Real-Time Embedded Convex Optimization

Stephen Boyd

joint work with Michael Grant, Jacob Mattingley, Yang Wang

Electrical Engineering Department, Stanford University

ISMP 2009
Outline

- Real-time embedded convex optimization
- Examples
- Parser/solvers for convex optimization
- Code generation for real-time embedded convex optimization
Embedded optimization

• embed solvers in real-time applications

• *i.e.*, solve an optimization problem at each time step

• used now for applications with hour/minute time-scales
  – process control
  – supply chain and revenue ‘management’
  – trading
What’s new

embedded optimization at \textit{millisecond/microsecond time-scales}
Applications

- real-time resource allocation
  - update allocation as objective, resource availabilities change
- signal processing
  - estimate signal by solving optimization problem over sliding window
  - replace least-squares estimates with robust (Huber, $\ell_1$) versions
  - re-design (adapt) coefficients as signal/system model changes
- control
  - closed-loop control via rolling horizon optimization
  - real-time trajectory planning
- all of these done now, on long (minutes or more) time scales
  but could be done on millisecond/microsecond time scales
Outline

- Real-time embedded convex optimization
- Examples
- Parser/solvers for convex optimization
- Code generation for real-time embedded convex optimization
Grasp force optimization

- choose \( K \) grasping forces on object to
  - resist external wrench (force and torque)
  - respect friction cone constraints
  - minimize maximum grasp force

- convex problem (second-order cone program or SOCP):

\[
\text{minimize} \quad \max_i \| \mathbf{f}^{(i)} \|_2 \\
\text{subject to} \quad \sum_i Q^{(i)} \mathbf{f}^{(i)} = \mathbf{f}^{\text{ext}} \\
\sum_i p^{(i)} \times (Q^{(i)} \mathbf{f}^{(i)}) = \mathbf{\tau}^{\text{ext}} \\
\mu_i f_z^{(i)} \geq \left( f_x^{(i)} + f_y^{(i)} \right)^{1/2}
\]

variables \( \mathbf{f}^{(i)} \in \mathbb{R}^3, \ i = 1, \ldots, K \) (contact forces)
Example
Grasp force optimization solve times

- example with $K = 5$ fingers (grasp points)
- reduces to SOCP with 15 vars, 6 eqs, 5 3-dim SOCs
- custom code solve time: 50µs (SDPT3: 100ms)
Robust Kalman filtering

- estimate state of a linear dynamical system driven by IID noise
- sensor measurements have occasional outliers (failures, jamming, . . . )
- model:
  \[ x_{t+1} = Ax_t + w_t, \quad y_t =Cx_t + v_t + z_t \]
  \(- w_t \sim \mathcal{N}(0, W), \quad v_t \sim \mathcal{N}(0, V) \)
  \(- z_t \text{ is sparse}; \text{ represents outliers, failures, . . . } \)

- (steady-state) Kalman filter (for case \( z_t = 0 \)):
  \(- \text{ time update: } \hat{x}_{t+1|t} = A\hat{x}_{t|t} \)
  \(- \text{ measurement update: } \hat{x}_{t|t} = \hat{x}_{t|t-1} + L(y_t - C\hat{x}_{t|t-1}) \)

- we’ll replace measurement update with robust version to handle outliers
Measurement update via optimization

• standard KF: $\hat{x}_{t|t}$ is solution of quadratic problem

\[
\begin{align*}
\text{minimize} & \quad v^TV^{-1}v + (x - \hat{x}_{t|t-1})^T\Sigma^{-1}(x - \hat{x}_{t|t-1}) \\
\text{subject to} & \quad y_t = Cx + v
\end{align*}
\]

with variables $x$, $v$ (simple analytic solution)

• robust KF: choose $\hat{x}_{t|t}$ as solution of convex problem

\[
\begin{align*}
\text{minimize} & \quad v^TV^{-1}v + (x - \hat{x}_{t|t-1})^T\Sigma^{-1}(x - \hat{x}_{t|t-1}) + \lambda\|z\|_1 \\
\text{subject to} & \quad y_t = Cx + v + z
\end{align*}
\]

with variables $x$, $v$, $z$ (requires solving a QP)
Example

- 50 states, 15 measurements
- with prob. 5%, measurement components replaced with $(y_t)_i = (v_t)_i$
- so, get a flawed measurement (i.e., $z_t \neq 0$) every other step (or so)
State estimation error

\[ \| x - \hat{x}_{t|t} \|_2 \] for KF (red); robust KF (blue); KF with \( z = 0 \) (gray)
Robust Kalman filter solve time

• robust KF requires solution of QP with 95 vars, 15 eqs, 30 ineqs

• automatically generated code solves QP in 120 \( \mu \text{s} \) (SDPT3: 120 ms)

• standard Kalman filter update requires 10 \( \mu \text{s} \)
Linearizing pre-equalizer

- linear dynamical system with input saturation

- we'll design pre-equalizer to compensate for saturation effects
Linearizing pre-equalizer

- goal: minimize error $e$ (say, in mean-square sense)
- pre-equalizer has $T$ sample look-ahead capability
• system: \[ x_{t+1} = Ax_t + B \text{sat}(v_t), \quad y_t = Cx_t \]

• (linear) reference system: \[ x_{t+1}^{\text{ref}} = Ax_t^{\text{ref}} + Bu_t, \quad y_t^{\text{ref}} = Cx_t^{\text{ref}} \]

• \( e_t = Cx_t^{\text{ref}} - Cx_t \)

• state error \( \tilde{x}_t = x_t^{\text{ref}} - x_t \) satisfies

\[
\tilde{x}_{t+1} = A\tilde{x}_t + B(u_t - v_t), \quad e_t = C\tilde{x}_t
\]

• to choose \( v_t \), solve QP

\[
\begin{align*}
\text{minimize} & \quad \sum_{\tau=t}^{t+T} e_\tau^2 + \tilde{x}_{t+T+1}^T P \tilde{x}_{t+T+1} \\
\text{subject to} & \quad \tilde{x}_{\tau+1} = A\tilde{x}_\tau + B(u_\tau - v_\tau), \quad e_\tau = C\tilde{x}_\tau, \quad \tau = t, \ldots, t + T \\
& \quad |v_\tau| \leq 1, \quad \tau = t, \ldots, t + T 
\end{align*}
\]

\( P \) gives final cost; obvious choice is output Grammian
Example

• state dimension $n = 3$; $h$ decays in around 35 samples

• pre-equalizer look-ahead $T = 15$ samples

• input $u$ random, saturates ($|u_t| > 1$) 20% of time
**Outputs**

desired (black), no compensation (red), equalized (blue)
Errors

no compensation (red), with equalization (blue)
Inputs

no compensation (red), with equalization (blue)
Linearizing pre-equalizer solve time

- pre-equalizer problem reduces to QP with 96 vars, 63 eqs, 48 ineqs
- automatically generated code solves QP in $600\mu s$ (SDPT3: 310ms)
Constrained linear quadratic stochastic control

- linear dynamical system: \( x_{t+1} = Ax_t + Bu_t + w_t \)
  - \( x_t \in \mathbb{R}^n \) is state; \( u_t \in \mathcal{U} \subset \mathbb{R}^m \) is control input
  - \( w_t \) is IID zero mean disturbance

- \( u_t = \phi(x_t) \), where \( \phi : \mathbb{R}^n \rightarrow \mathcal{U} \) is (state feedback) policy

- objective: minimize average expected stage cost (\( Q \geq 0, \ R > 0 \))

\[
J = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T-1} \mathbb{E} \left( x_t^T Q x_t + u_t^T R u_t \right)
\]

- constrained LQ stochastic control problem: choose \( \phi \) to minimize \( J \)
Constrained linear quadratic stochastic control

- optimal policy has form

\[
\phi(z) = \arg\min_{v \in U} \{ v^T R v + \mathbb{E} \, V(Az + Bv + w_t) \}
\]

where \( V \) is Bellman function

- but \( V \) is hard to find/describe except when \( U = \mathbb{R}^m \)
  (in which case \( V \) is quadratic)

- many heuristic methods give suboptimal policies, \textit{e.g.}

  - projected linear control
  - control-Lyapunov policy
  - model predictive control, certainty-equivalent planning
Control-Lyapunov policy

- also called approximate dynamic programming, horizon-1 model predictive control
- CLF policy is

\[ \phi_{\text{clf}}(z) = \arg\min_{v \in \mathcal{U}} \{ v^T R v + \mathbb{E} V_{\text{clf}}(A z + B v + w_t) \} \]

where \( V_{\text{clf}} : \mathbb{R}^n \to \mathbb{R} \) is the control-Lyapunov function
- evaluating \( u_t = \phi_{\text{clf}}(x_t) \) requires solving an optimization problem at each step
- many tractable methods can be used to find a good \( V_{\text{clf}} \)
- often works really well
Quadratic control-Lyapunov policy

- assume
  - polyhedral constraint set: \( U = \{ v \mid Fv \leq g \}, \ g \in \mathbb{R}^k \)
  - quadratic control-Lyapunov function: \( V_{cl}(z) = z^T P z \)

- evaluating \( u_t = \phi_{cl}(x_t) \) reduces to solving QP

\[
\begin{aligned}
\text{minimize} & \quad v^T R v + (Az + Bv)^T P (Az + Bv) \\
\text{subject to} & \quad Fv \leq g
\end{aligned}
\]
Control-Lyapunov policy evaluation times

- $t_{\text{clf}}$: time to evaluate $\phi_{\text{clf}}(z)$
- $t_{\text{lin}}$: linear policy $\phi_{\text{lin}}(z) = Kz$
- $t_{\text{kf}}$: Kalman filter update
- (SDPT3 times around $1000 \times$ larger)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$k$</th>
<th>$t_{\text{clf}}$ (μs)</th>
<th>$t_{\text{lin}}$ (μs)</th>
<th>$t_{\text{kf}}$ (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>5</td>
<td>10</td>
<td>35</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>15</td>
<td>30</td>
<td>85</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>20</td>
<td>67</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>1000</td>
<td>30</td>
<td>60</td>
<td>298</td>
<td>130</td>
<td>8300</td>
</tr>
</tbody>
</table>
Outline

- Real-time embedded convex optimization
- Examples
- Parser/solvers for convex optimization
- Code generation for real-time embedded convex optimization
Parser/solvers for convex optimization

- specify convex problem in natural form
  - declare optimization variables
  - form convex objective and constraints using a specific set of atoms and calculus rules (disciplined convex programming)

- problem is convex-by-construction

- easy to parse, automatically transform to standard form, solve, and transform back

- implemented using object-oriented methods and/or compiler-compilers

- huge gain in productivity (rapid prototyping, teaching, research ideas)
Example: cvx

- parser/solver written in Matlab

- convex problem, with variable $x \in \mathbb{R}^n$; $A$, $b$, $\lambda$, $F$, $g$ constants

\[
\begin{align*}
\text{minimize} & \quad \|Ax - b\|_2 + \lambda \|x\|_1 \\
\text{subject to} & \quad Fx \leq g
\end{align*}
\]

- cvx specification:

```matlab
cvx_begin
  variable x(n) % declare vector variable
  minimize (norm(A*x-b,2) + lambda*norm(x,1))
  subject to   F*x <= g
cvx_end
```
when cvx processes this specification, it

- verifies convexity of problem
- generates equivalent cone problem (here, an SOCP)
- solves it using SDPT3 or SeDuMi
- transforms solution back to original problem

the cvx code is easy to read, understand, modify
The same example, transformed by ‘hand’

transform problem to SOCP, call SeDuMi, reconstruct solution:

% Set up big matrices.
[m,n] = size(A); [p,n] = size(F);
AA = [speye(n), -speye(n), speye(n), sparse(n,p+m+1); ...
    F, sparse(p,2*n), speye(p), sparse(p,m+1); ...
    A, sparse(m,2*n+p), speye(m), sparse(m,1)];
bb = [zeros(n,1); g; b];
cc = [zeros(n,1); gamma*ones(2*n,1); zeros(m+p,1); 1];
K.f = m; K.l = 2*n+p; K.q = m + 1; % specify cone
xx = sedumi(AA, bb, cc, K); % solve SOCP
x = x(1:n); % extract solution
Outline

- Real-time embedded convex optimization
- Examples
- Parser/solvers for convex optimization
- Code generation for real-time embedded convex optimization
General vs. embedded solvers

• general solver (say, for QP)
  – handles single problem instances with any dimensions, sparsity pattern
  – typically optimized for large problems
  – must deliver high accuracy
  – variable execution time: stops when tolerance achieved

• embedded solver
  – solves many instances of the same problem family (dimensions, sparsity pattern) with different data
  – solves small or smallish problems
  – can deliver lower (application dependent) accuracy
  – often must satisfy hard real-time deadline
Embedded solvers

- (if a general solver works, use it)

- otherwise, develop custom code
  - by hand
  - automatically via code generation

- can exploit known sparsity pattern, data ranges, required tolerance at solver code development time

- we’ve had good results with interior-point methods; other methods (e.g., active set, first order) might work well too

- typical speed-up over (efficient) general solver: $100–10000 \times$
Convex optimization solver generation

- specify convex problem **family** in natural form, via disciplined convex programming
  - declare optimization variables, parameters
  - form convex objective and constraints using a specific set of atoms and calculus rules
- code generator
  - analyzes problem structure (dimensions, sparsity, . . . )
  - chooses elimination ordering
  - generates solver code for specific problem family
- idea:
  - spend (perhaps much) time generating code
  - save (hopefully much) time solving problem instances
**Parser/solver vs. code generation**

![Diagram showing the comparison between parser/solver and code generation processes.](image)

- **Problem instance** → **Parser/solver** → **$x^*$**
- **Problem family description** → **Code generator** → **Source code** → **Compiler** → **Custom solver**
- **Problem instance** → **Custom solver** → **$x^*$**
Example: cvxmod

- written in Python
- QP family, with variable $x \in \mathbb{R}^n$, parameters $P$, $q$, $g$, $h$

\[
\begin{align*}
\text{minimize} & \quad x^T P x + q^T x \\
\text{subject to} & \quad G x \leq h, \quad A x = b
\end{align*}
\]

- cvxmod specification:

```python
A = matrix(...); b = matrix(...) 
P = param('P', n, n, psd=True); q = param('q', n) 
G = param('G', m, n); h = param('h', m) 
x = optvar('x', n) 
qupfam = problem(minimize(quadform(x, P) + tp(q)*x), 
                 [G*x <= h, A*x == b])
```
cvxmod code generation

- generate solver for problem family `qpfam` with
  
  `qpfam.codegen()`

- output includes `qpfam/solver.c` and ancillary files

- solve instance with (C function call)

  `status = solve(params, vars, work);`
Using *cvxmod* generated code

```c
#include "solver.h"
int main(int argc, char **argv) {
    // Initialize structures at application start-up.
    Params params = init_params();
    Vars vars = init_vars();
    Workspace work = init_work(vars);
    // Enter real-time loop.
    for (;;) {
        update_params(params);
        status = solve(params, vars, work);
        export_vars(vars);
    }
}
```
cvxmod code generator

(preliminary implementation)

- handles problems transformable to QP
- primal-dual interior-point method with iteration limit
- direct $LDL^T$ factorization of KKT matrix
- (slow) method to determine variable ordering (at code generation time)
- explicit factorization code generated
### Sample solve times for cvxmod generated code

<table>
<thead>
<tr>
<th>problem family</th>
<th>vars</th>
<th>constrs</th>
<th>SDPT3 (ms)</th>
<th>cvxmod (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>control1</td>
<td>140</td>
<td>190</td>
<td>250</td>
<td>0.4</td>
</tr>
<tr>
<td>control2</td>
<td>360</td>
<td>1080</td>
<td>1400</td>
<td>2.0</td>
</tr>
<tr>
<td>control3</td>
<td>1110</td>
<td>3180</td>
<td>3400</td>
<td>13.2</td>
</tr>
<tr>
<td>order_exec</td>
<td>20</td>
<td>41</td>
<td>490</td>
<td>0.05</td>
</tr>
<tr>
<td>net_utility</td>
<td>50</td>
<td>150</td>
<td>130</td>
<td>0.23</td>
</tr>
<tr>
<td>actuator</td>
<td>50</td>
<td>106</td>
<td>300</td>
<td>0.17</td>
</tr>
<tr>
<td>robust_kalman</td>
<td>95</td>
<td>45</td>
<td>120</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Conclusions

• can solve convex problems on **millisecond, microsecond** time scales
  – (using existing algorithms, but not using existing codes)
  – there should be many applications

• parser/solvers make rapid prototyping easy

• new code generation methods yield solvers that
  – are extremely fast, even competitive with ‘analytical methods’
  – can be embedded in real-time applications
References

- *Automatic Code Generation for Real-Time Convex Optimization* (Mattingley, Boyd)
- *Fast Evaluation of Quadratic Control-Lyapunov Policy* (Wang, Boyd)
- *Fast Model Predictive Control Using Online Optimization* (Wang, Boyd)
- cvx (Grant, Boyd, Ye)
- cvxmod (Mattingley, Boyd)

all available on-line, but cvxmod code gen not yet ready for prime-time