Implementation of an Iterative Linear Quadratic Regulator (iLQR)

Gabriel Desfrene   Antoine Groudiev

January 14, 2025

./images/logo-ens-psl.png

Plan

Problem statement

The iLQR algorithm

Our implementation

Demonstration time

Conclusion

Plan

General formulation

- Dynamics function:

$$x_{t+1} = f(x_t, u_t)$$

- Goal: minimize a quadratic cost function
- Cost function:

$$J(u) = \sum_{t=0}^{T-1} \left(x_t^\top Q x_t + u_t^\top R u_t \right) + \frac{1}{2}(x_T - x^*)^\top Q_f (x_T - x^*)$$

- Q: state cost matrix
- Q_f: final state cost matrix
- R: control cost matrix

Example: Simple Pendulum

- State: $x = [\theta \quad \dot{\theta}]$
- Control: u, torque applied to the pendulum
- Dynamics: physical laws (simulator)
- Target: $x = [0 \quad 0]$
- Cost function:

$$J(u) = \frac{1}{2}\left(\theta_f^2 + \dot{\theta}_f^2\right) + \frac{1}{2}\int_0^T r u^2(t)\mathrm{d}t$$

corresponding to $Q_f = I_2$, $Q = 0_2$, $R = rI_1$

./images/pendulum.png

Example: Cartpole

- State: $x = [y \quad \theta \quad \dot{y} \quad \dot{\theta}]$
- Control: u, force applied to the cart
- Dynamics: physical laws (simulator)
- Target: $x = [0 \quad 0 \quad 0 \quad 0]$
- Cost function:

$$J(u) = \frac{1}{2}\left(\theta_f^2 + \dot{\theta}_f^2 + y_f^2 + \dot{y}_f^2\right) + \frac{1}{2}\int_0^T r u^2(t)\mathrm{d}t$$

corresponding to $Q_f = I_4$, $Q = 0_4$, $R = rI_1$

./images/cartpole.png

Plan

General idea

- iLQR is an iterative algorithm
- Start with an initial trajectory
- Iteratively improve it using a local linear approximation
- Stop when the trajectory converges

Linearizing the dynamics

The equation $x_{t+1} = f(x_t, u_t)$ is linearized (at each step) as:

$$\delta x_{t+1} = A_t \delta x_t + B_t \delta u_t$$

with:

- A_t: Jacobian of f with respect to x evaluated at (x_t, u_t)
- B_t: Jacobian of f with respect to u evaluated at (x_t, u_t)

We are in LQR (Linear Quadratic Regulator, cf. TP5) setup!

Trajectory refinement using LQR

1. **Forward pass**: compute the successive states (x_t) for the current controls (u_t), and the corresponding cost J
2. **Backward pass**: compute the gains, i.e. how much we should change the controls in each direction to minimize the cost
3. **Forward rollout**: apply the gains to the controls to obtain a new trajectory
4. Repeat until convergence

For the complete derivations, see [1] or [3].

Computing the Jacobians

Finite differences method

We want to compute:

- $A_t = \frac{\partial f}{\partial x}(x_t, u_t)$, i.e. how much the state at time $t+1$ changes when we slightly change the state at time t
- $B_t = \frac{\partial f}{\partial u}(x_t, u_t)$, i.e. how much the state at time $t+1$ changes when we slightly change the control at time t

In a black box setting, we can use finite differences:

$$[A_t]_i \approx \frac{f(x_t + \varepsilon e_i, u_t) - f(x_t - \varepsilon e_i, u_t)}{2\varepsilon}$$

$$[B_t]_i \approx \frac{f(x_t, u_t + \varepsilon e_i) - f(x_t, u_t - \varepsilon e_i)}{2\varepsilon}$$

for some small ε and the canonical basis (e_i)

Computing the Jacobians

Using Pinocchio

```
# compute and store the Jacobians using Pinocchio
pin.computeABADerivatives(model, data_sim, q, v, u)

# retrieve the Jacobians and use them in the solver
data_sim.ddq_dq, data_sim.ddq_dv, data_sim.Minv
```

Tricks for practical convergence

- **Gradient clipping:** limit the size of the control updates norm to α to avoid divergence

$$\delta u_t = \frac{\delta u_i}{\max\left(1, \frac{\|\delta u_i\|}{\alpha}\right)}$$

- **Gaussian initialization**: start with a small random control sequence instead of a zero sequence

$$u_t \sim \mathcal{N}(0, \Sigma)$$

Plan

What language to use?

Python

- Easy to use
- Support for many libraries
- Embarrassingly slow

C++

- Fast
- Not very funny

Rust

- Fast
- Easy bindings for Python

Therefore, we chose to have a Rust core with Python bindings

From Rust to Python, and the other way around

- Instantiate the solver in Python
- Use Python libraries to define the dynamics
- The Rust solver does the computations, and calls the Python `dynamics` function and the Pinocchio functions for the Jacobians
- Supports both methods for computing the Jacobians

API Basic usage

```
def dynamics(x, u):
    return ... # simulator

Q = np.zeros((state_dim, state_dim)) # state cost
Qf = np.eye(state_dim) # final state cost
R = 1e-5 * np.eye(control_dim) # control cost (minimize the energy)

s = ilqr.ILQRSolver(state_dim, control_dim, Q, Qf, R)
target = np.zeros(state_dim) # upright pendulum with no velocity
output = s.solve(np.concatenate((q0, v0)), target, dynamics, time_steps=N,
                 gradient_clip=10.0,  # max norm of the gradient
                 initialization=0.5)  # std of the Gaussian initialization
```

Plan

Problem statement

The iLQR algorithm

Our implementation

Demonstration time

Conclusion

Demonstration time!

Plan

References

[1] Brian Jackson and Taylor Howell. *iLQR Tutorial*. Sept. 2019. URL: https://rexlab.ri.cmu.edu/papers/iLQR_Tutorial.pdf.

[2] Weiwei Li and Emanuel Todorov. “Iterative linear quadratic regulator design for nonlinear biological movement systems”. In: *First International Conference on Informatics in Control, Automation and Robotics*. Vol. 2. SciTePress. 2004, pp. 222–229.

[3] Harley Wiltzer. *iLQR Without Obfuscation*. Feb. 2020. URL: https://harwiltz.github.io/posts/20200201-ilqr/index.html.