Embedded Convex Optimization for Control

Stephen Boyd Akshay Agrawal Shane Barratt

Stanford University

December 14, 2020

About this talk

- ideas, sloppy math
- opinions (some controversial)
- covers lots of work done by others with no explicit attribution
- sadly, no fun videos or cool examples

Outline

Convex optimization control policies

Why?

Tuning

Technology

Conclusions

Convex optimization control policies

- many control policies are based on solving a convex optimization problem
- we call these *convex optimization control policies* (COCPs)
- examples
  - linear quadratic regulator (LQR), Kalman filter (KF)
  - convex control
  - approximate dynamic programming (ADP)
  - model predictive control (MPC) / receding horizon control (RHC)
  - single and multiple period (financial) trading
  - actuator allocation
  - real-time resource allocation
- a few of these are analytically solvable; we focus on the others

Traditional quadratic control

- dynamics $x_{t+1} = Ax_t + Bu_t + w_t$, w_t IID zero mean
- convex quadratic stage cost $x^T Qx + u^T Ru$
- minimize expected average stage cost
- optimal (LQR) policy has form

  $$u_t = \operatorname*{argmin}_u \left(u^T Ru + (Ax_t + Bu)^T P(Ax_t + Bu) \right)$$

  i.e., find u_t by minimizing a convex quadratic function
- analytically solve to get $u_t = Kx_t$

Convex control via dynamic programming

- dynamics $x_{t+1} = f(x_t, u_t, \omega_t)$, ω_t IID, f affine in x, u
- stage cost g convex in x, u
- minimize expected average stage cost
- optimal policy is

$$u_t = \operatorname*{argmin}_u \mathsf{E}\left(g(x_t, u, \omega_t) + V(f(x_t, u, \omega_t))\right)$$

- V is (convex) value or Bellman function
- u_t obtained by minimizing a convex function

Approximate dynamic programming

- use dynamic programming form with *approximate* value function
- ADP policy is

$$u_t = \operatorname*{argmin}_u \mathbf{E}\left(g(x_t, u, \omega_t) + \hat{V}(f(x_t, u, \omega_t))\right)$$

- $\hat{V}$ is (convex) approximate or surrogate value function
- $\hat{V}$ chosen to
  - capture general shape of V
  - make optimization problem tractable, *i.e.*, convex in u
- requires only that f is affine in u, g is convex in u

Model predictive control

- dynamics function f affine in x, u, stage cost g convex in x, u
- MPC policy: solve

$$\begin{array}{ll} \text{minimize} & \sum_{\tau=t}^{t+H} g(x_\tau, u_\tau, \hat{\omega}_{\tau|t}) \\ \text{subject to} & x_{\tau+1} = f(x_\tau, u_\tau, \hat{\omega}_{\tau|t}), \quad \tau = t, \ldots, t+H-1 \end{array}$$

  and take u_t as control
- x_t is given; $x_{t+1}, \ldots, x_{t+H}$ are variables
- $\hat{\omega}_{\tau|t}$ is *forecast* of ω_τ made at time t
- *plan* full trajectory x_τ, u_τ over $\tau = t, t+1, \ldots, t+H$; *use* only u_t

Multi-forecast model predictive control

- use *multiple forecasts* $\hat{\omega}^i_{\tau|t}$, $i = 1, \ldots, K$
- interpret as K different *scenarios* or *contingencies*
- MF-MPC policy: solve

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{K} \sum_{\tau=t}^{t+H} g(x^i_\tau, u^i_\tau, \hat{\omega}^i_{\tau|t}) \\ \text{subject to} & x^i_{\tau+1} = f(x^i_\tau, u^i_\tau, \hat{\omega}^i_{\tau|t}), \quad \tau = t, \ldots, t+H-1, \quad i = 1, \ldots K \\ & u^1_t = \cdots = u^K_t \end{array}$$

  and take u^1_t as control
- *plan* for all contingencies, but require first action to be the *same for all*

Single period trading

- w_t is (given, current) asset allocation weight in period t, $1^T w_t = 1$
- $\tilde{w}_t$ is post-trade allocation, chosen by maximizing

$$\alpha_t^T \tilde{w}_t - \gamma \tilde{w}_t^T \Sigma_t \tilde{w}_t - \phi_t^{\mathrm{hld}}(\tilde{w}_t) - \phi_t^{\mathrm{tc}}(\tilde{w}_t - w_t)$$

  (risk and cost-adjusted expected return) subject to $1^T \tilde{w}_t = 1$
- α_t is forecast return, Σ_t is return covariance, $\gamma > 0$ is risk aversion
- ϕ^{hld} and ϕ^{tc} are convex holding and transaction cost functions (can be $+\infty$ to encode constraints)
- readily extended to multi-period (MPC)

Actuator allocation

- higher level control policy produces desired forces and torques f_t
- *actuator allocation*: choose actuator values u_t by solving

$$\begin{array}{ll} \text{minimize} & g_t(u) + \lambda \|u - u_{t-1}\|_2^2 \\ \text{subject to} & u \in \mathcal{U}_t, \quad A_t u = f_t \end{array}$$

- g_t is convex cost function (fuel use, energy, ...)
- second objective term encourages smooth actuator values, $\lambda > 0$
- $\mathcal{U}_t$ is actuator constraint set
- A_t maps actuator values into net forces and torques

- gracefully handles actuator failure, degradation, varying effectiveness

Resource allocator

- m resources to be distributed across n agents or tasks
- $a_t \in \mathbf{R}_+^m$ is available resources
- action is resource allocation $u_t \in \mathbf{R}^{m \times n}$
- choose u_t by solving
$$\begin{array}{ll} \text{maximize} & U_t(u) \\ \text{subject to} & u \geq 0, \quad u\mathbf{1} \leq a_t \end{array}$$
- U_t is concave utility, usually separable across tasks

Convex optimization policy: General form

convex optimization control policy (COCP): action u_t is solution of

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

with variable u (and possibly others, not shown)

- f_i are convex in u
- x_t is the state or context
- $\theta \in \Theta$ are *parameters* that flavorize the policy

Outline

Convex optimization control policies

Why?

Tuning

Technology

Conclusions

Procedural versus declarative policies

- *procedural policy*:
  - designer explicitly specifies what to do in given context
  - *e.g.*, $u_t = -K_{\mathrm{P}} e_t - K_{\mathrm{I}} \sum_{\tau=0}^{t} e_\tau$
- *declarative policy*:
  - designer articulates what she wants and requires
  - and *lets the optimization solver figure out how to do it*

Advantages (non-controversial)

COCPs

- are interpretable; we understand exactly what they do
- respect constraints better than simple projection / clipping
- can incorporate (almost never active) safety constraints
- gracefully handle changing dynamics / availabilities / failures
- can be effectively tuned (more later)

a non-disadvantage:

- COCPs can be made fast, totally reliable, even division free in some cases

Advantages (possibly controversial)

- COCPs never do anything crazy, like characterize a stop sign as a banana
- parametrizing COCP is better than raw controller or policy
  (stated in LQR context since 1960)

Outline

Design flow

1. build high fidelity simulator, using real historical data, generative model, etc.
2. implement code that evaluates true performance objective(s)
3. choose a parametrized convex optimization based policy
4. tune the parameters until you're OK with the simulated performance

Traditional tuning / tweaking

- typically done by hand for a few parameters that scale objective terms
- the method:
  1. start with a reasonable value for θ
  2. simulate and evaluate performance objective
  3. update θ by hand (typically one parameter at a time)
  4. repeat until (happy $\|$ bored $\|$ out of time)
- alternative: fire up a derivative free method, then go to lunch

Auto-tuning

- compute $\nabla_\theta \mathcal{L}(\theta^k)$
- $\mathcal{L}$ is true performance objective evaluated via simulation
- update $\theta^{k+1} = \Pi_\Theta \left(\theta^k - t^k \nabla_\theta \mathcal{L}(\theta^k)\right)$

- $\mathcal{L}$ often not differentiable
- follow NN tradition and ignore
- use automatic differentiation to compute "∇" $\mathcal{L}(\theta^k)$

- θ can contain more than a few parameters
- use different test and validation simulations to avoid over-tuning

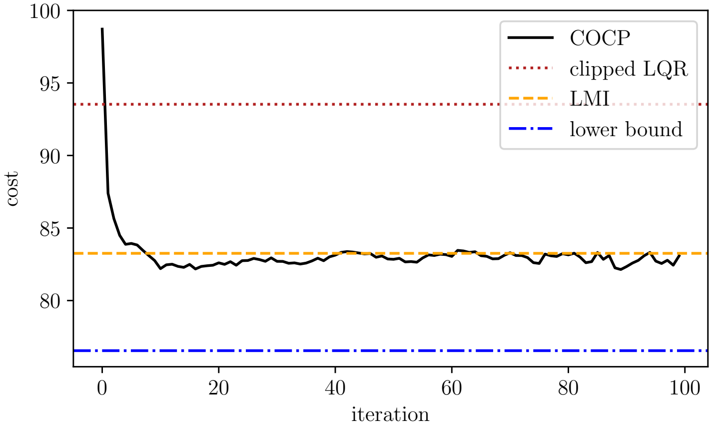
Example: ADP for box-constrained LQR

- $x_{t+1} = Ax_t + Bu_t + w_t$, $w_t \sim \mathcal{N}(0, I)$
- actuator limit $\|u_t\|_\infty \leq 1$
- cost is average value of $x_t^T Q x_t + u_t^T R u_t$
- ADP policy: u_t is solution of

$$\begin{array}{ll} \text{minimize} & u^T R u + \|\theta(Ax_t + Bu)\|_2^2 \\ \text{subject to} & \|u\|_\infty \leq 1 \end{array}$$

- we'll compare to clipped LQR and LMI-based upper- and lower-bounds

Auto-tuning ADP for box-constrained LQR



Example: Single period trading engine

- $w_t \in \mathbf{R}^7$ are weights on 7 ETFs
- post-trade allocation $\tilde{w}_t$ is solution of

$$\begin{array}{ll} \text{maximize} & \alpha_t^T w - \gamma_t w^T \Sigma_t w - \gamma_t^{\text{hld}} \mathbf{1}^T (w)_- - \gamma_t^{\text{tc}} \|w - w_t\|_1 \\ \text{subject to} & \mathbf{1}^T w = 1, \quad \|w\|_1 \leq 1.5, \quad w \leq 0.5 \end{array}$$

- α_t and Σ_t depend on VIX (volatility index) quintiles
- 15 parameters: $(\gamma, \gamma^{\text{hld}}, \gamma^{\text{tc}})$ for each of 5 VIX quintiles
- simulations on (realistic) log-normal returns conditioned on VIX index, 0.1% transaction costs, 0.02% shorting costs

Tuning objective

- Sharpe ratio: annualized return / annualized volatility
- drawdown at time t is $d_t = (h_t - v_t)/h_t = 1 - v_t/h_t$
  - v_t is portfolio value
  - $h_t = \max_{\tau=1,\ldots,t} v_\tau$ is previous high value

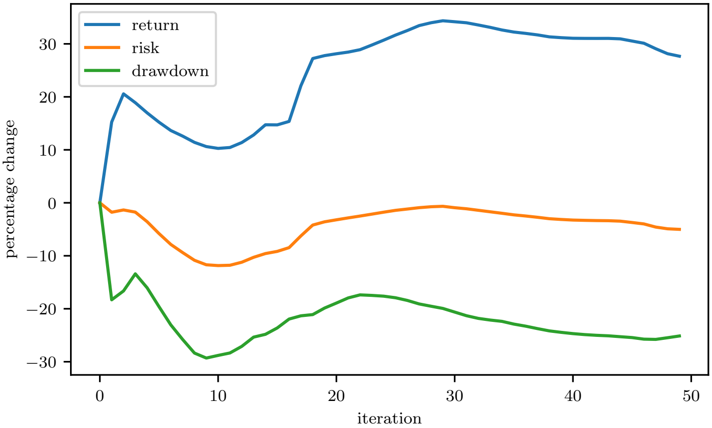
- tuning objective: maximize Sharpe ratio minus average drawdown %
- initialize with $\gamma = 5$ and true costs
- we'll compare to a policy that ignores VIX, uses common α and Σ

Tuning results

policy	return	volatility	Sharpe	drawdown	objective
common	9.2%	7.9%	1.2	2.6%	-1.4
initial	13.5%	7.1%	1.9	1.3%	0.6
tuned	17.3%	6.7%	2.6	1.0%	1.6

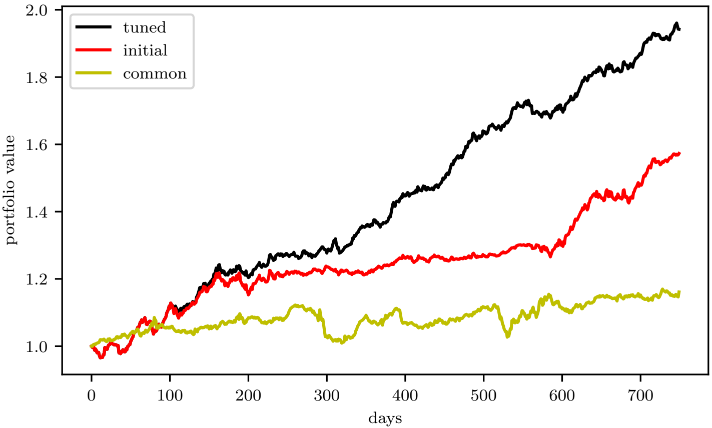
(average of eight 750-day simulations, not used for tuning)

Tuning progress



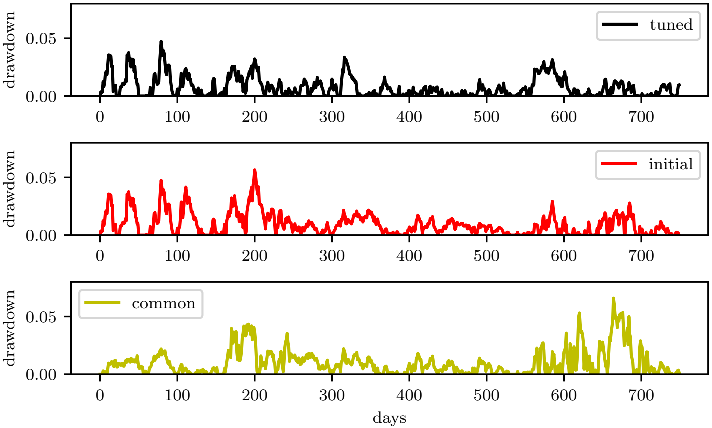
(average of eight 750-day simulations)

Wealth trajectory



(one simulation)

Drawdown



(one simulation)

Outline

Convex optimization control policies

Why?

Tuning

Technology

Conclusions

Domain specific languages for convex optimization

- DSLs make it easy to specify and solve convex problems
- grammar and semantics based on a single rule from convex analysis
- examples: YALMIP, CVX, CVXPY, Convex.jl, CVXR

- basic deal:
  - you accept strong restrictions on the problems you can specify
  - in return, your problem is solved globally and efficiently

CVXPY example

```
import cvxpy as cp

x = cp.Parameter((n, 1))
theta = cp.Parameter((n, n))

u = cp.Variable((m, 1))
x_next = cp.Variable((n, 1))

objective = cp.sum_squares(theta @ x_next) + cp.quad_form(u, R)
constraints = [x_next == A @ x + B @ u, cp.norm(u, "inf") <= 1]
cocp = cp.Problem(cp.Minimize(objective), constraints)

cocp.solve()
```

How they work

three steps:

1. *canonicalize* your problem description into a standard form
2. *solve* the standard form problem
3. *retrieve* solution of your problem from the standard form solution

normal people do not need to know this; they just call the `solve()` method

can view as three-step mapping from problem parameters to solution

$$\text{parameters} \longrightarrow \boxed{\mathcal{C}} \longrightarrow \boxed{\mathcal{S}} \longrightarrow \boxed{\mathcal{R}} \longrightarrow \text{solution}$$

Differentiating through a convex optimization problem

- if you accept some additional restrictions on how parameters enter the problem description, canonicalization and retrieval maps can be *linear*
- parameters-to-solution map is $R\mathcal{S}C$, where R and C are sparse *matrices*
- eliminates canonicalization / retrieval cost when you solve for different parameters

- derivative of parameters-to-solution map: $R(D\mathcal{S})C$
- can be chained to automatically and efficiently compute $\nabla_\theta \mathcal{L}(\theta)$ (even when $\mathcal{L}(\theta)$ involves solving many convex problems)

CVXPY layers

```
from cvxpylayers.torch import CvxpyLayer

layer = CvxpyLayer(cocp, parameters=[theta, x], variables=[u])

cost = 0.
for t in range(100):
  u_t, = layer(theta_torch, x_t)
  cost += stage_cost(u_t, x_t)
  x_t = dynamics(x_t, u_t)
cost.backward()
gradient = theta_torch.grad
```

Bonus: Code generation

- $C\mathcal{S}R$ form gives easy method for code generation
- compute R and C explicitly as sparse matrices
- canonicalization, retrieval now super fast
- link to suitable embedded solver like OSQP

Outline

Conclusions (non-controversial)

COCPs

- are simple and interpretable
- we understand how they work
- will never do anything crazy
- handle constraints, changes, failures gracefully
- can be safety fenced with constraints
- can be effectively tuned, quasi-automatically

there are or will soon be high-level tools to design and implement such controllers

Conclusion (controversial)

- *tuned COCP is the PID controller of the 21st century*