Code Generation for Embedded Convex Optimization

Jacob Mattingley
Stanford University

October 2010
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**

```
Problem instance → General solver → $x^*$
```

```
Problem family description → Code generator → Source code → Compiler → Custom solver
```

```
Problem instance → Embedded solver → $x^*$
```

Part I: Introduction
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

```c
1 # Welcome to cvxgen.
2 # Here's a sample problem to get you started.
3
dimensions
4 n = 10
5 end
6
7 parameters
8 A (8,n)
9 b (8)
10 c (n)
11 end
12
13 variables
14 x (n)
15 end
16
17 minimize
18 c'*x + norm1(x)
19 subject to
20 A*x == b
21 x >= -1
22 end
```
CVXGEN: Automatic checking

```latex
1 \# Welcome to cvxgen.
2 \# Here's a sample problem to get you started.
3
4 dimensions
5 n = 10
6 end
7
8 parameters
9 A (8,n)
10 b (8)
11 c (n)
12 end
13
14 variables
15 x (n)
16 end
17
18 minimize
19 c'\times + \text{norm1}(x) - (1/10)*\text{norminf}(x)
20 subject to
21 A\times = b
22 x >= -1
23 end
```

19. objective must be convex.
CVXGEN: Formatted problem statement

Problem statement
minimize $c^T x + \|x\|_1$
subject to $Ax = b$
$x \geq -1$

Parameters
$A \in \mathbb{R}^{8 \times 10}, b \in \mathbb{R}^8, c \in \mathbb{R}^{10}$

Optimization variables
$x \in \mathbb{R}^{10}$
**CVXGEN: Single-button code generation**

![CVXGEN Interface](cvxgen.png)

<table>
<thead>
<tr>
<th>PROBLEM</th>
<th>CODEGEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>edit</td>
<td>generate C</td>
</tr>
<tr>
<td>view</td>
<td>matlab</td>
</tr>
<tr>
<td>std form</td>
<td>CODE INFO</td>
</tr>
<tr>
<td></td>
<td>statistics</td>
</tr>
<tr>
<td></td>
<td>kkt sparsity</td>
</tr>
<tr>
<td></td>
<td>OTHER OUTPUT</td>
</tr>
<tr>
<td></td>
<td>latex spec</td>
</tr>
<tr>
<td></td>
<td>latex math</td>
</tr>
<tr>
<td></td>
<td>cvx</td>
</tr>
<tr>
<td></td>
<td>cvxmod</td>
</tr>
<tr>
<td></td>
<td>OTHER TOOLS</td>
</tr>
<tr>
<td></td>
<td>user's guide</td>
</tr>
<tr>
<td></td>
<td>report a bug</td>
</tr>
</tbody>
</table>

**Problem size**

Your problem has 306 non-zero KKT matrix entries, which is relatively few. Code generation should be relatively fast.

(cvxgen is best for optimization problems with up to around 2000 entries.)

**Code generation status**

You have not generated code for this problem.

[Generate code](cvxgen.png)
**CVXGEN: Completed code generation**

### Problem size
Your problem has 306 non-zero KKT matrix entries, which is relatively few. Code generation should be relatively fast.
(cvxgen is best for optimization problems with up to around 2000 entries.)

### Code generation status
You generated code a moment ago. The code matches the problem statement.

#### Generated files

<table>
<thead>
<tr>
<th>File</th>
<th>Status</th>
<th>Preview</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvxgen.zip</td>
<td>complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cvxgen.tar.gz</td>
<td>complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Makefile</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>csolve.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>csolve.m</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>cvxsolve.m</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>ldl.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>make_csolve.m</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>matrix_support.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>solver.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>solver.h</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>testsolver.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
<tr>
<td>util.c</td>
<td>complete</td>
<td>preview</td>
<td></td>
</tr>
</tbody>
</table>

*Sizes in bytes: cvxgen.zip = 26 k, cvxgen.tar.gz = 23 k, Makefile = 1 k, csolve.c = 6 k, csolve.m = 1 k, cvxsolve.m = 1 k, ldl.c = 89 k, make_csolve.m = 1 k, matrix_support.c = 8 k, solver.c = 8 k, solver.h = 4 k, testsolver.c = 5 k, util.c = 3 k*
CVXGEN: Fast, problem-specific code
CVXGEN: Automatic problem transformations

KKT matrix
88x88; 306 non-zeros. 10 variables, transformed to 20 in the solver.

Minimization objective
\[ \frac{1}{2} \begin{bmatrix} t_{01} \\ x \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t_{01} \\ x \end{bmatrix} + \begin{bmatrix} 1 \\ c \end{bmatrix}^T \begin{bmatrix} t_{01} \\ x \end{bmatrix} \]

Equality constraint
\[ \begin{bmatrix} 0 & A \\ \end{bmatrix} \begin{bmatrix} t_{01} \\ x \end{bmatrix} = \begin{bmatrix} b \end{bmatrix} \]

8x20; 80 nonzeros. Sparsity diagram follows.
CVXGEN: Automatically generated Matlab interface

Step 1: Download the build script

Follow these instructions to download and build a Matlab mex solver.

You only need this step once, to put cvxgen.m in your current directory or Matlab path.

urlwrite('http://cvxgen.stanford.edu/download/cvxgen.m', 'cvxgen.m');

Step 2: Download custom code for this problem

Use this code for one-step download and build of a custom mex solver in Matlab.

cvxgen(368256)
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 language
- Solve problems in canonical form: with variable $x \in \mathbb{R}^n$,

  $\minimize \frac{1}{2} x^T Q x + q^T x$
  subject to $G x \leq h, \quad A x = b$

- 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

Part III: Techniques
Transformation to canonical form: Example

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

  \[
  \begin{align*}
  \text{minimize} & \quad x^TQx + c^Tx + \alpha \|y\|_1 \\
  \text{subject to} & \quad A(x - b) \leq 2y
  \end{align*}
  \]

- After epigraphical expansion, with new variable $t$:

  \[
  \begin{align*}
  \text{minimize} & \quad x^TQx + c^Tx + \alpha 1^T t \\
  \text{subject to} & \quad A(x - b) \leq 2y, \quad -t \leq y \leq t
  \end{align*}
  \]

- After reshaping variables and parameters into standard form:

  \[
  \begin{align*}
  \text{minimize} & \quad \begin{bmatrix} x \\ y \\ t \end{bmatrix}^T \begin{bmatrix} Q & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} + \begin{bmatrix} c \\ \alpha 1 \\ 0 \end{bmatrix}^T \begin{bmatrix} x \\ y \\ t \end{bmatrix} \\
  \text{subject to} & \quad \begin{bmatrix} A & -2I & 0 \\ 0 & -I & -I \\ 0 & I & I \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \leq \begin{bmatrix} Ab \\ 0 \\ 0 \end{bmatrix}
  \end{align*}
  \]
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

\[
\begin{bmatrix}
Q & 0 & G^T & A^T \\
0 & Z & S & 0 \\
G & I & 0 & 0 \\
A & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\Delta x^{\text{aff}} \\
\Delta s^{\text{aff}} \\
\Delta z^{\text{aff}} \\
\Delta y^{\text{aff}}
\end{bmatrix} =
\begin{bmatrix}
-(A^T y + G^T z + P x + q) \\
-S z \\
-(G x + s - h) \\
-(A x - b)
\end{bmatrix}.
\]

3. Compute centering-plus-corrector direction by solving

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

with

\[
\mu = s^T z / p \\
\sigma = \left( (s + \alpha \Delta s^{\text{aff}})^T (z + \alpha \Delta z^{\text{aff}}) / (s^T z) \right)^3 \\
\alpha = \sup\{\alpha \in [0, 1] \mid s + \alpha \Delta s^{\text{aff}} \geq 0, \ z + \alpha \Delta z^{\text{aff}} \geq 0\}.
\]
Algorithm (continued)

4. Combine the updates with
\[
\Delta x = \Delta x^{\text{aff}} + \Delta x^{\text{cc}} \quad \Delta s = \Delta s^{\text{aff}} + \Delta s^{\text{cc}}
\]
\[
\Delta y = \Delta y^{\text{aff}} + \Delta y^{\text{cc}} \quad \Delta z = \Delta z^{\text{aff}} + \Delta z^{\text{cc}}
\]

5. Find
\[
\alpha = \min\{1, \ 0.99 \sup\{\alpha \geq 0 \mid s + \alpha \Delta s \geq 0, \ z + \alpha \Delta z \geq 0\}\},
\]
and update
\[
x := x + \alpha \Delta x \quad s := s + \alpha \Delta s
\]
\[
y := y + \alpha \Delta y \quad z := z + \alpha \Delta z
\]
Solving KKT system

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

\[ K = \begin{bmatrix}
    Q & 0 & G^T & A^T \\
    0 & S^{-1}Z & I & 0 \\
    G & I & 0 & 0 \\
    A & 0 & 0 & 0
\end{bmatrix} \]

- 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 $\tilde{K}$ instead
- Choose regularization constant $\epsilon > 0$, then instead factor:

$$
P \begin{pmatrix}
  Q & 0 & G^T & A^T \\
  0 & S^{-1}Z & I & 0 \\
  G & I & 0 & 0 \\
  A & 0 & 0 & 0
\end{pmatrix} + \begin{pmatrix}
  \epsilon I & 0 \\
  0 & -\epsilon I
\end{pmatrix}

P^T = P\tilde{K}P^T = LDL^T

- $\tilde{K}$ 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 $\tilde{K}$.

  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 \( \ell \) with different residuals \( r \), of
  \[
  K\ell = r
  \]
- Instead, solve
  \[
  \ell = \tilde{K}^{-1}r = P^T L^{-T} D^{-1} L^{-1} Pr
  \]
- 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 \( \tilde{K}^{-1} \approx K^{-1} \)

- Use iterative refinement:
  
  Solve \( \tilde{K}\ell^{(0)} = r \).

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

  1. Compute approximate correction by solving \( \tilde{K}\delta\ell^{(0)} = r - K\ell^{(0)} \).
  2. Update iterate \( \ell^{(1)} = \ell^{(0)} + \delta\ell^{(0)} \).
  3. Repeat until \( \ell^{(k)} \) is sufficiently accurate.

- Iterative refinement with \( \tilde{K} \) 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 $\hat{D}_{ii} \neq D_{ii}$, due to numerical errors
- Therefore, given sign $\xi_i$ of $D_{ii}$, use
  \[ D_{ii} = \xi_i((\xi_i\hat{D}_{ii})_+ + \epsilon) \]
- 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]

```c
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:

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

- Computing single entry in factorization:

```c
```

- Parameter stuffing:

```c
    + 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

<table>
<thead>
<tr>
<th></th>
<th>Scheduling</th>
<th>Battery</th>
<th>Suspension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>279</td>
<td>153</td>
<td>104</td>
</tr>
<tr>
<td>Constraints</td>
<td>465</td>
<td>357</td>
<td>165</td>
</tr>
<tr>
<td>CVX, Intel i7</td>
<td>4.2 s</td>
<td>1.3 s</td>
<td>2.6 s</td>
</tr>
<tr>
<td>CVXGEN, Intel i7</td>
<td>850 µs</td>
<td>360 µs</td>
<td>110 µs</td>
</tr>
<tr>
<td>CVXGEN, Atom</td>
<td>7.7 ms</td>
<td>4.0 ms</td>
<td>1.0 ms</td>
</tr>
</tbody>
</table>

Part IV: Verification
**Computation speeds: Examples**

<table>
<thead>
<tr>
<th></th>
<th>Scheduling</th>
<th>Battery</th>
<th>Suspension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>279</td>
<td>153</td>
<td>104</td>
</tr>
<tr>
<td>Constraints</td>
<td>465</td>
<td>357</td>
<td>165</td>
</tr>
<tr>
<td>CVX, Intel i7</td>
<td>4.2 s</td>
<td>1.3 s</td>
<td>2.6 s</td>
</tr>
<tr>
<td>CVXGEN, Intel i7</td>
<td>850 µs</td>
<td>360 µs</td>
<td>110 µs</td>
</tr>
</tbody>
</table>
## Computation speeds: Examples

<table>
<thead>
<tr>
<th></th>
<th>Scheduling</th>
<th>Battery</th>
<th>Suspension</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
<td>279</td>
<td>153</td>
<td>104</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>465</td>
<td>357</td>
<td>165</td>
</tr>
<tr>
<td>CVX, Intel i7</td>
<td>4.2 s</td>
<td>1.3 s</td>
<td>2.6 s</td>
</tr>
<tr>
<td>CVXGEN, Intel i7</td>
<td>850 µs</td>
<td>360 µs</td>
<td>110 µs</td>
</tr>
<tr>
<td>CVXGEN, Atom</td>
<td>7.7 ms</td>
<td>4.0 ms</td>
<td>1.0 ms</td>
</tr>
</tbody>
</table>

Part IV: Verification
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: $\ell_1$-norm minimization with box constraints
- Iteration count with default settings:

```
Number of instances: with iteration count: ≤ 5 6 7 8 9 10 11 12 13 14
```

- $16k$ instances with iteration count $\leq 5$
- $20k$ instances with iteration count $6$
- $37k$ instances with iteration count $7$
- $22k$ instances with iteration count $8$
- $5k$ instances with iteration count $9$
- $696$ instances with iteration count $10$
- $106$ instances with iteration count $11$
- $7$ instances with iteration count $12$
- $13$ instances with iteration count $13$
- $1$ instance with iteration count $14$
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, $\epsilon = 10^{-11}$

Part IV: Verification
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
5. Acknowledgements
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, manuscript


