort, typically 80%, is solution of KKT system
- Each iteration requires two solves with (symmetrized) KKT matrix

$$K = \left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right]$$

- Quasisemidefinite: block diagonals PSD, NSD
- Use permuted LDL^T factorization with diagonal D, unit lower-triangular L

Solving KKT system: **Permutation issues**

- Factorize $PKP^T = LDL^T$, with permutation matrix P
- L, D unique, if they exist
- P determines nonzero count of L, thus computation time
- Standard method: choose P at solve time
  - Uses numerical values of K
  - Maintains stability
  - Slow (complex data structures, branching)
- CVXGEN: choose P at development time
  - Factorization does not even exist, for some P
  - Even if factorization exists, stability highly dependent on P
  - **How do we fix this?**

Solving KKT system: **Regularization**

- Use regularized KKT system $\widetilde{K}$ instead
- Choose regularization constant $\epsilon > 0$, then instead factor:

$$P\left(\left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{c|c} \epsilon I & 0 \\ \hline 0 & -\epsilon I \end{array}\right]\right) P^T = P\widetilde{K}P^T = LDL^T$$

- $\widetilde{K}$ now quasidefinite: block diagonals PD, ND
- Factorization always exists (Gill et al, '96)

Solving KKT system: **Selecting the permutation**

- Select P at development time to minimize nonzero count of L
- Simple greedy algorithm:

  Create an undirected graph from $\widetilde{K}$.

  While nodes remain, repeat:

  1. For each uneliminated node, calculate the fill-in if it were eliminated next.
  2. Eliminate the node with lowest induced fill-in.
- Can prove that P determines signs of D_{ii} (will come back to this)

Solving KKT system: **Solution**

- Algorithm requires two solutions ℓ with different residuals r, of

$$K\ell = r$$

- Instead, solve

$$\ell = \widetilde{K}^{-1} r = P^T L^{-T} D^{-1} L^{-1} P r$$

- Use cached factorization, forward- and backward-substitution
- **But: solution to wrong system**
- Use iterative refinement

Solving KKT system: **Iterative refinement**

- Want solution to $K\ell = r$, only have operator $\widetilde{K}^{-1} \approx K^{-1}$
- Use iterative refinement:

  Solve $\widetilde{K}\ell^{(0)} = r$.

  Want correction $\delta\ell$ such that $K(\ell^{(0)} + \delta\ell) = r$. Instead:

  1. Compute approximate correction by solving $\widetilde{K}\delta\ell^{(0)} = r - K\ell^{(0)}$.
  2. Update iterate $\ell^{(1)} = \ell^{(0)} + \delta\ell^{(0)}$.
  3. Repeat until $\ell^{(k)}$ is sufficiently accurate.
- Iterative refinement with $\widetilde{K}$ provably converges
- CVXGEN uses only one refinement step

Solving KKT system: **Eliminating failure**

- Regularized factorization cannot fail with exact arithmetic
- Numerical errors can still cause divide-by-zero exceptions
- Only divisions in algorithm are by D_{ii}
- Factorization computes $\widehat{D}_{ii} \neq D_{ii}$, due to numerical errors
- Therefore, given sign ξ_i of D_{ii}, use

$$D_{ii} = \xi_i((\xi_i \widehat{D}_{ii})_+ + \epsilon)$$

- Makes division 'safe'
- Iterative refinement still provably converges

Code generation

- Code generation converts symbolic representation to compilable code
- Use templates [color key: C code, control code, C substitutions]

```
void kkt_multiply(double *result, double *source) {
  - kkt.rows.times do |i|
    result[#{i}] = 0;
    - kkt.neighbors(i).each do |j|
      - if kkt.nonzero? i, j
        result += #{kkt[i,j]}*source[#{j}];
}
```

- Generate extremely explicit code

Code generation: **Extremely explicit code**

- Embedded constants, exposed for compiler optimizations:

```
// r3 = -Gx - s + h.
multbymG(r3, x);
for (i = 0; i < 36; i++)
  r3[i] += -s[i] + h[i];
```

- Computing single entry in factorization:

```
L[265] = (- L[254]*v[118] - L[255]*v[119] - L[256]*v[120] - L[257]*v[121]
          - L[258]*v[122] - L[259]*v[123] - L[260]*v[124] - L[261]*v[125]
          - L[262]*v[126] - L[263]*v[127] - L[264]*v[128])*d_inv[129];
```

- Parameter stuffing:

```
b[4] = params.A[4]*params.x_0[0] + params.A[9]*params.x_0[1]
      + params.A[14]*params.x_0[2] + params.A[19]*params.x_0[3]
      + params.A[24]*params.x_0[4];
```

Part IV: **Verification**

1. Computation speed
2. Reliability

Computation speeds

- Maximum execution time more relevant than average
- Test millions of problem instances to verify performance

Computation speeds: **Examples**

	Scheduling	*Battery*	*Suspension*
Variables	279	153	104
Constraints	465	357	165
CVX, Intel i7	4.2 s	1.3 s	2.6 s

Computation speeds: **Examples**

	Scheduling	*Battery*	*Suspension*
Variables	279	153	104
Constraints	465	357	165
CVX, Intel i7	4.2 s	1.3 s	2.6 s
CVXGEN, Intel i7	850 µs	360 µs	110 µs

Computation speeds: **Examples**

	Scheduling	*Battery*	*Suspension*
Variables	279	153	104
Constraints	465	357	165
CVX, Intel i7	4.2 s	1.3 s	2.6 s
CVXGEN, Intel i7	850 µs	360 µs	110 µs
CVXGEN, Atom	7.7 ms	4.0 ms	1.0 ms

Reliability testing

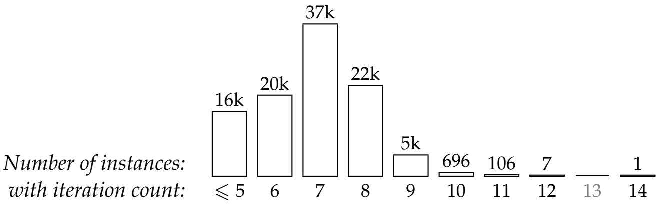
- Analyzed millions of instances from many problem families
- Goal: tune algorithm for total reliability, high speed
- Investigated:
  - **Algorithms**: primal-barrier, primal-dual, primal-dual with Mehrotra
  - **Initialization methods** including two-phase, infeasible-start, least-squares
  - **Regularization** and iterative refinement
  - **Algebra**: dense, library-based, sparse, flat; all with different solution methods
  - **Code generation**, using profiling to compare strategies
  - **Compiler integration**, using profiling and disassembly

Reliability testing: **Example**

- Computation time proportional to iteration count
- Thus, simulate many instances and record iteration count
- Example: ℓ_1-norm minimization with box constraints

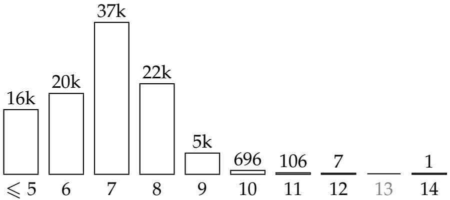
Reliability testing: **Example**

- Computation time proportional to iteration count
- Thus, simulate many instances and record iteration count
- Example: ℓ_1-norm minimization with box constraints
- Iteration count with default settings:

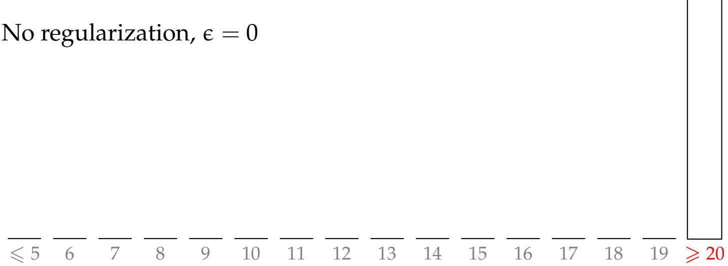


Reliability testing: **No KKT regularization**

- Default regularization, $\epsilon = 10^{-7}$

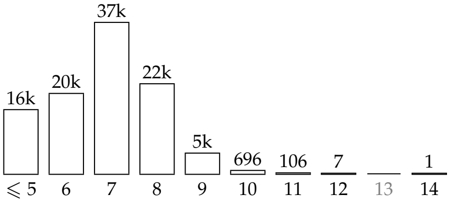


- No regularization, $\epsilon = 0$

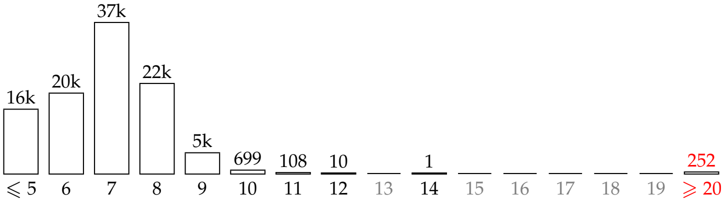


Reliability testing: **Decreased KKT regularization**

- Default regularization, $\epsilon = 10^{-7}$

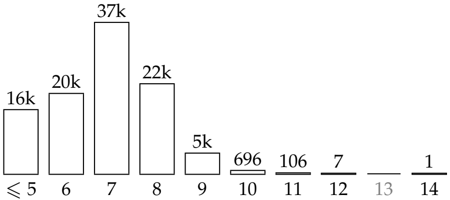


- Decreased regularization, $\epsilon = 10^{-11}$

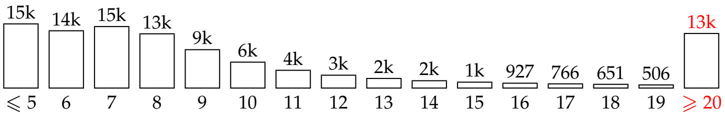


Reliability testing: **Increased KKT regularization**

- Default regularization, $\epsilon = 10^{-7}$

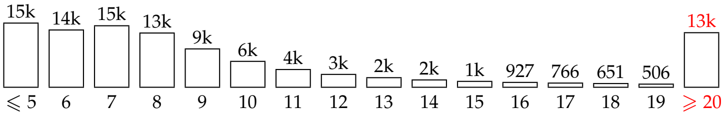


- Increased regularization, $\epsilon = 10^{-2}$

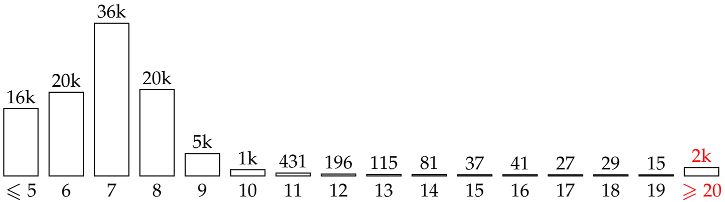


Reliability testing: **Iterative refinement**

- Default of 1 iterative refinement step, with $\epsilon = 10^{-2}$



- Increased to 10 iterative refinement steps, with $\epsilon = 10^{-2}$



Reliability testing: **Summary**

- Regularization and iterative refinement allow reliable solvers
- Iteration count relatively insensitive to parameters

Part V: **Final notes**

1. Conclusions
2. Contributions
3. Extensions
4. Publications

Conclusions

Contributions

- Framework for embedded convex optimization
- Design and demonstration of reliable algorithms
- First application of code generation to convex optimization

CVXGEN

- Fastest solvers ever written
- Already in use

Extensions

- Blocking, for larger problems
- More general convex families
- Different hardware

Publications

- CVXGEN: A Code Generator for Embedded Convex Optimization, J. Mattingley and S. Boyd, *Optimization and Engineering*, 2012
- Receding Horizon Control: Automatic Generation of High-Speed Solvers, J. Mattingley, Y. Wang and S. Boyd, *IEEE Control Systems Magazine*, 2011
- Real-Time Convex Optimization in Signal Processing, J. Mattingley and S. Boyd, *IEEE Signal Processing Magazine*, 2010
- Automatic Code Generation for Real-Time Convex Optimization, J. Mattingley and S. Boyd, chapter in *Convex Optimization in Signal Processing and Communications*, Cambridge University Press, 2009