Analysis of the Convergence Properties of Policy-Gradient Methods for Linear Quadratic Regulators paper

Jakub Jon [*]

1 Introduction

The paper of interest, titled "Global Convergence of Policy Gradient Methods for the Linear Quadratic Regulator", [1], aims to bridge the gap between model-free reinforcement learning and optimal control theory by providing theoretical guarantees for policy gradient methods in the context of the Linear Quadratic Regulator (LQR) problem.

The main claim of the paper is that despite being a non-convex optimization problem, policy gradient methods can still globally converge to the optimal solution (state-space feedback controller $\mathbf{K}$) with number of optimization iterations being polynomially bounded.

This is a significat result, as, unlike for theoretical guarantees in the field of optimal control, which are mostly model-based (the dynamics are fully known), the results in this paper require no explicit knowledge on the system dynamics. The only assumption is linearity, i.e. the system's continuous dynamics may be described as $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$, where $\mathbf{A}$ and $\mathbf{B}$ are time-independent system matrices, and $\mathbf{x}$, $\mathbf{u}$ represent the system's state and input, respectivelly.

In the first part of the paper, the authors analyze the convergence properties of three different optimization algorithms for the settings where the dynamics are fully known, i.e. matrices $\mathbf{A}$ and $\mathbf{B}$ are given.

These theoretical results are subsequently generalized for the settings where matrices $\mathbf{A}$ and $\mathbf{B}$ are unknown and all the necessary oracle quantities have to be estimated from direct interaction with the environment. This is done by proving that the neccessary quantities, introduced later in this work, can be estimated to any degree of accuracy and that the optimization algorithms have the same convergence guarantees, even in the presence of small estimation inaccuracies and pertubations.

This final project sets out to analyze some of the concepts presented in [1], implementing main methods in Python and analyzing and/or veryfying the main claims on arguably the simplest of dynamical systems, the double integrator [2].

To contribute to the open-source community and help other engineers understand the theoretical results of this paper through hands-on tinkering, all code used for simulation and graph generation is available online[1].

[*]Jakub Jon is with the Department of Electrical Engineering at North Carolina State University. Email: `jjon@ncsu.edu`.

[1]github.com/lnotspotl/tailqr

2 Analysis

2.1 Definitions

Let us first define the Linear Quadratic Regulator problem, giving an overview of the main terminology and formulations. The main goal of the LQR problem is to find a stabilizing state-space controller $\mathbf{K}$, with $\mathbf{u}_t = -\mathbf{K}\mathbf{x}_t$, such that an infinite horizon quadratic cost is minimized. What's penalized is the deviation of the state $\mathbf{x}_t$ from the origin as well as excesive use of the control authority $\mathbf{u}_t$. The relative importance of these two factors is given by the scaling of the matrices $\mathbf{Q} \succeq 0$ and $\mathbf{R} \succ 0$. The only constraint on the optimization problem is the dynamics, which, in our case, is defined to be discrete, linear and time-invariant. See Equation 1.

$$\begin{aligned} \min_{\mathbf{K}} \quad & \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[\sum_{t=0}^{\infty} \left(\mathbf{x}_t^\top \mathbf{Q} \mathbf{x}_t + \mathbf{u}_t^\top \mathbf{R} \mathbf{u}_t \right) \right] \\ \text{subject to} \quad & \mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t, \end{aligned} \tag{1}$$

2.2 Discrete Lyapunov Equation

Part of control theory is concerned with assesing stability of dynamic systems. For this discussion, assume we have a discrete linear time-invariant system whose dynamics are defined as $x_{k+1} = \mathbf{A}x_k$. We can define a so-called Lyapunov function $V(\mathbf{x}_k)$ as

$$V(\mathbf{x}_k) = \mathbf{x}_k^T \mathbf{P} \mathbf{x_k} \tag{2}$$

with $\mathbf{P}$ being a positive definite matrix. This function defines a generalized energy function. If we can guarantee that energy dissipates over time, i.e. the total energy of our system at time $k+1$ is lower than the energy of our system at time k, then stability in the sense of Lyapunov is guaranteed. We can formalize that as

$$V(\mathbf{x_{k+1}}) - V(\mathbf{x_k}) < 0$$
$$\mathbf{x}_{k+1}^T \mathbf{P} \mathbf{x}_{k+1} - \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k < 0$$
$$\mathbf{x}_k^T \mathbf{A}^T \mathbf{P} \mathbf{A} \mathbf{x}_k - \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k = \mathbf{x}_k^T \left(\mathbf{A}^T \mathbf{P} \mathbf{A} - \mathbf{P} \right) \mathbf{x}_k < 0$$

This needs to hold for all possible $\mathbf{x}_k$, and thus the matrix $\mathbf{A}^T\mathbf{P}\mathbf{A} - \mathbf{P}$ neccesarily has to be negative definite. In other words, if for every possible positive definite $\mathbf{Q}$, there exists a corresponding positive definite matrix $\mathbf{P}$ such that the following equation is satisfied,

$$\mathbf{A}^T\mathbf{P}\mathbf{A} - \mathbf{P} + \mathbf{Q} = \mathbf{0}$$

then the system, as defined by matrix $\mathbf{A}$, is stable in the sense of Lyapunov. There are many solvers for the discrete Lyapunov equation. In Python, one can get a solution by calling `scipy`'s `solve_discrete_lyapunov` function, with matrices $\mathbf{A}$ and $\mathbf{Q}$ as parameters.

2.3 Countinuous-Time Double Integrator

The double integrator is arguably the simplest of dynamical systems, typically used for sanity checks when developing control systems. The double integrator system's dynamics is governed by the following set equations

$$\dot{x}_1 = x_2, \tag{3}$$
$$\dot{x}_2 = u, \tag{4}$$

where x_1 is the position, x_2 is the velocity and finally u is the control input. This can be written in a compact form as

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u = \mathbf{A}_c \mathbf{x} + \mathbf{B}_c u$$

Hence, this is an example of a linear time-invariant system, which fits the problem definition in 2.1.

The double integrator system can be thought of as a sliding brick on an ice surface (no friction) that is being pushed around by force u. The goal is to move the brick from anywhere to the origin. See Figure 1.

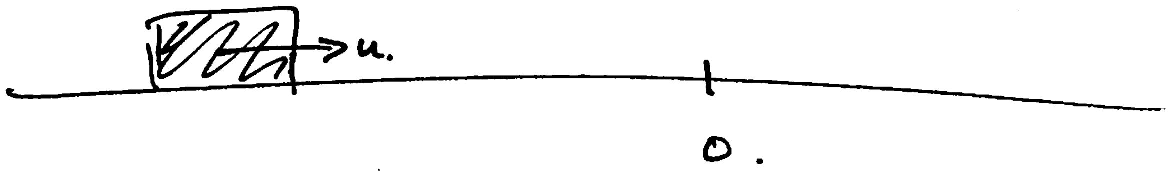

Figure 1: Double Integrator, 0 denotes the origin [3]

2.4 Discrete-Time Double Integrator

In [1], the authors used discrete-time dynamics instead of the continuous time dynamics. We can take the equations of motion, as presented in 2.3, and discretize them used the RK4[2] integration scheme.

$$\mathbf{x}_{k+1} = \begin{bmatrix} x_{1,\mathrm{k}} \\ x_{2,\mathrm{k}} \end{bmatrix} = \mathbf{A}\mathbf{x}_k + \mathbf{B}u_k = \begin{bmatrix} 1 & \mathrm{dt} \\ 0 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} \frac{\mathrm{dt}^2}{2} \\ \mathrm{dt} \end{bmatrix} u_k$$

In the equation obove, dt denotes the discrete time-step.

2.5 Analysis of $\mathbf{P}_K$

The original paper, [1], defines $\mathbf{P}_K$ as

$$\mathbf{P}_K = \mathbf{Q} + \mathbf{K}^T\mathbf{R}\mathbf{K} + (\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) \tag{5}$$

By rearanging Equation 5 and grouping some of the terms together, we get

$$(\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) - \mathbf{P}_K + \left[\mathbf{Q} + \mathbf{K}^T\mathbf{R}\mathbf{K}\right] = \mathbf{0} \tag{6}$$

Notice that this is an instance of the discrete Lyapunov equation, as $\mathbf{P}_K$ and $\mathbf{Q} + \mathbf{K}^T\mathbf{R}\mathbf{K}$ are both positive definite matrices. We can solve for $\mathbf{P}_K$ by calling `solve_discrete_lyapunov` with $\mathbf{A} - \mathbf{B}\mathbf{K}$ and $\mathbf{Q} + \mathbf{K}^T\mathbf{R}\mathbf{K}$ as arguments.

PK computation

```
def compute_PK(A, B, Q, R, K):
    closed_loop = A - B @ K
    return scipy.solve_discrete_lyapunov(closed_loop.T, Q + K.T @ R @ K)
```

[2] https://en.wikipedia.org/wiki/Runge-Kutta_methods

2.6 Analysis of Σ_K

By definition, we have

$$\mathbf{\Sigma}_K = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \sum_{t=0}^{\infty} \mathbf{x}_t\mathbf{x}_t^T \tag{7}$$

where $\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}u_t = (\mathbf{A} - \mathbf{B}\mathbf{K})\mathbf{x}_t$. Expanding the sum in Equation 7, we get

$$\mathbf{\Sigma}_K = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T + (\mathbf{A}-\mathbf{B}\mathbf{K})\mathbf{x}_0\mathbf{x}_0^T(\mathbf{A}-\mathbf{B}\mathbf{K})^T + (\mathbf{A}-\mathbf{B}\mathbf{K})^2\mathbf{x}_0\mathbf{x}_0^T(\mathbf{A}-\mathbf{B}\mathbf{K})^{2T} + \dots\right]$$

Notice that we can separate this expecation into two expectations, and since $(\mathbf{A} - \mathbf{B}\mathbf{K})$ doesn't depend on $\mathbf{x}_0$, we can factor it out of the expectation.

$$\mathbf{\Sigma}_K = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T\right] + (\mathbf{A}-\mathbf{B}\mathbf{K})\mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T + (\mathbf{A}-\mathbf{B}\mathbf{K})\mathbf{x}_0\mathbf{x}_0^T(\mathbf{A}-\mathbf{B}\mathbf{K})^T + \dots\right](\mathbf{A}-\mathbf{B}\mathbf{K})^T$$

$$\mathbf{\Sigma}_K = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T\right] + (\mathbf{A}-\mathbf{B}\mathbf{K})\mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\sum_{t=0}^{\infty} \mathbf{x}_t\mathbf{x}_t^T\right](\mathbf{A}-\mathbf{B}\mathbf{K})^T$$

$$\mathbf{\Sigma}_K = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T\right] + (\mathbf{A}-\mathbf{B}\mathbf{K})\mathbf{\Sigma}_K(\mathbf{A}-\mathbf{B}\mathbf{K})^T$$

Again, rearanging the terms in the above equation, we get

$$(\mathbf{A}-\mathbf{B}\mathbf{K})\mathbf{\Sigma}_K(\mathbf{A}-\mathbf{B}\mathbf{K})^T - \mathbf{\Sigma}_K + \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T\right] = 0$$

This is yet another instance of the discrete Lyapunov equation. Assuming $\mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0\mathbf{x}_0^T\right] = \mathbf{I}$, we can solve for $\mathbf{\Sigma}_K$ by calling the `solve_discrete_lyapunov` with $(\mathbf{A}-\mathbf{B}\mathbf{K})^T$ and $\mathbf{I}$ as parameters.

SigmaK computation

```python
def compute_SigmaK(A, B, Q, R, K):
    closed_loop = A - B @ K
    return scipy.solve_discrete_lyapunov(closed_loop, np.eye(2))
```

2.7 Analysis of C(k)

Let us now turn to the calculation of $C(\mathbf{K})$.

$$C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathbf{x}_0^T\mathbf{P}_K\mathbf{x}_0\right]$$

Notice that $C(\mathbf{K})$ is a scalar, so we may as well calculate it as a trace.

$$C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathrm{tr}(\mathbf{x}_0^T\mathbf{P}_K\mathbf{x}_0)\right]$$

Now, we will exploit the cyclic property of the trace operator[3]

$$C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[\mathrm{tr}(\mathbf{P}_K\mathbf{x}_0\mathbf{x}_0^T)\right]$$

Since $\mathbf{P}_K$ is a constant in the context of the above expectation, we can take it out of the expectation. Moreover, we can switch the order of the expectation operator and the trace operator, as they both are an instance of a linear operator.

$$C(\mathbf{K}) = \mathrm{tr}(\mathbf{P}_K\mathbb{E}_{\mathbf{x}_0\sim\mathcal{D}} \left[(\mathbf{x}_0\mathbf{x}_0^T)\right]) = \mathrm{tr}(\mathbf{P}_K\mathbf{\Sigma}_0)$$

[3] https://en.wikipedia.org/wiki/Trace_(linear_algebra)

where $\mathbf{\Sigma}_0 = \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[\mathrm{tr}(\mathbf{x}_0 \mathbf{x}_0^T) \right]$. In this work, we will pick distributions with $\mathbf{\Sigma}_0 = \mathbf{I}$ and thus $C(\mathbf{K}) = \mathrm{tr}(\mathbf{P}_K)$. $\mathbf{P}_K$ may be computed as described in Subsection 2.5.

Ck computation

```python
def compute_CK(A, B, Q, R, K):
    PK = compute_PK(A,B,Q,R,K)
    return np.trace(PK)
```

Sigma0 computation

```python
def compute_Sigma0(A, B, Q, R, K):
    return np.eye(2)
```

3 Algorithms

3.1 Exact Gradient Algorithms

In this section, we will assume the initial state ditribution $\mathcal{D}$ to be $\mathcal{N}(\mathbf{0}, \mathbf{I})$, that is a gaussian distribution with zero mean and unit variance. This satisfies our assumption that $\mathbf{\Sigma}_0 = \mathbf{I}$, and thus we can make use of the very formulas presented in Sections 2.5, 2.6 and finally 2.7. The goal of the optimization is to find such a policy parameterized by $\mathbf{K}$ that minimizes the cost function defined in Subsection 2.1. This leads us to the algorithms we will be using to minimize the cost function $C(\mathbf{K})$, where $\mathbf{K}$ is the decision variable.

3.1.1 Gradient $\nabla C(\mathbf{K})$

Let's revisit the formulation of $C(\mathbf{K})$.

$$C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[\mathbf{x}_0^T \mathbf{P}_K \mathbf{x}_0 \right], \tag{8}$$

$$\mathbf{P}_K = \mathbf{Q} + \mathbf{K}^T \mathbf{R} \mathbf{K} + (\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) \tag{9}$$

By substituting the Equation 9 into Equation 8, we get

$$C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[\mathbf{x}_0^T (\mathbf{Q} + \mathbf{K}^T \mathbf{R} \mathbf{K} + (\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K})) \mathbf{x}_0 \right]$$

Note that the expectation is an instance of a linear operator, therefore, after applying the gradient operator, we can move the ∇ symbol inside of the expectation. Doing a little bit of matrix calculus, we get

$$\nabla C(\mathbf{K}) = \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[2\mathbf{R}\mathbf{K}\mathbf{x}_0\mathbf{x}_0^T - 2\mathbf{B}^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) \mathbf{x}_0 \mathbf{x}_0^T + \nabla(\mathbf{x}_1^T \mathbf{P}_K \mathbf{x}_1) \right] \tag{10}$$

The last term in the Equation 10, again, can be substituted into, giving us a recursive equation. By recursively expanding the last term and moving all terms independent of $\mathbf{x}_0$ in front of the expectation operator, we get

$$\nabla C(\mathbf{K}) = 2 \left(\mathbf{R}\mathbf{K} - \mathbf{B}^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) \right) \mathbb{E}_{\mathbf{x}_0 \sim \mathcal{D}} \left[\sum_{t=0}^{\infty} \mathbf{x}_t \mathbf{x}_t^T \right] = 2 \left(\mathbf{R}\mathbf{K} - \mathbf{B}^T \mathbf{P}_K (\mathbf{A} - \mathbf{B}\mathbf{K}) \right) \mathbf{\Sigma}_K \tag{11}$$

All matrices in Equation 11 can easily be computed, thus we end here.

Gradient Calculation

```python
def compute_gradCK(A, B, Q, R, K):
    PK = compute_PK(A,B,Q,R,K)
    SigmaK = compute_SigmaK(A,B,Q,R,K)
    return 2 * (R @ K - B.T @ PK @ (A - B @ K)) * SigmaK
```

3.1.2 Gradient domination

According to [1], the distance on between $C(\mathbf{K})$ and $C(\mathbf{K}^*) = \min_{\tilde{\mathbf{K}}} C(\tilde{\mathbf{K}})$ is dominated by norm of the gradient of $C(\mathbf{K})$ evaluated at K, and the smallest singular value of $\mathbf{\Sigma}_K$, analyzed in Subsection 2.6.

$$C(\mathbf{K}) - C(\mathbf{K}^*) \leq \frac{||\mathbf{\Sigma}_{K^*}||}{\sigma_{\text{min}}(\mathbf{\Sigma}_K)^2 \sigma_{\text{min}}(\mathbf{R})} ||\nabla C(\mathbf{K})||_F^2,$$

where $\sigma_{\text{min}}(\mathbf{N})$ denotes the smallest singular value of an arbitrary matrix $\mathbf{N}$. This is a general results and should hold for all stabilizing $\mathbf{K}$. For that reason, in all our simulations, we continually check this condition, making sure it never gets violated.

Gradient Domination Checking

```python
def check_gradient_domination(A, B, Q, R, K, K_opt):
    Ck = compute_CK(A,B,Q,R,K)
    Ck_opt = compute_CK(A,B,Q,R,K_opt)

    Sigma_K = compute_SigmaK(A,B,Q,R,K)
    Sigma_K_opt = compute_SigmaK(A,B,Q,R,K_opt)

    # Get minimum singular values of K and R
    min_sv_K = np.min(np.linalg.svd(Sigma_K)[1])
    min_sv_R = np.min(np.linalg.svd(R)[1])

    grad_CK = compute_gradCK(A,B,Q,R,K)

    C_diff = Ck - Ck_opt
    ub = np.linalg.norm(Sigma_K_opt, ord=2)
    ub = ub / (min_sv_K**2 * min_sv_R) * np.linalg.norm(grad_CK)**2
    assert ub > C_diff, f"C_diff_upper: {ub}, C_diff: {C_diff}"
```

3.1.3 Policy Gradient Descent

The simplest of first-order optimization algorithms is the gradient descent algorithm. The update rule is given by Equation 12. $\mathbf{K}_{n+1}$ denotes the new full-state feedback controller, $\mathbf{K}_n$ is the current feedback controller, η is the learning rate and $\nabla C(\mathbf{K}_n)$ represents the gradient of the cost function w.r.t. the current feedback controller $\mathbf{K}_n$.

$$\mathbf{K}_{n+1} = \mathbf{K}_n - \eta \nabla C(\mathbf{K}_n) \tag{12}$$

The optimization loop is implemented as follows:

Exact Policy Gradient Optimization Code

```
def exact_policy_gradient(A,B,Q,R,alpha,K0,max_iters):
    K = K0
    for i in range(max_iters):
        grad_CK = compute_gradCK(A,B,Q,R,K)
        K_new = K - alpha * grad_C_K
        K = K_new
    return K
```

Of all the optimization algorithms presented in this subsection, the exact gradient descent algorithm has the weakest convergence guarantees. Nevertheless, the gradient descent algorithm only requires the first order oracle, quierying just for the gradient of the cost function $\nabla C(\mathbf{K}_n)$, rendering it the simplest and easiest to implement optimization algorithm.

3.1.4 Natural Policy Gradient Descent

The natural policy gradient descent algorithms improves upon the convergence guarantess of the gradient descent algorithm by post-multiplying the gradient by $\mathbf{\Sigma}_{K_n}^{-1}$. This one factor, as proven in [1], gives stronger convergence guarantees. The downside is that one has to have access to a more powerful oracle, capable of producing both $\nabla C(\mathbf{K}_n)$ and $\mathbf{\Sigma}_{K_n}$. The update rule is given by Equation 13.

$$\mathbf{K}_{n+1} = \mathbf{K}_n - \eta \nabla C(\mathbf{K}_n) \mathbf{\Sigma}_{K_n}^{-1} \tag{13}$$

Below, we give our implementation of the Nautral Policy Gradient optimization algorithm.

Exact Natural Policy Gradient Optimization Code

```
def exact_natural_policy_gradient(A,B,Q,R,alpha,K0,max_iters):
    K = K0
    for i in range(max_iters):
        grad_CK = compute_gradCK(A,B,Q,R,K)
        SigmaK = compute_SigmaK(A,B,Q,R,K)
        K_new = K - alpha * grad_CK @ scipy.linalg.inv(SigmaK)
        K = K_new
    return K
```

3.1.5 Gauss-Newton Algorithm

The algorithm with the strongest convergence guarantess is the Gauss-Newton algorithm, moving towards the optimal feedback controller $\mathbf{K}$ quadratically, instead of linearly, as is the case for the gradient descent algorithm (natural gradient descent is somewhere inbetween). The oracle for the Gauss-Newton algorithm is the most complicated. However, in exchange for its complexity, one gets access to all $\mathbf{P}_{K_n}$, $\nabla C(\mathbf{K}_n)$ and $\mathbf{\Sigma}_{K_n}$ values. The update rule exploits this extra information and is given in Equation 14.

$$\mathbf{K}_{n+1} = \mathbf{K}_n - \eta \left(\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{K_n} \mathbf{B} \right)^{-1} \nabla C(\mathbf{K}_n) \mathbf{\Sigma}_{K_n}^{-1}. \tag{14}$$

The following is our implementation of the Gauss-Newton optimization algorithm.

Exact Gauss-Newton Optimization Code

```
def exact_gauss_newton(A,B,Q,R,alpha,K0,max_iters):
    K = K0
    for i in range(max_iters):
        PK = compute_PK(A,B,Q,R,K)
        gradCK = compute_gradCK(A,B,Q,R,K)
        Sigma_K = compute_SigmaK(A,B,Q,R,K)
        T = scipy.linalg.inv(R + B.T @ PK @ B)
        K_new = K - alpha * T @ grad_C_K @ scipy.linalg.inv(Sigma_K)
        K = K_new
    return K
```

3.1.6 Exact Gradient Algorithm Convergence

We have conductect two experiments, each with different matrices $\mathbf{Q}$ and $\mathbf{R}$ definining the quadratic cost function. The optimal state-space feedback controller $\mathbf{K}^*$ is produced by running the Gauss-Newton algorithm till convergence. The assumption we are making here is that the original paper's claim, that all the above methods converge to the globally optimal state-space controller, is true.

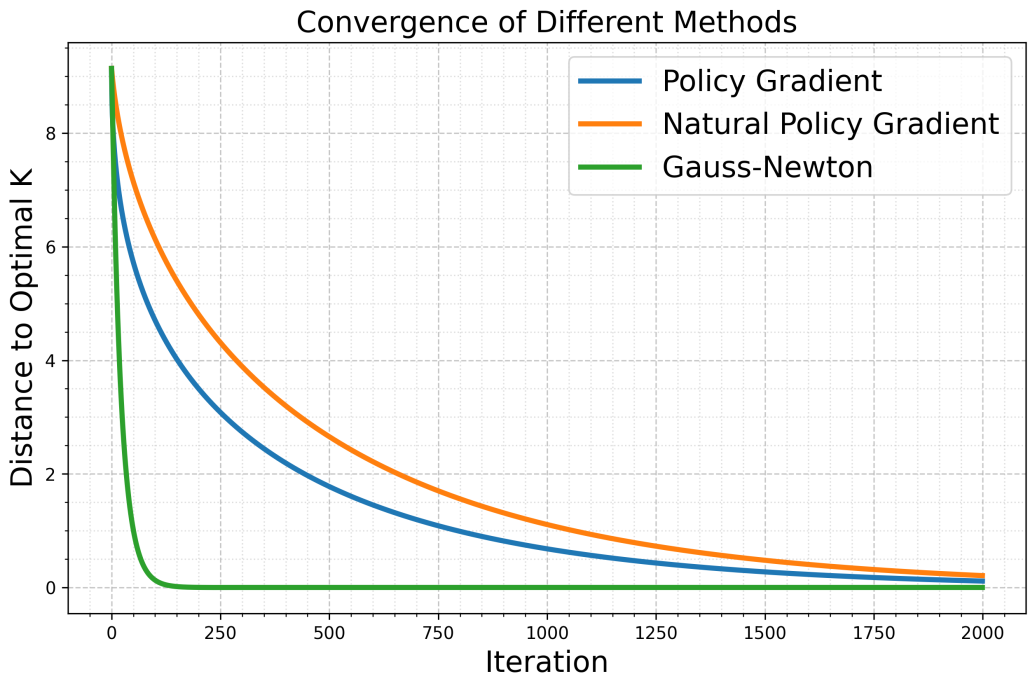

Figure 2: Convergence for exact gradient optimization algorithms with $\mathbf{Q} = \text{diag}(2, 2)$ and $\mathbf{R} = (0.01)$

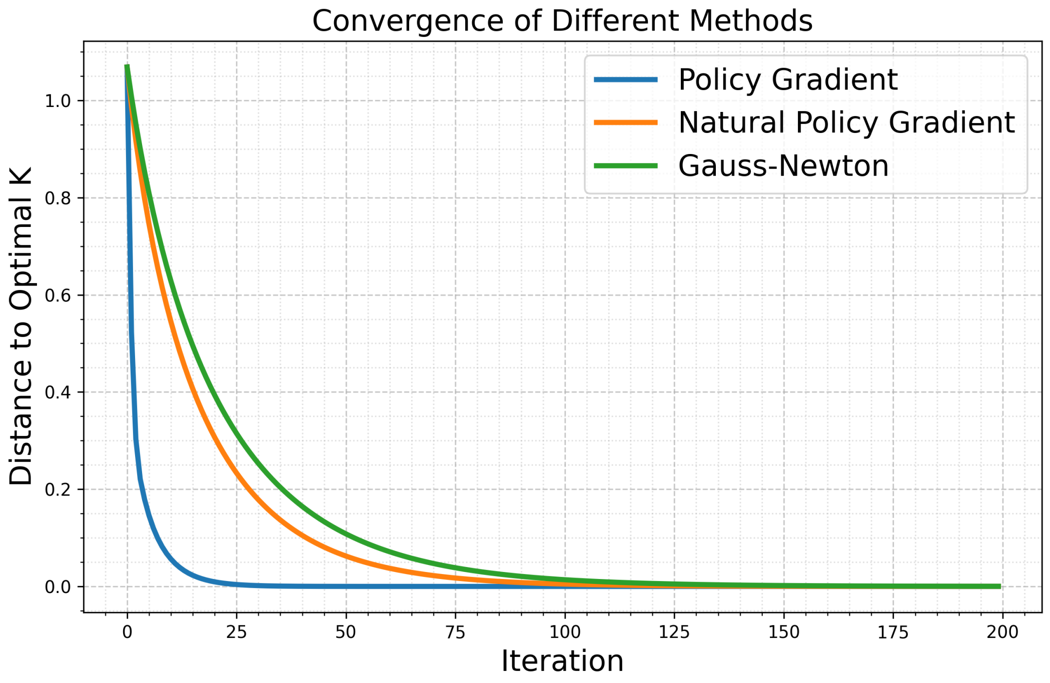

Figure 3: Convergence for exact gradient optimization algorithms with $\mathbf{Q} = \mathrm{diag}(2, 2)$ and $\mathbf{R} = (1)$

The two pictures above clearly show that while the Gauss-Newton optimization may have stronger convergence guarantees, it may not necessarily converge to the optimal solution faster than other algorithms, potentially those with weaker convergence guarantees. A convergence guarantee is just an upper-bound, telling us something about the worst-case scenario.

3.2 Approximate Gradient Algorithms

In the Reinforcement Learning settings, we generally do not have access to the system matrices $\mathbf{A}$ and $\mathbf{B}$ defining the systems dynamics. In that case, the gradient $\nabla C(\mathbf{K})$ and/or the state correlation matrix $\mathbf{\Sigma}_K$ have to be approximated using Monte-Carlo methods. The original paper [1] provides a pseudocode to do that, here we give our Python implementation:

$\nabla C(\mathbf{K})$ and $\mathbf{\Sigma}_K$ estimation code

```python
def policy_gradient_estimation(A, B, Q, R, K, m, rollout_length, r, d):
    C_estimates, sigma_estimates = [], []
    for _ in range(m):
        # Sample random matrix U_i with Frobenius norm \leq r
        U_i = np.random.uniform(-1, 1, size=K.shape)
        U_i = r / np.linalg.norm(U_i, ord="fro") * U_i
        K_i = K + U_i # Perturbed policy
        total_cost = 0.0
        states = []
        # Sample initial state
        x = (np.random.rand(d) * 2 - 1) * (2 * np.sqrt(3))
        for _ in range(rollout_length):
            states.append(x)
            u = -np.dot(K_i, x)
            total_cost += x.T @ Q @ x  # state cost
            total_cost += u.T @ R @ u  # input cost
            x = np.dot(A, x) + np.dot(B, u)
        # Calculate empirical estimates
        C_i = total_cost
        Sigma_i = np.zeros((d, d))
        for s in states:
            Sigma_i += np.outer(s, s)
        C_estimates.append((C_i, U_i))
        sigma_estimates.append(Sigma_i)
    # Calculate final estimates
    gradient_estimate = np.zeros(K.shape)
    for C_i, U_i in C_estimates:
        gradient_estimate += (d / (m * r * r)) * C_i * U_i
    sigma_estimate = np.mean(sigma_estimates, axis=0)
    return gradient_estimate, sigma_estimate
```

The above code uses a modified version of the famous gradient estimation technique, proposed by Nesterov and Spokoiny in their publication on gradient-free optimization of convex functions [4]:

$$\nabla f_{\sigma^2}(\mathbf{x}) = \frac{1}{\sigma^2}\mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(0,\sigma^2\mathbf{I})}\left[f(\mathbf{x}+\boldsymbol{\epsilon})\boldsymbol{\epsilon}\right] \tag{15}$$

3.2.1 Approximate Policy Gradient Descent

We use the exact same code as in Subsection 3.1.3, with the only difference being the use of the `policy_gradient_estimation` function, instead of making `compute_gradCK` function calls.

Approximate Policy Gradient Optimization Code

```python
def approximate_policy_gradient(
    A,B,Q,R,m,rollout_length, r, d, alpha, K0, max_iters
):
    K = K0
    for i in range(max_iters):
        grad_CK, _ = policy_gradient_estimation(A,B,Q,R,m,rollout_length,r,d)
        K_new = K - alpha * grad_C_K
        K = K_new
    return K
```

3.2.2 Approximate Natural Policy Gradient Descent

Unlike the approximate policy gradient descent algorithm, the approximate natural policy gradient descent method makes use of the estimated state correlation matrix $\mathbf{\Sigma}_K$. Except for that, again, the algorithm stays the same as in Subsection 3.1.4.

Approximate Policy Gradient Optimization Code

```python
def approximate_natural_policy_gradient(
    A,B,Q,R,m,rollout_length, r, d, alpha, K0, max_iters
):
    K = K0
    for i in range(max_iters):
        grad_CK, SigmaK = policy_gradient_estimation(
            A,B,Q,R,m,rollout_length,r,d)
        K_new = K - alpha * grad_C_K * np.linalg.inv(SigmaK)
        K = K_new
    return K
```

3.2.3 Approximate Gradient Algorithm Convergence

Guarantees for the approximate algorithms are only valid if the norm of a sample from $\mathcal{D}$ is bounded by $L > 0$, where L is an arbitraty constant. To keep the code the same, instead of sampling from $\mathcal{D} = \mathcal{N}(0, I)$, we will sample the initial position $\mathbf{x}_0$ from $\mathcal{D} = \text{Uni}(-2\sqrt{3}, +2\sqrt{3})$, where $\text{Uni}(-2\sqrt{3}, +2\sqrt{3})$ is such a distribution that samples each element of $\mathbf{x}_0$ uniformly between $-2\sqrt{3}$ and $+2\sqrt{3}$. Each sample has a bounded norm, it's mean is zero and the covariance matrix equals to the identity matrix, thus fitting all our and the paper's assumptions.

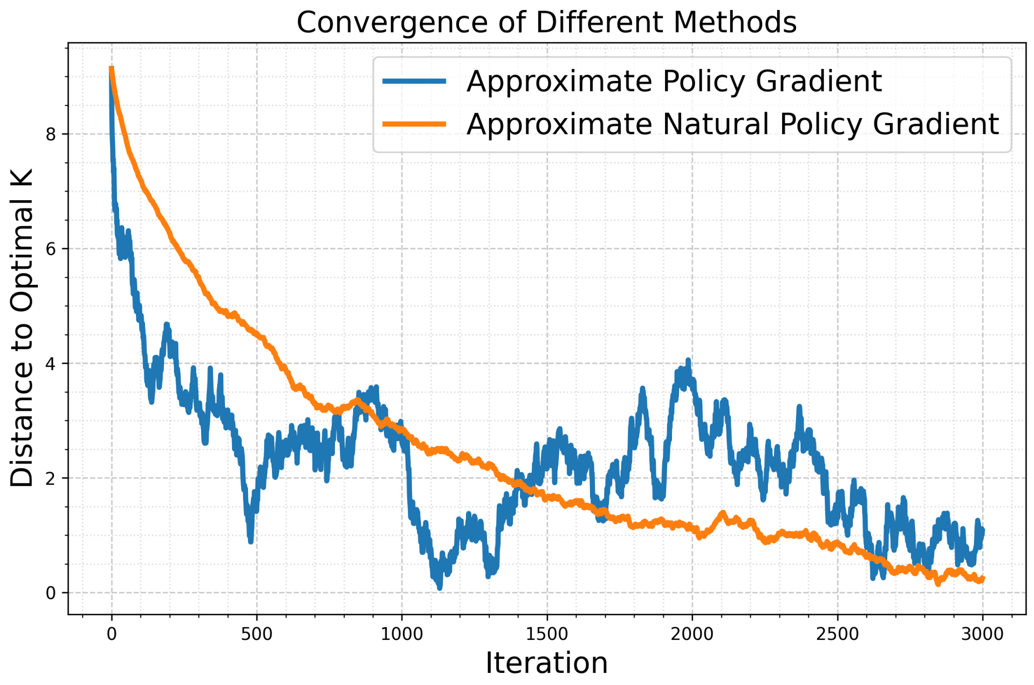

Figure 4: Convergence for approximate gradient optimization algorithms with $\mathbf{Q} = \mathrm{diag}(2, 2)$ and $\mathbf{R} = (0.01)$. The rest of the parameters may be found in the source code.

4 Conclusion

In conclusion, this project has gone through the main ideas presented in the "Global Convergence of Policy Gradient Methods for the Linear Quadratic Regulator" paper, implementing multiple of them in Python for the simplest of dynamic systems - the double integrator. This not only provides the possibility of experimental verification of claims presented in [1], but it also gives engineers the ability to build on top of the code and understand the main theory better, due to its interactive nature. Finally, as the authors of the original paper have not open-sourced any of their verification source codes, this project aims to fill that gap[4], making the paper itself more understandable through being interactive, through being accesible to a wider audience and through being easier to understand, if read alongside the original paper.

References

[1] Maryam Fazel, Rong Ge, Sham M. Kakade, and Mehran Mesbahi. Global convergence of policy gradient methods for linearized control problems. *CoRR*, abs/1801.05039, 2018.

[2] Marcello Romano and Fabio Curti. Analytic solution of the time-optimal control of a double integrator from an arbitrary state to the state-space origin, 2019.

[3] Russ Tedrake. *Underactuated Robotics*. 2023.

[4] github.com/lnotspotl/tailqr

[4] Vladimir Spokoiny Yurii Nesterov. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2015.