Report
Implementation of an Iterative Linear Quadratic Regulator (iLQR)

Gabriel Desfrene Antoine Groudiev

Abstract

This report presents the Iterative Linear Quadratic Regulator (iLQR) approach to solve optimal control problems, and describes our implementation of this algorithm. We first introduce the problem of optimal control, the Linear Quadratic Regulator (LQR) algorithm used to solve it, and the extension provided by the iLQR algorithm. We then present our own implementation of the iLQR algorithm in `Rust`, and the `Python` bindings allowing its use alongside libraries such as Pinocchio.

1 Problem statement

The Iterative Linear Quadratic Regulator (iLQR) is a *trajectory optimization* method for discrete, nonlinear, and finite-horizon optimal control problems. It is an extension of the Linear Quadratic Regulator (LQR) algorithm, which is used to solve optimal control problems for linear systems with quadratic cost functions.

1.1 General formulation

Consider a discrete system described by a dynamics function:

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

meaning that we can compute the state x_{t+1} at time $t+1$ given the state x_t at time t and a control input u_t at time t. Our goal is to find the sequence of controls (u_t) that minimizes a given cost function; the iLQR method assumes that the cost function is quadratic, that is:

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

where Q_f is the final state cost matrix, Q the state cost matrix, and R the control cost matrix, all of them being positive semi-definite. Note that this restriction to quadratic functions is not a major limitation, as many cost functions are quadratic in practice, or can be approximated as such.

1.2 Example: the inverted pendulum

Consider a simple pendulum, where we denote $x = (\theta, \dot{\theta})$ the state of the system, with θ the angle between the pendulum and the vertical axis, and $\dot{\theta}$ its angular velocity. The dynamics of the system are given by physical laws, and can be seen as a black-box function implemented in a simulator. We can use the following cost function to stabilize the pendulum in the upright position:

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

with $r = 10^{-5}$. This corresponds to $Q_f = I_2$, $Q = O_2$, and $R = rI_1$. Such a cost function will penalize large angles and velocities, while minimizing the control effort (the integral term), with a weight r to balance the two objectives.

2 The iLQR algorithm

The iLQR iteratively refines a trajectory (x_t, u_t) by linearizing the dynamics around the current trajectory, and solving a sequence of LQR problems to find the optimal control inputs. The equation $x_{t+1} = f(x_t, u_t)$ is linearized as:

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

where A_t (resp. B_t) is the Jacobian of f with respect to x (resp. u) evaluated at (x_t, u_t). In practice, such Jacobians can be provided by the simulator, or approximated using finite differences.

The iLQR procedures iterates between the following three steps until convergence:

1. A *forward pass*, in which we compute the successive states (x_t) for the current controls (u_t), and the corresponding cost J.
2. A *backward pass*, in which we compute the gains, i.e. how much we should change the controls in each direction to minimize the cost.
3. A *forward rollout*, in which we apply the gains to the controls to obtain a new trajectory.

3 Implementation

We implemented the iLQR algorithm in `Rust`, a systems programming language known for its performance and safety guarantees. The core of the algorithm is implemented in `Rust`, while the `Python` bindings are generated with the help of the `pyo3` library, allowing its use alongside Pinocchio, a `C++` library for rigid body dynamics. The `Python` bindings use `NumPy` arrays as input and output, making it easy to integrate the iLQR algorithm in existing `Python` code, such as the examples provided by the `Pinocchio` library.

The code is available on GitHub: `https://github.com/Red-Rapious/iLQR`, and includes several examples of problems of interest. Our implementation successfully solves multiple classical trajectory optimization problems, such as the cart-pole and simple inverted pendulum.

4 Results

We tested our algorithm on two examples that we adapted from the `Pinocchio` library:

- `inverted-pendulum` that aims at balancing a pendulum in the upright position by applying a torque to its joint,
- `cartpole` that aims at balancing a pendulum in the upright position by moving the cart below.

The Table 1 summarize the results given by our algorithm for the two examples.

Example name	Cost Threshold	Iterations	Error (ℓ_2)	Time (sec)
`inverted-pendulum`	1.0	7	0.615	0.24
	0.1	8	0.023	0.38
`cartpole`	1.0	10	0.135	0.41
	0.1	22	0.047	0.76

Table 1: Results of our iLQR implementation.