Notes on Optimal Control and Learning (In-Progress)

Bipin Koirala

December 2024

Contents

1 Preliminary on Optimization **3**
1.1 Some Notation . . . 3
1.2 Newton's Method . . . 4
1.3 How to Optimize when the Hessian isn't P.D.? . . . 5
1.4 Line Search . . . 5

2 Optimization Crash Course **6**
2.1 Equality Constraints . . . 6
2.2 Inequality Constraints . . . 7
2.2.1 KKT Conditions (for Inequality Constraints) . . . 8
2.2.2 Optimization Algorithms for Solving these KKT Conditions . . . 8
2.2.3 Regularization on Unconstrained Optimization . . . 10
2.2.4 Merit Functions in Constrained Optimization . . . 11

3 Deterministic Optimal Control **12**
3.1 Pontryagin's Maximum/Minimum Principle . . . 13
3.2 Linear Quadratic Regulator (LQR) . . . 16
3.2.1 LQR with Indirect Shooting (Pontryagin's Principle) . . . 16
3.2.2 LQR as Quadratic Programming . . . 18
3.2.3 Bellman's Principle & LQR via Dynamic Programming . . . 23
3.3 Convex Model Predictive Control (MPC) . . . 25
3.4 Non-Linear Trajectory Optimization . . . 28
3.5 DDP Recap . . . 31
3.5.1 Regularization . . . 31
3.5.2 Handling Constraints in DDP . . . 32
3.6 Minimum/Free-time Problems . . . 33
3.7 Direct Trajectory Optimization . . . 33
3.7.1 Direct Collocation . . . 34
3.8 Algorithm Recap . . . 36
3.9 Control in the case of a Bad system model . . . 37
3.9.1 Iterative Learning Control (ILC) . . . 38

4 Stochastic Optimal Control **39**
4.1 Linear Quadratic Gaussian (LQG) . . . 39
4.2 Optimal State Estimation . . . 40
4.2.1 Kalman Filter . . . 41
4.2.2 Probabilistic Interpretation of Kalman Filter . . . 42
4.3 Brief Overview of Game-Theoretic Trajectory Optimization . . . 44
4.4 PDE & BSDE for Stochastic Optimal Control . . . 45

5 Deep Learning from a Dynamical System Perspective 47
5.1 Information Propagation inside DNN (Fully Connected) 47
5.2 Stability Analysis and Implications 48
5.2.1 Deep Information and Gradient Descent 51
5.3 Training DNN with Optimal Control 51
5.3.1 Mean-Field Pontryagin's Minimum Principle 52
5.3.2 Mean-Field Hamilton-Jacobi-Bellman Equation 53
5.4 Stochastic Optimization as a Dynamical System 54
5.4.1 Introduction to Stochastic Mini-Batch Gradient Descent 54
5.4.2 Continuous-time Dynamics of SGD 55
5.5 Control Inspired Learning Algorithms 56
5.5.1 Method of Successive Approximation 57
5.5.2 Layer Parallel Training Algorithms 58

6 Examples: Runge-Kutta (ResNet) and ODENets 59
6.0.1 Runge-Kutta Networks 60
6.0.2 ODENet 60
6.1 A Simple Experiment 61
6.2 Future Directions 64

1 Preliminary on Optimization

1.1 Some Notation

Given $f(x) : \mathbb{R}^n \to \mathbb{R}$:

$$\frac{\partial f}{\partial x} \in \mathbb{R}^{1\times n} \quad \text{(row vector)}$$

This is because $\frac{\partial f}{\partial x}$ is a linear operator mapping Δx into Δf.

Taylor Expansion Style

The Taylor expansion of $f(x + \Delta x)$ is:

$$f(x+\Delta x) \approx f(x) + \left(\frac{\partial f}{\partial x}\right)\Delta x$$

Similarly, for $g(y) : \mathbb{R}^m \to \mathbb{R}^n$:

$$\frac{\partial g}{\partial y} \in \mathbb{R}^{n\times m}$$

because $g(y+\Delta y) \approx g(y) + \frac{\partial g}{\partial y}\Delta y$.
Note: These conventions make the chain rule work:

$$f(g(y+\Delta y)) \approx f(g(y)) + \left.\frac{\partial f}{\partial x}\right|_{g(y)} \left.\frac{\partial g}{\partial y}\right|_{y} \Delta y$$

For convenience, we define:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x}\right)^T \in \mathbb{R}^{n\times 1}$$

Hessian

The Hessian matrix is defined as:

$$\nabla^2 f(x) = \frac{\partial}{\partial x}(\nabla f(x)) = \frac{\partial^2 f}{\partial x^2} \in \mathbb{R}^{n\times n}$$

The Hessian provides information about the local curvature of a function.

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Some facts about Hessians:

- Their eigenvalues will always be real numbers.
- $H > 0$: Concave up $\cup$
- $H < 0$: Concave down $\cap$
- $H = 0$: More information needed (graph) to understand the behavior.
- If eigenvalues are mixed (positive/negative), then it is a saddle point.

Root Finding Problem

Given a function $f(x)$, find x^* such that $f(x^*) = 0$.

Fixed Point Iteration

- Simplest solution method.
- Only works if the fixed point is stable.
- Iterate the dynamics until convergence to x^*.
- Works only for stable x^* and has slow convergence.

1.2 Newton's Method

- Fit a linear approximation to $f(x)$:

$$f(x + \Delta x) \approx f(x) + \left.\frac{\partial f}{\partial x}\right|_x \Delta x$$

- Set approximation to 0 and solve for Δx:

$$f(x) + \frac{\partial f}{\partial x}\Delta x = 0 \quad \Rightarrow \quad \Delta x = -\left(\frac{\partial f}{\partial x}\right)^{-1} f(x)$$

- Apply correction: $x \leftarrow x + \Delta x$ (update).
- Repeat until convergence.

Minimization

$$\min_x f(x), \quad f(x) : \mathbb{R}^n \to \mathbb{R}$$

If f is smooth, $\nabla f(x) = 0$ at the local minimum. Now we have a root-finding problem $\nabla f(x) = 0$. Apply Newton's method:

$$\nabla f(x + \Delta x) \approx \nabla f(x) + \frac{\partial}{\partial x}(\nabla f(x))\Delta x = 0$$

Thus, $\Delta x = -\left(\nabla^2 f(x)\right)^{-1} \nabla f(x)$ and $x \leftarrow x + \Delta x$
Intuition: Fit a quadratic approximation to $f(x)$, which exactly minimizes the approximation.

Takeaway Message

Newton's method is a local root-finding method that will converge to the closest fixed point to the initial guess (whether it is a minimum, maximum, or saddle).

Sufficient Conditions

- $\nabla f(x) = 0$ is the *first-order necessary condition* for a minimum. It is not a sufficient condition.
- For a scalar case:
  - $\nabla^2 f(x) > 0 \Rightarrow$ descent (minimization).
  - $\nabla^2 f(x) < 0 \Rightarrow$ ascent (maximization).

1.3 How to Optimize when the Hessian isn't P.D.?

The solution to this is **Regularization** to ensure we're always minimizing:

$$H \leftarrow \nabla^2 f(x)$$

while $H \not> 0$

$$H \leftarrow H + \beta I \quad (\beta > 0)$$

where β is a scalar hyperparameter.
This approach is also called "damped Newton." It guarantees descent and shrinks steps.
The update step:

$$\Delta x = -H^{-1} \nabla f(x)$$

$$x \leftarrow x + \Delta x$$

Note: $\beta > 0 \Rightarrow H > 0$.

Algorithm 1 Regularization for Positive Definite Hessian (Damped Newton)

```
1: Input: Function $f(x)$, initial point $x$, tolerance $\epsilon$, regularization parameter $\beta > 0$
2: Output: Updated point $x$
3: Initialize $H \leftarrow \nabla^2 f(x)$
4: while $H$ is not positive definite do
5:    $H \leftarrow H + \beta I$ {Regularize the Hessian by adding a scaled identity matrix}
6: end while
7: Compute step direction: $\Delta x \leftarrow -H^{-1} \nabla f(x)$
8: Update: $x \leftarrow x + \Delta x$
9: Return $x$ =0
```

1.4 Line Search

Often, the Δx step from Newton's method will overshoot the minimum. To fix this issue, we apply the following:
1. Check $f(x + \Delta x)$ and "backtrack" until we get a "good" reduction.
There exist many strategies to do this. One simple and effective way is called the **Armijo Rule**.
To do this:

- Start with a small step $\alpha = 1$
- While $f(x + \alpha \Delta x) > f(x) + b\alpha \nabla f(x)^T \Delta x$, reduce α,

$$\alpha \leftarrow c\alpha$$

  where b is a tolerance factor, and $c \in (0, 1)$.

Intuition: Make sure the step agrees with linearization within some tolerance "b".
Note: In general, $c = \frac{1}{2}$ and $b \in [10^{-4}, 0.1]$.

Algorithm 2 Backtracking Line Search with Armijo Rule

```
1: Input: Function $f(x)$, gradient $\nabla f(x)$, step direction $\Delta x$, initial step size $\alpha = 1$, parameters $c \in (0, 1)$,
   $b \in (0, 1)$
2: Output: Step size $\alpha$
3: Initialize $\alpha \leftarrow 1$
4: while $f(x + \alpha \Delta x) > f(x) + b\alpha \nabla f(x)^T \Delta x$ do
5:    $\alpha \leftarrow c \cdot \alpha$ {Reduce step size by a factor of $c$}
6: end while
7: Return $\alpha$ =0
```

Listing 1: Backtracking Regularized Newton Step

```
import numpy as np

def backtracking_regularized_newton_step(f, grad_f, hessian_f, x, beta=1e-3, alpha=1, c=0.5,
     b=1e-4, max_iter=100):
    """
    Performs one step of the backtracking regularized Newton method.

    Parameters:
    f: Function to minimize.
    grad_f: Gradient of the function.
    hessian_f: Hessian of the function.
    x: Current point.
    beta: Regularization parameter for Hessian (default=1e-3).
    alpha: Initial step size (default=1).
    c: Scaling factor for backtracking (default=0.5).
    b: Tolerance for Armijo rule (default=1e-4).
    max_iter: Maximum iterations for backtracking (default=100).

    Returns:
    x_new: Updated point.
    """

    # Calculate gradient and Hessian at the current point
    g = grad_f(x)
    H = hessian_f(x)

    # Ensure the Hessian is positive definite
    while np.linalg.eigvalsh(H).min() <= 0:
        H += beta * np.eye(H.shape[0])

    # Compute the Newton step
    delta_x = -np.linalg.inv(H).dot(g)

    # Backtracking line search using Armijo rule
    for _ in range(max_iter):
        if f(x + alpha * delta_x) <= f(x) + b * alpha * np.dot(g, delta_x):
            break
        alpha *= c  # Reduce the step size

    # Update the point
    x_new = x + alpha * delta_x

    return x_new
```

Key Take-away

- Newton's method, with simple and cheap modifications ("globalization strategies"), offers quadratic convergence speed when close to the actual solution.
- This will only give you local optima.

---

2 Optimization Crash Course

2.1 Equality Constraints

Suppose we have a function $f(x) : \mathbb{R}^n \to \mathbb{R}$, and we want to minimize the function "cost" such that some constraints $C(x) : \mathbb{R}^n \to \mathbb{R}^m$ are satisfied. Here, $C(x)$ can be a vector.
Mathematically:

$$\min_x f(x) \quad \text{s.t.} \quad C(x) = 0$$

In order to solve this optimization problem, we have the **first-order necessary conditions**, i.e.:

1. We need $\nabla f(x) = 0$ only in the "free direction."
2. We need to satisfy the constraint, i.e. $C(x) = 0$.

Any non-zero component of ∇f must be normal to the constraint surface/manifold, i.e.:

$$\nabla f + \lambda^T \nabla C = 0$$

where λ is the Lagrange multiplier (dual variable).
In general:

$$\frac{\partial f}{\partial x} + \lambda^T \frac{\partial C}{\partial x} = 0 \quad \text{where} \quad \lambda \in \mathbb{R}^m$$

Based on this gradient condition, we define the **Lagrangian**:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T C(x)$$

such that (the first-order necessary conditions are):

$$\begin{aligned}\nabla_x \mathcal{L}(x, \lambda) &= \nabla f(x) + (\nabla C(x))^T \lambda = 0 \\ \nabla_\lambda \mathcal{L}(x, \lambda) &= C(x) = 0\end{aligned}$$

Now we can solve this via Newton's method (using firt-order taylor series expansion):

$$\begin{aligned}\nabla_x \mathcal{L}(x + \Delta x, \lambda + \Delta\lambda) &\approx \nabla_x \mathcal{L}(x, \lambda) + \frac{\partial}{\partial x}\left(\nabla_x \mathcal{L}(x, \lambda)\right)\Delta x + \frac{\partial}{\partial \lambda}\left(\nabla_x \mathcal{L}(x, \lambda)\right)\Delta\lambda \\ &= \nabla f(x) + (\nabla C(x))^T \lambda + \left[\nabla^2 f(x) + \sum_i \lambda_i \nabla^2 C_i(x)\right]\Delta x + (\nabla C(x))^T \Delta\lambda\end{aligned}$$

$$\nabla_\lambda \mathcal{L}(x + \Delta x, \lambda + \Delta\lambda) \approx C(x) + \frac{\partial C}{\partial x}\Delta x = 0$$

Then, we derive the above system of equations in matrix form:

$$\begin{bmatrix} \nabla^2 f(x) & \left(\frac{\partial C(x)}{\partial x}\right)^T \\ \frac{\partial C(x)}{\partial x} & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L}(x, \lambda) \\ -C(x) \end{bmatrix}$$

These conditions are called the **KKT conditions** (Karush-Kuhn-Tucker).

Note: It is really annoying to compute $\frac{\partial^2 \mathcal{L}}{\partial x^2}$ exactly because $\frac{\partial^2 \mathcal{L}}{\partial x^2} = \left[\nabla^2 f(x) + \sum_i \lambda_i \nabla^2 C_i(x)\right]$ involves second derivatives and curvature terms, which are often expensive to compute. We often drop the second "curvature constraint" term to make computations easier. However, when we do this, we get slightly slower convergence than full Newton's method, but each iteration is cheaper. If we can't drop the curvature term, we may need to regularize $\frac{\partial^2 \mathcal{L}}{\partial x^2}$ even if $\nabla^2 f > 0$.

2.2 Inequality Constraints

This time the optimization problem becomes:

$$\min_x f(x) \quad \text{s.t.} \quad C(x) \leq 0$$

For this problem:

- First-order necessary conditions:
  1. $\nabla f = 0$ in the free directions.
  2. Satisfy the inequality constraints: $C(x) \leq 0$.

2.2.1 KKT Conditions (for Inequality Constraints)

1. Stationarity:
$$\nabla f + \left(\frac{\partial C(x)}{\partial x}\right)^T \lambda = 0$$
2. Primal feasibility:
$$C(x) \leq 0$$
3. Dual feasibility:
$$\lambda \geq 0$$
4. Complementarity:
$$\lambda^T C(x) = 0$$

These conditions imply:

- If the constraint is *active* ($C(x) = 0$), then $\lambda > 0$ (same as equality case).
- If the constraint is *inactive* ($C(x) < 0$), then $\lambda = 0$ (same as the unconstrained case).
- Complementary encodes the "on/off" switching of constraints.

2.2.2 Optimization Algorithms for Solving these KKT Conditions

Active-Set Method

- Guess the active/inactive constraints.
- Solve the resulting constrained (equality) problem.
- Used when you have good heuristics for the active set.

Barrier/Interior-Point Method

- Take all your inequalities and replace them with a barrier function in the objective:
$$\min_x f(x) \quad \text{s.t.} \quad C(x) \leq 0 \quad \rightarrow \quad \min_x f(x) - \frac{1}{\rho}\sum_i \log(-C_i(x))$$
- **Note:** This is the gold standard for convex problems.

Penalty Methods

- Usually performs worse; has issues with ill-conditioning.
- Replace the constraints with a term that penalizes violation:
$$\min_x f(x) \quad \text{s.t.} \quad C(x) \leq 0 \quad \rightarrow \quad \min_x f(x) + \frac{\rho}{2}\left(\max(0, C(x))\right)^2$$

Augmented Lagrangian

The Augmented Lagrangian method adds a Lagrange multiplier estimate to the penalty method. The optimization problem becomes:

$$\min_x f(x) + \tilde{\lambda}^T C(x) + \frac{\rho}{2} \left(\max(0, C(x))\right)^2$$

or

$$\min_x \mathcal{L}_p(x, \tilde{\lambda})$$

where:

$$\mathcal{L}_p(x, \tilde{\lambda}) = \text{Augmented Lagrangian}$$

The Lagrange multipliers $\tilde{\lambda}$ are updated by "off-loading" the penalty term into $\tilde{\lambda}$ at each iteration.

$$\frac{\partial}{\partial x}\left(f(x) + \tilde{\lambda}^T C(x) + \frac{\rho}{2} C(x)^2\right) = 0$$

$$\nabla f(x) + \left[\tilde{\lambda} + \rho C(x)\right]^T \nabla C(x) = 0$$

$$\Rightarrow \quad \tilde{\lambda} \leftarrow \tilde{\lambda} + \rho C(x) \quad \text{For active constraints.}$$

Pseudo Code
Repeat until convergence:

1. Solve the minimization problem:
$$\min_x \mathcal{L}_p(x, \tilde{\lambda})$$
2. Update the Lagrange multipliers:
$$\tilde{\lambda} \leftarrow \max(0, \tilde{\lambda} + \rho C(x))$$
(Clamping to make $\tilde{\lambda} \geq 0$.)
3. Update the penalty parameter:
$$\rho \leftarrow \alpha \rho \quad (\alpha \text{ is typically } 10)$$

Advantages:

- Fixes ill-conditioning of the penalty method.
- Converges with a finite ρ.
- Works well on non-convex problems.

Example on Augmented Lagrangian: Consider the canonical problem;

$$\min_x f(x) \quad \text{s.t.} \quad C(x) = 0 \;\&\; H(x) \leq 0$$

I. Form Lagrangian:

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T C(x) + \mu^T H(x)$$

II. The KKT conditions are:

1. Stationarity:
$$\nabla_x \mathcal{L}(x, \lambda, \mu) = \nabla f(x) + \left(\nabla C(x)\right)^T \lambda + \left(\frac{\partial H}{\partial x}\right)^T \mu = 0$$
2. Primal feasibility:
$$C(x) \leq 0$$
3. Dual feasibility:
$$\mu \geq 0$$

4. Complementarity:

$$\mu^T H(x) = 0$$

NOTE: Any (x, λ, μ) that satisfies these KKT conditions is globally optimal solution. (Only true for convex problems.)

III. Augmented Lagrangian:

$$\mathcal{L}_\rho(x, \lambda, \mu, \rho) = \mathcal{L}(x, \lambda, \mu) + \frac{\rho}{2}||C(x)||_2^2 + \frac{1}{2}H^T(x)I_\rho H(x)$$

where; $||C(x)||_2^2 = C(x)^T C(x)$ and $I_\rho = \begin{bmatrix} I_{11} & & & \\ & I_{22} & & \\ & & \ddots & \\ & & & I_{mm} \end{bmatrix}$ with; $I_\rho(i,i) = \begin{cases} 0 & \text{if } H_i(x) < 0 \text{ and } \mu_i = 0 \\ \rho & \text{otherwise} \end{cases}$

Algorithm:

1. Initialize $x_0 = 0$, $\lambda_0 = 0$, $\rho = 1$, and $\beta = 1$.
2. Loop:
   - Solve $\min_x \mathcal{L}_p(x, \lambda, \rho)$.
   - Check KKT conditions.
   - If satisfied, exit; otherwise, update x, λ, and ρ.
3. Update dual variables:
$$\lambda_{k+1} = \lambda_k + \rho C(x_k)$$
4. Update penalty:
$$\rho \leftarrow \alpha\rho \quad (\alpha \text{ is typically } 10)$$
5. Check convergence (KKT conditions) for both primal and dual feasibility.

Quadratic Programming

For $Q > 0$, minimize:

$$\min_x \frac{1}{2}x^T Q x + q^T x \quad \text{s.t.} \quad Ax - b \leq 0, \quad Cx - d = 0$$

- Super useful in controls and can be solved very fast (up to kHz).
- Can be solved using Augmented Lagrangian method.

2.2.3 Regularization on Unconstrained Optimization

Given:

$$\min_x f(x) \quad \text{s.t.} \quad C(x) = 0$$

We might like to turn this problem into:

$$\min_x f(x) + I_\infty(C(x))$$

where I_∞ is called an indicator function, i.e.

$$I_\infty(C(x)) = \begin{cases} 0 & \text{if } C(x) = 0 \\ +\infty & \text{if } C(x) \neq 0 \end{cases}$$

This method is practically terrible, but it gives the same effect as solving:

$$\min_x \max_\lambda f(x) + \lambda^T C(x)$$

whenever $C(x) \neq 0$, the inner max problem blows up.

Similar for inequalities:

$$\min_x f(x) \quad \text{s.t.} \quad C(x) \leq 0$$

can be written as:

$$\min_x f(x) + I_\infty^+(C(x)) \quad \text{where} \quad I_\infty^+(C(x)) = \begin{cases} 0 & \text{if } C(x) \leq 0 \\ +\infty & \text{if } C(x) > 0 \end{cases}$$

Thus, we solve:

$$\min_x \max_{\lambda \geq 0} f(x) + \lambda^T C(x)$$

Interpretation: KKT conditions define a saddle point in (x, λ) space. The KKT system should have $\dim(x)$ positive eigenvalues and $\dim(\lambda)$ negative eigenvalues at an optimum. This is called "quasi-definite."
To regularize this KKT system, add a positive regularizer to x and a negative one to λ, i.e.:

$$\begin{bmatrix} \nabla^2 f(x) + \beta I & \left(\frac{\partial C(x)}{\partial x}\right)^T \\ \frac{\partial C(x)}{\partial x} & -\beta I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -C(x) \end{bmatrix}$$

where $\beta > 0$.

Even with the regularization, we can sometimes overshoot the constraints during the optimization process. Therefore, we still need line-search methods.

How do we do a line search on a root-finding problem (i.e., find x^* s.t. $C(x^*) = 0$)? We do that by defining a *merit function* $P(x)$ that measures the distance to the solution.

A standard choice: $P(x) = \frac{1}{2}||C(x)||_2^2$ (or any other norm).
Now, do Armijo on $P(x)$:

```
1: $\alpha \leftarrow 1$
2: **while** $P(x + \alpha\Delta x) > P(x) + b\alpha\nabla P(x)^T \Delta x$ **do**
3:     $\alpha \leftarrow \theta\alpha$
4: **end while**
5: $x \leftarrow x + \alpha\Delta x$ =0
```

2.2.4 Merit Functions in Constrained Optimization

Now, consider the constrained optimization problem:

$$\min_x f(x) \quad \text{s.t.} \quad C(x) \leq 0, \quad d(x) = 0$$

The augmented Lagrangian is given by:

$$\mathcal{L}_p(x, \lambda, \mu) = f(x) + \lambda^T C(x) + \mu^T d(x)$$

There are many options for merit functions. For example:

$$P(x, \lambda, \mu) = \frac{1}{2} \left\| r_{\mathcal{KKT}}(x, \lambda, \mu) \right\|^2 \quad \text{(residuals)}$$

Here, the residuals come from the KKT conditions, i.e.,

$$\text{Residuals: } \begin{bmatrix} \nabla_x \mathcal{L} \\ \min(0, C(x)) \\ d(x) \end{bmatrix}$$

Since this can be expensive to compute, another possible merit function is:

$$P(x, \lambda, \mu) = f(x) + \rho \left\| \begin{bmatrix} \min(0, C(x)) \\ d(x) \end{bmatrix} \right\|_1$$

where ρ is a scalar weight, and any norm can be used.

Note:

- Excessive constraint penalties can cause problems.
- The augmented Lagrangian method comes with a merit function for free, i.e., $P(x, \lambda, \mu) = \mathcal{L}_p(x, \lambda, \mu)$.

---

3 Deterministic Optimal Control

Continuous Time Case

The problem formulation is:

$$\min_{x(t),u(t)} J(x(t), u(t)) = \int_{t_0}^{t_f} l(x(t), u(t))dt + l_f(x(t_f)) \quad \text{s.t} \quad \dot{x}(t) = f(x(t), u(t))$$

(possibly with other constraints).

- The solution to this problem is an **open-loop** state and control trajectory. We don't get a feedback policy.
- In stochastic settings, we **do** get a feedback policy.
- There are a handful of solutions that have analytic form, but in most cases, we compute the solution numerically.

Discrete Time Case

The discrete-time case formulation is:

$$\min_{x_{1:N},u_{1:N-1}} J(x_{1:N}, u_{1:N-1}) = \sum_{n=1}^{N-1} l(x_k, u_k) + l_f(x_N) \quad \text{s.t.} \quad x_{n+1} = f(x_n, u_n), \quad u_{\min} \leq u_k \leq u_{\max}, \quad C(x_n) \leq 0$$

(e.g., torque limits, obstacle/safety constraints).

- This is a finite-dimensional problem, unlike the continuous case.
- Samples x_n, u_n are often called "knot points."
- Conversion strategies:
  - **Continuous** $\rightarrow$ **Discrete**: Use integration methods (e.g., Runge-Kutta).
  - **Discrete** $\rightarrow$ **Continuous**: Use interpolation.

3.1 Pontryagin's Maximum/Minimum Principle

- Whether you're minimizing or maximizing depends on the problem (e.g., reward maximization in RL).
- At a high level, Pontryagin's principle is simply the KKT conditions for optimal control problems (for the discrete-time case).
- These are the **first-order necessary conditions** for a deterministic optimal control problem.

Given the problem:

$$\min_{x_{1:N}, u_{1:N-1}} \sum_{n=1}^{N-1} l(x_n, u_n) + l_f(x_N) \quad \text{subject to} \quad x_{n+1} = f(x_n, u_n)$$

Pontryagin's principle doesn't play well with inequality/state constraints. Now, define the Lagrangian:

$$\mathcal{L} = \sum_{n=1}^{N-1} l(x_n, u_n) + \lambda_{n+1}^T (f(x_n, u_n) - x_{n+1}) + l_f(x_N)$$

This result is usually stated in terms of the Hamiltonian:

$$H(x, u, \lambda) = l(x, u) + \lambda^T f(x, u)$$

Plug Hamiltonian to the Lagrangian

$$\mathcal{L} = \big(l(x_1, u_1) + \lambda_2^T f(x_1, u_1)\big) + \sum_{n=2}^{N-1} \big(l(x_n, u_n) + \lambda_{n+1}^T f(x_n, u_n) - \lambda_n^T x_n\big) + l_f(x_N) - \lambda_N^T x_N$$

$$\mathcal{L} = H(x_1, u_1, \lambda_2) + \left[\sum_{n=2}^{N-1} H(x_n, u_n, \lambda_{n+1}) - \lambda_n^T x_n\right] + l_f(x_N) - \lambda_N^T x_N$$

Fun Fact: In discrete-time, integration by parts is simply index manipulation. If you were to do the above operation in continuous-time, you'd have to solve integration by parts.

Now, take derivatives with respect to x and λ:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \lambda_n} &= \frac{\partial H}{\partial \lambda_n} - x_{n+1} = f(x_n, u_n) - x_{n+1} = 0 \\
\frac{\partial \mathcal{L}}{\partial x_n} &= \frac{\partial H}{\partial x_n} - \lambda_n^T = \frac{\partial l}{\partial x_n} + \lambda_{n+1}^T \frac{\partial f}{\partial x_n} - \lambda_n^T = 0 \\
\frac{\partial \mathcal{L}}{\partial x_N} &= \frac{\partial l_F}{\partial x_N} - \lambda_N^T = 0 \\
\frac{\partial H}{\partial u} &: \quad \text{This gives the control law for minimizing } H.
\end{aligned}$$

For u, we write the minimization explicitly to handle torque limits:

$$u_n = \arg\min_{u \in \mathcal{U}} H(x_n, u, \lambda_{n+1}) \quad \text{s.t.} \quad u \in \mathcal{U}$$

This is because we always have a limit on control input. In Summary;

- $x_{n+1} = \nabla_\lambda H(x_n, u_n, \lambda_{n+1})$: Gives you the dynamics (forward).
- $\lambda_n = \nabla_x H(x_n, u_n, \lambda_{n+1})$: Propagates the multiplier backwards.
- $u_n = \arg\min_{u \in \mathcal{U}} H(x_n, u, \lambda_{n+1})$.

- $\lambda_N = \frac{\partial l_F}{\partial x_n}$

In continuous time, these can be stated as:

$$\dot{x} = \nabla_\lambda H(x, u, \lambda)$$

$$\dot{\lambda} = -\nabla_x H(x, u, \lambda)$$

$$u = \arg\min_{u \in \mathcal{U}} H(x, u, \lambda)$$

$$\lambda(t_f) = \frac{\partial l_f}{\partial x}$$

These equations can also be obtained by doing calculus of variations.

RECAP: Pontryagin's Principle is a fundamental tool in optimal control theory, providing necessary conditions for an optimal control problem. It helps in finding the control law that minimizes (or maximizes) a given performance index while satisfying the system dynamics and constraints.

Solution Procedure:

- **Define the Hamiltonian** H, based on system dynamics and cost function.
- **Set up and solve the state dynamics and Lagrangian differential equations**: Often involves solving a two point B.V. problem, as initial conditions are given for state variables and terminal conditions for the co-state variables.
- **Determine the optimal control law** u by minimizing or maximizing H.
- **Verify** that the necessary conditions are met in $[t_0, t_f]$.

Example: Solve the following problem:

$$\min_u J(u) = \int_0^1 (x(t)^2 + u(t)^2)dt \quad \text{s.t.} \quad \dot{x}(t) = u(t), \quad x(0) = 1$$

Solution:
(i) Form Hamiltonian;

$$\begin{aligned} H(x, u, \lambda) &= \lambda f(x(t)^2, u(t)^2) + x(t)^2 + u(t)^2 \\ &= \lambda \dot{x}(t) + x(t)^2 + u(t)^2 \\ &= \lambda u(t) + x(t)^2 + u(t)^2 \end{aligned}$$

(ii) Pontryagin's Principle;
State equation:

$$\dot{x}(t) = \frac{\partial H}{\partial \lambda} = u(t)$$

Co-state equation:

$$\dot{\lambda}(t) = \frac{-\partial H}{\partial x} = -2x(t)$$

Stationary Condition:

$$u^* = \arg\min_{u \in \mathcal{U}} H(x, \mu, \lambda) \Rightarrow \frac{\partial H}{\partial u} = 0$$

$$u^* = \frac{-\lambda(t)}{2}$$

Terminal Cost:

$$\lambda(t_f) = \frac{\partial l_F}{\partial x_N}$$

Since, there is no terminal cost associated with this problem; $\lambda(1) = 0$

(iii) Solve the state and co-state equations;
The state and co-state equations are:

$$\dot{x}(t) = -\frac{\lambda(t)}{2}, \quad \text{with} \quad x(0) = 1 \quad \text{(B.C.)}$$

$$\dot{\lambda}(t) = -2x(t), \quad \text{with} \quad \lambda(1) = 0 \quad \text{(B.C.)}$$

Now, we can write these as a system of differential equations:

$$\begin{bmatrix} \dot{x}(t) \\ \dot{\lambda}(t) \end{bmatrix} = \begin{bmatrix} 0 & -1/2 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ \lambda(t) \end{bmatrix}$$

Let matrix A represent the system dynamics. We find the eigenvalues and eigenvectors of A, and the solution will be a linear combination of the solutions corresponding to each eigenvalue. Thus, the solution is:

$$\begin{bmatrix} x(t) \\ \lambda(t) \end{bmatrix} = c_1 e^{\lambda_1 t} \begin{bmatrix} 1 \\ -2 \end{bmatrix} + c_2 e^{\lambda_2 t} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where $\lambda_1 = 1$ and $\lambda_2 = -1$ are the eigenvalues of A. Using boundary conditions:

$$x(0) = c_1 e^0 + c_2 e^0 = c_1 + c_2 = 1$$

$$\lambda(1) = -2c_1 e^1 + 2c_2 e^{-1} = 0$$

Solving for c_1 and c_2:

$$c_1 = \frac{1}{1+e^2}, \quad c_2 = \frac{e^2}{1+e^2}$$

The optimal solution for $x^*(t)$ is:

$$x^*(t) = \left(\frac{1}{1+e^2}\right) e^t + \left(\frac{e^2}{1+e^2}\right) e^{-t}$$

The optimal solution for $u^*(t)$ is:

$$u^*(t) = -\frac{1}{2}\lambda^*(t) = -\frac{1}{2}\left[\left(-\frac{2}{1+e^2}\right) e^t + \left(\frac{2e^2}{1+e^2}\right) e^{-t}\right]$$

$$u^*(t) = \left(\frac{1}{1+e^2}\right) e^t - \left(\frac{e^2}{1+e^2}\right) e^{-t}$$

Interpretation of Optimal Solutions
These optimal solutions will tell you:

- How the state of the system evolves over time from $t = 0$ to $t = 1$ to minimize the total cost.
- What control input $u(t)$ (e.g., force, power, torque, etc.) should be applied at each instant to achieve this optimal state trajectory.
- The minimal value of the cost $J(u)$, indicating the best performance achievable with the optimal control strategy.

3.2 Linear Quadratic Regulator (LQR)

Consider the Linear Quadratic Regulator (LQR) problem:

$$\min_{x_n, u_n} J(x_n, u_n) = \sum_{n=1}^{N-1} \left(\frac{1}{2} x_n^T Q_n x_n + \frac{1}{2} u_n^T R_n u_n \right) + \frac{1}{2} x_N^T Q_N x_N$$

subject to:

$$x_{n+1} = A_n x_n + B_n u_n \quad \text{where} \quad Q \geq 0 \quad \text{and} \quad R > 0$$

In the time-invariant case:

$$A_n = A, \quad B_n = B, \quad Q_n = Q, \quad R_n = R$$

Key Points:

- This formulation can locally approximate many non-linear problems. Smooth non-linear problems can be tackled by Taylor expanding the first and second order and setting up the problem as above.
- In many cases, this method works surprisingly well.
- There are many extensions of LQR: (finite/infinite horizon), (time-variant/time-invariant), (deterministic/stochastic).
- **Fun Fact:** When stabilizing $\rightarrow$ time-invariant. When tracking $\rightarrow$ time-variant

Three Ways of Viewing the LQR Problem

3.2.1 LQR with Indirect Shooting (Pontryagin's Principle)

Note:

$$x_{n+1} = Ax_n + Bu_n$$

$$\lambda_n = Qx_n + A^T \lambda_{n+1}$$

$$\lambda_N = Q_N x_N$$

$$u_n = -R_n^{-1} B_n^T \lambda_{n+1}$$

The Hamiltonian is:

$$H = \frac{1}{2} x_n^T Q x_n + \frac{1}{2} u_n^T R u_n + \lambda_{n+1}^T (Ax_n + Bu_n)$$

Procedure:

1. Start with an initial guess for $u_{1:N-1}$.
2. Simulate/rollout to get $x_{1:N}$.
3. Perform the backward pass to compute λ and Δu.
4. Rollout with a line search (Armijo) on Δu.
5. Go to 3 until convergence.

Historical Context: Back in the 1960s-80s, people used to do this numerical optimization by guessing initial state x_0 and λ_N, then integrate these equations forward and backward. This essentially amounts to doing gradient descent.

Example: Double Integrator

The state-space equations for the double integrator are:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

where:

- $x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}$ represents position and velocity,
- u represents force applied to the system.

The system represents a frictionless brick sliding on ice.
In discrete time, this becomes:

$$x_{n+1} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} x_n + \begin{bmatrix} \frac{h^2}{2} \\ h \end{bmatrix} u_n$$

where h is the time step.

Remark: These indirect shooting methods don't perform well when the time horizon is long. They tend to be less efficient and are not commonly used in practice for such cases.

Listing 2: Python code for discrete dynamics of Double Integrator

```python
# Discrete Dynamics
h = 0.1  # time-step
A = np.array([[1, h], [0, 1]])
B = np.array([0.5 * h**2, h]).reshape(-1, 1)

n = 2  # number of states
m = 1  # number of controls
T_final = 5.0  # Final time, number of time steps
N = int(T_final/h) + 1  # number of time-steps
t_hist = np.arange(0, h * (N), step = h)  # time-stamps

# Cost weights
Q = np.eye(2)      # for stage cost
R = np.array([[0.01]]).reshape(-1, 1)  # control weight matrix
Q_N = np.eye(2)    # for terminal cost

def J(xhist, uhist, Q, Q_N, R):
    '''Cost function:
    xhist: shape(N, 2, 1)
    uhist: shape(N-1, 1, 1)'''

    x_final = xhist[-1]
    cost = 0.5 * x_final.T @ Q_N @ x_final  # terminal cost

    for n in range(len(xhist) - 1):
        xn = xhist[n]
        un = uhist[n]
        cost += 0.5 * (xn.T @ Q @ xn + un.T @ R @ un)

    return cost

def rollout(xhist, uhist, A, B):
    '''Simulate Dynamics'''
    xnew = np.zeros_like(xhist)
    xnew[0] = xhist[0]

    for n in range(len(xhist) - 1):
        xnew[n + 1] = A @ xnew[n] + B @ uhist[n]
    return xnew

# Initial state
x0 = np.array([[1.0, 0.0]]).reshape(2, -1)

# Initial guess
xhist = np.tile(x0, reps=(N, 1, 1))
uhist = np.zeros((N-1, 1, 1))

Delta_u = np.ones_like(uhist)
hhist = np.zeros((N-1, 1, 1))

# Initial rollout to get state trajectory
xhist = rollout(xhist, uhist, A, B)
initial_cost = J(xhist, uhist, Q, Q_N, R).item()
print('Initial Cost:', initial_cost)

b = 1e-2  # line-search tolerance
alpha = 1.0  # alpha for Armijo
iteration = 0

while max(abs(Delta_u)).item() > 1e-2:

    # Backward pass to compute lambda and Delta u
    Ahist[-1] = Q_N @ xhist[-1]
    for k in reversed(range(len(xhist)-1)):
        Ahist[k] = Q @ xhist[k] + A.T @ Ahist[k+1]
        Delta_u[k] = -(uhist[k] + np.linalg.lstsq(R, B.T @ Ahist[k+1], rcond=None)[0])

    # Forward pass with line search to compute x
    alpha = 1.0
    unew = uhist + alpha * Delta_u
    xnew = rollout(xhist, unew, A, B)

    # Line-search with Armijo Rule
    while J(xnew, unew, Q, Q_N, Q, n, R) > J(xhist, uhist, Q, Q_N, Q, n, R) - b * alpha *
        Delta_u.reshape(-1, 1).T @ Delta_u.reshape(-1, 1):
        alpha *= 0.5
        unew = uhist + alpha * Delta_u
        xnew = rollout(xhist, unew, A, B)

    uhist = unew
    xhist = xnew

    iteration += 1

print(f'It took {iteration} iterations to converge')
print(f'Final Cost: {J(xhist, uhist, Q_N, Q, n, R).item():.3f}')
```

3.2.2 LQR as Quadratic Programming

Assume the initial state x_0 is given (not a decision variable). Stack all the states and controls over the trajectory into a long vector:

$$z = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad \text{This vector is going to be the decision variable.}$$

The cost function can be written as:

$$J = \frac{1}{2} z^T H z$$

where H is a diagonal matrix:

$$H = \begin{bmatrix} R_0 & 0 & 0 & \cdots & 0 \\ 0 & Q_1 & 0 & \cdots & 0 \\ 0 & 0 & R_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & Q_N \end{bmatrix}$$

Now, define d and C:

$$d = \begin{bmatrix} Ax_0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} B_0 & -I & 0 & 0 & 0 & \cdots & \cdots & 0 \\ 0 & A_1 & B_1 & -I & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 0 & A_2 & B_2 & -I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & A_{N-1} & B_{N-1} & -I \end{bmatrix}$$

Such that:

$$Cz = d$$

Now we can write the LQR problem as a standard quadratic program (QP):

$$\min_z \frac{1}{2} z^T H z \quad \text{s.t.} \quad Cz = d$$

The Lagrangian is given by:

$$\mathcal{L}(z, \lambda) = \frac{1}{2} z^T H z + \lambda^T (Cz - d)$$

where the gradients are:

$$\nabla_z \mathcal{L} = Hz + C^T \lambda = 0, \quad \nabla_\lambda \mathcal{L} = Cz - d = 0$$

This leads to the system of equations:

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ d \end{bmatrix}$$

This system can be written as:

$$\Omega \omega = \Gamma \quad \text{where} \quad \omega = \begin{bmatrix} z \\ \lambda \end{bmatrix}, \quad \Omega = \begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} 0 \\ d \end{bmatrix}$$

Thus, we get the exact solution by solving one linear system!

Listing 3: Solving LQR via QP

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import kron, identity, csr_matrix, block_diag

# Update C matrix in the loop
def create_matrix(N, A, B):
    I2 = csr_matrix(np.eye(2))  # 2x2 identity matrix

    # Define the block matrix [B -I2]
    block_matrix = np.hstack((B, -I2.toarray()))
    block_matrix = csr_matrix(block_matrix)

    # Create the (N-1)x(N-1) identity matrix
    IN_minus_1 = identity(N-1)

    # Perform the Kronecker product
    C = kron(IN_minus_1, block_matrix, format='csr')
    C = C.toarray()

    # Add A matrix to C (just below -I)
    a = 0
    b = 4
    d = 3
    for n in range(N-2):
        C[a:b, c:d] = A
        a = b
        b += 2
        c = d + 1
        d += 3
    return C

# Discrete Dynamics
h = 0.1
A = np.array([[1, h], [0, 1]])
B = np.array([0.5 * h**2, h]).reshape(-1, 1)

n = 2  # number of states
m = 1  # number of controls

T_final = 5.0
N = int(T_final / h) + 1
t_hist = np.arange(0, h * (N), step = h)

# Cost weights
Q = np.eye(2)
R = np.array([[0.1]]).reshape(-1, 1)
Q_N = np.eye(2)

# Initial condition
x0 = np.array([[1.0, 0.0]]).reshape(2, 1)

# Create matrix H
QR_block = block_diag([Q, R]).toarray()  # type: ignore
IN_minus_2 = np.eye(N-2)
kron_prod = np.kron(IN_minus_2, QR_block)
H = block_diag(R, kron_prod, Q_N).toarray()  # type: ignore

# For constraints C & d
C = create_matrix(N, A, B)  # type: ignore

# Construct vector d
d = np.zeros(C.shape[0]).reshape(-1, 1)
d[0:n] = -A @ x0

# Solve the linear system of equations
Omega = np.block([[H, C.T], [C, np.zeros((C.shape[0], C.T.shape[1]))]])
Gamma = np.hstack([np.zeros((len(Omega) - len(d), 1)), d.flatten()]).reshape(-1, 1)
omega = np.linalg.lstsq(Omega, Gamma, rcond=None)[0]

# Extract z and control history uhist
z = omega.flatten()[:H.shape[1]]
uhist = z[0::3][:0:N-1]

# Extract state history xhist
indices = [i for i in range(len(z)) if (i % 3) != 0]
xhist = z[indices]
xhist = xhist.reshape(len(xhist)//2, 2, -1)

# Extract position and velocity
position = xhist[:, 0].flatten()
velocity = xhist[:, 1].flatten()

# Insert initial condition x0
position = np.insert(position, 0, x0.flatten()[0])
velocity = np.insert(velocity, 0, x0.flatten()[1])

plt.subplot(2,1,1)
plt.plot(t_hist, position, 'b.-', label='Position')
plt.plot(t_hist, velocity, 'r.-', label='Velocity')
plt.xlim([0, 5])
plt.grid()
plt.legend()

plt.subplot(2,1,2)
plt.plot(t_hist[:-1], uhist, 'k.-', label='Control')
plt.xlim([0, 5])
plt.grid()
plt.legend()

plt.show()
```

Remark: Gets the same solution as 'Indirect-Shooting' but way faster.

A closer look at LQR-QP

The KKT system for LQR is very sparse and has a lot of structure:

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ d \end{bmatrix}$$

$$\begin{bmatrix} R & & & & & & B^T & & \\ & Q & & & & & -I & A^T & \\ & & R & & & & & B^T & \\ & & & Q & & & & -I & A^T \\ & & & & R & & & & B^T \\ & & & & & Q_N & & & -I \\ B & -I & & & & & 0 & 0 & 0 \\ & A & B & -I & & & 0 & 0 & 0 \\ & & & A & B & -I & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ x_2 \\ u_2 \\ x_3 \\ u_3 \\ x_4 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -Ax_4 \\ 0 \\ 0 \end{bmatrix}$$

Performing back-substitution on this structure:
1. From the equation $Q_N x_4 - \lambda_4 = 0 \Rightarrow \lambda_4 = Q_N x_4$.
2. Using $Ru_3 + B^T \lambda_4 = 0$:

$$Ru_3 + B^T Q_N x_4 = 0$$
$$Ru_3 + B^T Q_N (Ax_3 + Bu_3) = 0$$
$$u_3 = -\underbrace{(R + B^T Q_N B)^{-1} B^T Q_N A}_{K_3} x_3$$

3. Similarly

$$Qx_3 - \lambda_3 + A^T \lambda_4 = 0$$
$$Qx_3 - \lambda_3 + A^T Q_N x_4 = 0$$
$$Qx_3 - \lambda_3 + A^T Q_N (Ax_3 + Bu_3) = 0$$
$$\lambda_3 = \underbrace{(Q + A^T Q_N (A - BK_3))}_{P_3} x_3 = 0$$

Now we have a recursion for K and P:

$$P_N = Q_N$$
$$K_k = (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$
$$P_k = Q + A^T P_{k+1} (A - BK_k)$$

This is called the **Riccati recursion/equation**.

Notes

- We can solve the QP by doing a **backward Riccati recursion** followed by a forward rollout to compute $x_{1:N}$ and $u_{1:N-1}$.
- A general (dense) QP has complexity $O(N^3(n^3+m^3))$ where:
  - n: State dimension
  - m: Control dimension
  - N: Time horizon
- The Riccati solution has complexity $O(N(n^3+m^3))$.
- **Important:** With the Riccati recursion, we now have a feedback policy $u = -Kx$ instead of an open-loop trajectory.

Infinite-Horizon LQR

- For time-invariant LQR, K matrices converge to constant values.
- For stabilization problems, we usually use a constant K.

The backward recursion for P is:

$$K_k = (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

$$P_k = Q + A^T P_{k+1}(A - BK_k)$$

In the infinite-horizon limit ($k \to \infty$):

$$P_{k+1} = P_k = P_\infty$$

This recursion can be solved as a root-finding or fixed-point problem.
Note: Python/MATLAB/Julia's `DARE` function can compute this for you.

Controllability

- How do we know if LQR will work?
- We already know $Q \geq 0$ and $R > 0$.
- For time-invariant cases, there is a simple answer.

For any initial state x_0; x_N is given by:

$$x_N = A^N x_0 + A^{N-1} B u_{N-1} + A^{N-2} B u_{N-2} + \cdots + B u_0$$

or:

$$x_N = \begin{bmatrix} B & AB & A^2B & \dots & A^{N-1}B \end{bmatrix} \begin{bmatrix} u_{N-1} \\ u_{N-2} \\ \vdots \\ u_0 \end{bmatrix} + A^N x_0$$

Let $C = \begin{bmatrix} B & AB & A^2B & \dots & A^{N-1}B \end{bmatrix}$ be the controllability matrix. Then x_N becomes:

$$x_N = C \begin{bmatrix} u_{N-1} \\ u_{N-2} \\ \vdots \\ u_0 \end{bmatrix} + A^N x_0$$

This is equivalent to a least-squares problem for $u_{0:N-1}$:

$$\begin{bmatrix} u_{N-1} \\ u_{N-2} \\ \vdots \\ u_0 \end{bmatrix} = C^T(CC^T)^{-1}\left(x_N - A^N x_0\right)$$

For CC^T to be invertible:

$$\text{rank}(C) = n, \quad \text{where } n = \dim(x)$$

This uses the **pseudo-inverse**.

- Depending on the time-step, the matrix C can be arbitrarily large. Does this mean we need the full C matrix?
- **Answer:** The Cayley-Hamilton theorem tells us we can stop at n time-steps in C because A^N can be written as a linear combination of lower powers of A:

  $$A^N = \sum_{n=0}^{n-1} \alpha_n A^n \quad (\text{for some } \alpha_n)$$

  Therefore, adding more time steps/columns to C doesn't increase its rank:

  $$C = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix}$$

Fun Fact: For the linear time-varying (LTV) case, we don't have a result like this, so we need to check the rank for the full matrix C, i.e., for the entire trajectory.

3.2.3 Bellman's Principle & LQR via Dynamic Programming

Bellman's Principle:

- Optimal control problems have an inherently sequential structure.
- Past control inputs only affect future states; future inputs cannot affect past states.
- **Bellman's Principle (Principle of Optimality)** states that sub-trajectories of optimal trajectories must also be optimal for appropriately defined sub-problems.

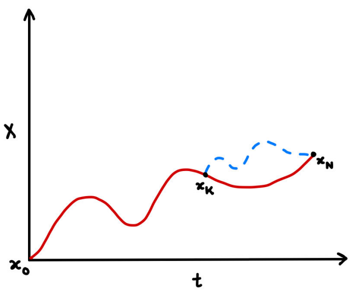
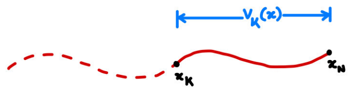


Figure 1: If a path has a lower cost starting at x_k, we would have taken it from x_0. This implies that sub-trajectories of optimal trajectories are also optimal.

Dynamic Programming

- Bellman's Principle suggests starting from the end of the trajectory and working backwards.
- We can see hints of this from the Riccati algorithm.
- Define the "optimal cost-to-go" or **value function** $V_k(x)$: it encodes the cost incurred starting from state x at time k if we act "optimally."



LQR via Dynamic Programming (DP)

For LQR: (by construction: terminal cost).

$$V_N(x) = \frac{1}{2}x^T Q_N x = \frac{1}{2}x^T P_N x$$

Now, back up one step and calculate $V_{N-1}(x)$:

$$V_{N-1}(x) = \min_u \left(\frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u^T R u + V_N(A_{N-1}x_{N-1} + Bu_{N-1}) \right)$$

Take the gradient of $V_{N-1}(x)$ with respect to u and set it to zero:

$$\frac{\partial}{\partial u}V_{N-1}(x) = Ru + B^T P_N(Ax_{N-1} + Bu) = 0$$

Solving for u_{N-1}:

$$u_{N-1} = -\underbrace{(R + B_{N-1}^T P_N B_{N-1})^{-1} B_{N-1}^T P_N A_{N-1}}_{K_{N-1}} x_{N-1}$$

Now plug $u = -Kx$ back into the expression for $V_{N-1}(x)$:

$$V_{N-1}(x) = \frac{1}{2}x^T \underbrace{\left[Q + K^T R K + (A - BK)^T P_N (A - BK)\right]}_{P_{N-1}} x$$

Thus:

$$V_{N-1}(x) = \frac{1}{2}x^T P_{N-1} x$$

Now we have a backward recursion for K and P that we iterate until $k = 0$.

(In general) Dynamic Programming Algorithm

- The cost-to-go (value) is initialized at N with the terminal cost $V_N(x) = l_N(x)$.
- The dynamic programming recursion is:

$$V_k(x) = \min_{u \in \mathcal{U}} \{l(x,u) + V_{k+1}(f(x,u))\}$$

for $k = N-1, N-2, \ldots, 1$.
If we somehow know $V_k(x)$, the optimal policy is:

$$u_k^*(x) = \arg\min_{u \in \mathcal{U}} \{l(x,u) + V_{k+1}(f(x,u))\}$$

Dynamic programming (DP) equations can also be written in terms of the **action-value** or **Q-function**:

$$S_k(x,u) = l(x,u) + V_{k+1}(f(x,u))$$

which avoids the need for an explicit dynamics model. Here $S(x,u)$ is usually denoted by $Q(x,u)$.

The Curse of Dimensionality

- Dynamic Programming is sufficient for global optima, but:
  - Only tractable for simple problems (e.g., LQR, low-dimensional systems).
  - $V(x)$ remains quadratic for LQR but becomes analytically intractable for simple non-linear problems.
  - Even if we could write $min\ S(x, u)$, it would be non-convex and hard to solve.
  - The cost of DP scales exponentially with the state dimension due to the complexity of representing $V(x)$.

Why Do We Care?

- Approximate DP with a function approximator for $V(x)$ or $S(x, u)$ is powerful and forms the basis for many reinforcement learning (RL) methods.
- DP generalizes to stochastic problems (just wrap everything in $\mathbb{E}[\ \cdot\ ]$); Pontryagin doesn't.

Lagrange Multipliers in Dynamic Programming

Recall from the Riccati derivation in the quadratic programming (QP) setup:

$$\lambda_n = P_n x_n$$

From DP:

$$V(x) = \frac{1}{2} x^T P x \Rightarrow \lambda_k = \nabla_x V_k(x)$$

Dynamics multipliers are cost-to-go gradients. This concept carries over to non-linear settings, not just LQR.

3.3 Convex Model Predictive Control (MPC)

- LQR is very powerful, but we often need to reason about constraints.
- Often, these are simple (e.g., torque limits).
- Constraints break the Riccati solution, but we can still solve the QP online.
- Convex MPC has become popular as computers have gotten faster. It was originally used in the 1970s in chemical plants, where the reaction rate of chemical processes was slow.

Examples of Convex Sets:

- Linear Subspace ($Ax = b$)
- Half-space/Box/Polytopes ($Ax \leq b$)
- Ellipsoids ($x^T P x \leq p; p > 0$)
- Cones ($||x_{2:n}||_2 \leq x_1; x \in \mathbb{R}^n$), which includes second-order cones.

Convex Function

A function $f(x) : \mathbb{R}^n \to \mathbb{R}$ is convex if its epigraph is a convex set. Examples:

- Linear function $f(x) = c^T x$
- Quadratic function $f(x) = \frac{1}{2} x^T Q x + g^T x$, where $Q \geq 0$
- Norms: $f(x) = ||x||$ (any norm)

Convex Optimization Problems

Convex optimization problems involve minimizing a convex function over a convex set. Examples:

- Linear Program (LP): Linear $f(x)$; Linear $C(x)$
- Quadratic Program (QP): Quadratic $f(x)$; Linear $C(x)$
- Quadratically Constrained QP (QCQP): Quadratic $f(x)$; Ellipsoidal $C(x)$
- Second-Order Cone Program (SOCP): Linear $f(x)$; Cone $C(x)$

Note: Convex optimization problems do not have spurious local optima and satisfy KKT conditions.

- If you have a local KKT solution, you have the global solution.
- Newton's method converges reliably and quickly (5–10 iterations).
- We can bound the solution time for real-time control.

Convex MPC

Think of it as "Constrained LQR." From dynamic programming, if we have a value function $V(x)$, we can obtain u_k by solving a one-step problem:

$$u_k^* = \arg\min_u \left(l(x,u) + V_{k+1}(f(x_n,u))\right)$$

For LQR, this is:

$$u_k^* = \arg\min_u \frac{1}{2}u^T R u + (Ax_k + Bu_k)^T P_{k+1}(Ax_k + Bu_k)$$

We can add constraints on u to this one-step problem, but this will perform poorly because $V(x)$ was computed without constraints. There is no reason that we cannot add more steps to this one-step problem.

$$\min_{x_{1:H}, u_{1:H-1}} \sum_{k=1}^{H-1} \left(\frac{1}{2}x_k^T Q x_k + \frac{1}{2}u_k^T R u_k\right) + x_H^T P_H x_H$$

where $x_H^T P_H x_H$ is the LQR cost-to-go (value).

- $H \ll N$ is called the "horizon."
- With no additional constraints, MPC with a "receding-horizon" exactly matches LQR for any H.
- **Intuition:** Explicit constrained optimization over the first H steps gets the state close enough to the reference that constraints are no longer active, and the LQR value function $V(x)$ is valid farther into the future.
- In general, a good approximation of $V(x)$ is important for good performance:
  - Better $V(x) \Rightarrow$ shorter horizon.
  - Longer $H \Rightarrow$ less reliance on $V(x)$.

Example: Planar Quadrotor

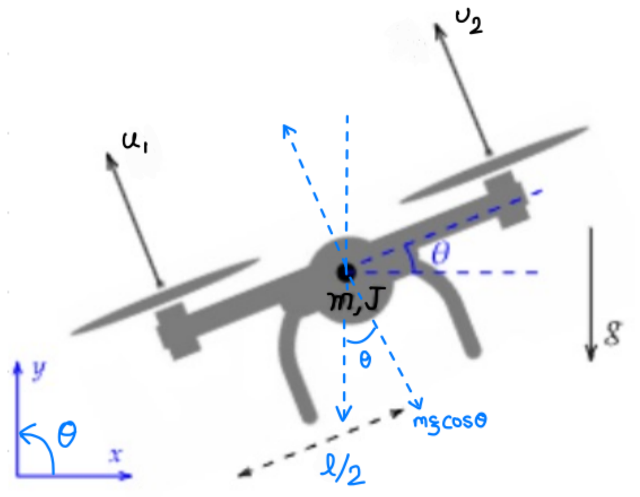


Figure 2: A planar quadrotor

The dynamics of a planar quadrotor are given by:

$$m\ddot{x} = -(u_1 + u_2)\sin\theta$$

$$m\ddot{y} = (u_1 + u_2)\cos\theta - mg$$

$$J\ddot{\theta} = \frac{l}{2}(u_2 - u_1)$$

Linearize about Hover

To linearize the system, we introduce small perturbations around the equilibrium:

$$x = x_0 + \Delta x, \quad y = y_0 + \Delta y, \quad \theta = 0 + \Delta\theta$$

$$u_1 = u_{1_0} + \Delta u_1, \quad u_2 = u_{2_0} + \Delta u_2$$

Given the equilibrium condition $u_{1_0} + u_{2_0} = mg$, we can approximate the sine and cosine functions for small angles $\Delta\theta$:

$$\sin(\Delta\theta) \approx \Delta\theta, \quad \cos(\Delta\theta) \approx 1$$

Substituting these into the equations of motion and neglecting higher-order terms:

$$m\Delta\ddot{x} \approx -(u_{1_0} + \Delta u_1 + u_{2_0} + \Delta u_2)\sin(0 + \Delta\theta)$$

or,

$$m\Delta\ddot{x} \approx -(u_{1_0} + u_{2_0})\Delta\theta + (\Delta u_1 + \Delta u_2)\sin(0) = -mg\Delta\theta$$

$$m\Delta\ddot{y} \approx (u_{1_0} + \Delta u_1 + u_{2_0} + \Delta u_2)\cos(0 + \Delta\theta) - mg$$

or,

$$m\Delta\ddot{y} \approx (u_{1_0} + u_{2_0})\cos 0 + (\Delta u_1 + \Delta u_2)\cos(\Delta\theta) - mg$$

or,

$$m\Delta\ddot{y} \approx mg + (\Delta u_1 + \Delta u_2) - mg$$

At hover: $u_{2_0} = u_{1_0}$

$$J\Delta\ddot{\theta} \approx \frac{1}{2}l(u_{2_0} + \Delta u_2 - u_{1_0} - \Delta u_1) = \frac{l}{2}(\Delta u_2 - \Delta u_1)$$

Now,

$$\begin{aligned}
\Delta\ddot{x} &= -g\Delta\theta \\
\Delta\ddot{y} &= \frac{1}{m}(\Delta u_1 + \Delta u_2) \\
\Delta\ddot{\theta} &= \frac{l}{2J}(\Delta u_2 - \Delta u_1)
\end{aligned}$$

Let x be the state vector and u be the control vector. Then, $x = \begin{bmatrix}\Delta x \\ \Delta y \\ \Delta\theta \\ \Delta\dot{x} \\ \Delta\dot{y} \\ \Delta\dot{\theta}\end{bmatrix}$ and $u = \begin{bmatrix}\Delta u_1 \\ \Delta u_2\end{bmatrix}$

The state-space representation is:

$$\dot{x} = \begin{bmatrix}\Delta\dot{x} \\ \Delta\dot{y} \\ \Delta\dot{\theta} \\ \Delta\ddot{x} \\ \Delta\ddot{y} \\ \Delta\ddot{\theta}\end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix}\Delta x \\ \Delta y \\ \Delta\theta \\ \Delta\dot{x} \\ \Delta\dot{y} \\ \Delta\dot{\theta}\end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1/m & 1/m \\ l/2J & -l/2J \end{bmatrix} \begin{bmatrix}\Delta u_1 \\ \Delta u_2\end{bmatrix}$$

MPC Cost Function:

$$J = \sum_{k=1}^{H-1} \frac{1}{2}\left((x_k - x_{\text{ref}})^T Q (x_k - x_{\text{ref}}) + \Delta u_k^T R \Delta u_k\right) + \frac{1}{2}(x_H - x_{\text{ref}})^T P_H (x_H - x_{\text{ref}})$$

3.4 Non-Linear Trajectory Optimization

- Linear approximations often work well; use them if you can.
- Non-linear dynamics make the MPC problem non-convex, which means no convergence guarantee.
- Non-linear MPC can still work well in practice with effort.

$$\min_{x_{1:N}, u_{1:N-1}} J = \sum_{n=1}^{N-1} l_n(x_n, u_n) + l_N(x_N)$$

subject to $x_{n+1} = f(x_n, u_n)$ (non-linear dynamics), $x_0 \in \mathcal{X}$, $x_N \in \mathcal{X}$, and $u_n \in \mathcal{U}$, where constraints and costs may be non-convex. Assume costs and constraints are C^2 (continuous second derivatives).

Differential Dynamic Programming (DDP)

- Non-linear trajectory optimization method based on approximate dynamic programming (DP).
- Uses a second-order Taylor expansion of the value function in DP to compute Newton steps.
- Very fast convergence is possible.
- Can stop early in real-time applications.

Value Function Expansion

$$V_n(x+\Delta x) \approx V_n(x) + \underbrace{p_n^T \Delta x}_{\text{Gradient}} + \frac{1}{2}\underbrace{\Delta x^T P_n \Delta x}_{Hessian}$$

where:

$$p_N = \nabla_x l_N(x), \quad P_N = \nabla_x^2 l_N(x)$$

Action-Value Function Expansion

$$S_n(x+\Delta x, u+\Delta u) \approx S_n(x,u) + \begin{bmatrix} g_x \\ g_u \end{bmatrix}^T \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix} + \frac{1}{2}\begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}^T \begin{bmatrix} G_{xx} & G_{xu} \\ G_{ux} & G_{uu} \end{bmatrix}\begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}$$

Note: The expression above is a second-order Taylor expansion.

Similar to the LQR case:

$$V_{k-1}(x) = \min_{\Delta u}\left[S_{k-1}(x,u) + g_x^T\Delta x + g_u^T \Delta u + \frac{1}{2}\Delta x^T G_{xx}\Delta x + \frac{1}{2}\Delta u^T G_{uu}\Delta u + \frac{1}{2}\Delta x^T G_{xu}\Delta u + \frac{1}{2}\Delta u^T G_{ux}\Delta x\right]$$

Take gradient with respect to Δu and set it equal to zero:

$$\nabla_{\Delta u}[\,\cdot\,] = g_u + G_{uu}\Delta u + G_{ux}\Delta x = 0$$

Solving for Δu:

$$\Delta u_{k-1} = -G_{uu}^{-1}g_u - G_{uu}^{-1}G_{ux}\Delta x = -d_{k-1} - K_{k-1}\Delta x$$

where d_{k-1} is the feedforward term and K_{k-1} is the feedback gain.
Now, plug back into S_{k-1} to get $V_{k-1}(x+\Delta x)$:

$$\begin{aligned} V_{k-1}(x+\Delta x) \approx V_{k-1}(x) + g_x^T\Delta x + g_u^T(-d_{k-1} - K_{k-1}\Delta x) + \frac{1}{2}\Delta x^T G_{xx}\Delta x \\ + \frac{1}{2}(d_{k-1}+K_{k-1}\Delta x)^T G_{uu}(d_{k-1}+K_{k-1}\Delta x) - \frac{1}{2}\Delta x^T G_{xu}(d_{k-1}+K_{k-1}\Delta x) \\ - \frac{1}{2}(d_{k-1}+K_{k-1}\Delta x)^T G_{ux}\Delta x \end{aligned}$$

Updating P_{k-1}:

$$P_{k-1} = G_{xx} + K_{k-1}^T G_{uu}K_{k-1} - G_{xu}K_{k-1} - K_{k-1}^T G_{ux}$$

and updating p_{k-1}:

$$p_{k-1} = g_x - K_{k-1}^T g_u + K_{k-1}^T G_{uu}d_{k-1} - G_{xu}d_{k-1}$$

Matrix Calculus Detour

To compute g and G, we need to use matrix calculus. Given $f(x) : \mathbb{R}^n \to \mathbb{R}^m$, the second-order Taylor expansion depends on the dimension:

- If $m = 1$:

$$f(x+\Delta x) \approx f(x) + \frac{\partial f}{\partial x}\Delta x + \frac{1}{2}\Delta x^T \frac{\partial^2 f}{\partial x^2}\Delta x$$

- If $m > 1$:

$$f(x+\Delta x) \approx f(x) + \underbrace{\frac{\partial f}{\partial x}}_{\mathbb{R}^{m\times n}}\Delta x + \frac{1}{2}\underbrace{\left(\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}\Delta x\right)\right)}_{\text{ugly !}}\Delta x$$

For $m > 1$, $\frac{\partial^2 f}{\partial x^2}$ is a third-rank tensor. Think of a '3D-matrix'.
To handle this, we use the vectorization operator:

$$A_{l\times m} = \begin{bmatrix} A_1 & A_2 & \dots & A_m \end{bmatrix}, \quad \text{vec}(A) = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_m \end{bmatrix} \in \mathbb{R}^{lm\times 1}$$

- $\text{vec}(ABC) = (C^T \otimes A)\,\text{vec}(B)$
- $\text{vec}(AB) = (B^T \otimes I)\,\text{vec}(A) = I \otimes A)\,\text{vec}(B)$

If we want to differentiate a matrix with respect to a vector, vectorize the matrix:

$$\frac{\partial A(x)}{\partial x} = \underbrace{\frac{\partial \text{vec}(A)}{\partial x}}_{lm\times n}$$

and is implied whenever we differentiate a matrix.

Taylor Expansion of $f(x)$

$$f(x+\Delta x) \approx f(x) + \underbrace{\frac{\partial f}{\partial x}}_{A} \Delta x + \frac{1}{2}(\Delta x^T \otimes I) \underbrace{\frac{\partial^2 f}{\partial x^2}}_{\frac{\partial \text{vec}(A)}{\partial x}} \Delta x$$

where $\frac{\partial}{\partial x}\left[\text{vec}(IA_x\Delta x)\right]$. Sometimes, we also need to worry about the transpose when differentiating through a transpose:

$$\frac{\partial}{\partial x}(A_x^T B) = (B^T \otimes I)T\frac{\partial A}{\partial x} \quad \text{such that} \quad T\,\text{vec}(A) = \text{vec}(A^T)$$

where T is a commutator matrix.

Action-Value Function Derivatives

The action-value function is given by:

$$S_k(x,u) = l_k(x,u) + V_{k+1}(f(x,u))$$

The derivatives with respect to x and u are:

$$\frac{\partial S}{\partial x} = \frac{\partial l}{\partial x} + \frac{\partial V_{k+1}}{\partial f}\underbrace{\frac{\partial f}{\partial x}}_{A} \quad \Rightarrow \quad \frac{\partial S}{\partial u} = \frac{\partial l}{\partial u} + \frac{\partial V_{k+1}}{\partial f}\underbrace{\frac{\partial f}{\partial u}}_{B}$$

$$g_x = \nabla_x l + A^T p_{k+1} \quad \Rightarrow \quad g_u = \nabla_u l + B^T p_{k+1}$$

Tensor-Term Derivatives

$$G_{xx} = \frac{\partial g_x}{\partial x} = \nabla_x^2 l(x,u) + A_k^T P_{k+1} A + (p_{k+1}\otimes I)^T T \frac{\partial A_k}{\partial x}$$

$$G_{uu} = \frac{\partial g_u}{\partial u} = \nabla_u^2 l(x,u) + B_k^T P_{k+1} B_k + (p_{k+1}\otimes I)^T T \frac{\partial B_k}{\partial u}$$

$$G_{xu} = \frac{\partial g_x}{\partial u} = \nabla_{xu}^2 l(x,u) + A_k^T P_{k+1} B_k + (p_{k+1}\otimes I)^T T \frac{\partial A_k}{\partial x}$$

Note: If you throw out the tensor terms, then you get Iterative LQR; otherwise, it is DDP.

3.5 DDP Recap

With DDP, we aim to solve the unconstrained non-linear trajectory optimization problem:

$$\min_{x_{1:N},u_{1:N-1}} J = \sum_{k=1}^{N-1} l(x_k,u_k) + l_N(x_N) \quad \text{s.t.} \quad x_{k+1} = f(x_k,u_k)$$

Then, we perform a backward pass on the value function and expand it using a second-order Taylor expansion:

$$V_k(x+\Delta x) \approx V_k(x) + p_k^T \Delta x + \frac{1}{2}\Delta x^T P_k \Delta x$$

where

$$P_N = \nabla_x^2 l_N(x) \quad \text{and} \quad p_N = \nabla_x l_N(x)$$

Bellman-Backward Pass

Then,

$$V_{k-1}(x+\Delta x) = \min_{\Delta u} S(x+\Delta x, u+\Delta u)$$

to get Δu_{k-1}, p_{k-1}, and P_{k-1}.

Forward Rollout with Armijo

Now, do the forward rollout (pseudo-code with Armijo line search):

Algorithm 3 Forward Rollout with Armijo Line Search

```
    Initialize: $\Delta J \leftarrow 0$, $x'_1 \leftarrow x_1$
    for $k = 1$ to $N-1$ do
3:      $u'_k \leftarrow u_k - \alpha d_k - K_k(x'_k - x_k)$
        $x'_{k+1} \leftarrow f(x'_k, u'_k)$
        $\Delta J \leftarrow \Delta J + \alpha g_{uk} d_k$
6:  end for
    {Line search initialization}
    $\alpha \leftarrow 1$
    repeat
9:      $x', u', \Delta J \leftarrow$ rollout$(x, u, d, K, \alpha)$
        $\alpha \leftarrow c \cdot \alpha$
    until $J(x', u') < J(x, u) - b\Delta J$
12: $x, u \leftarrow x', u'$
    Repeat until $\|d\|_\infty <$ tolerance $=0$
```

Armijo Parameters: $c = 1/2$; $b = 10^{-4} - 0.01$.

- DDP can converge in fewer iterations, but iLQR often wins in wall-clock time.
- Non-convex problems can lead to different local optima depending on the initial guess.

`https://andylee024.github.io/blog/2018/10/10/ddp/`

3.5.1 Regularization

- Just like standard Newton, $V(x)$ and/or $S(x,u)$ Hessians can become indefinite in the backward pass.
- Regularization is definitely necessary for DDP and is often a good idea with iLQR as well.

There are many options for regularizing:

- Add a multiple of the identity to $\nabla^2 S(x, u)$.
- Regularize P_k or G_{xx} as needed in the backward pass.
- Regularize just $G_{uu} = \nabla_u^2 S(x, u)$ (this is the only matrix you have to invert):

$$d = G_{uu}^{-1} g_u, \quad K = G_{uu}^{-1} G_{ux}$$

  Note: This is good for iLQR but not for DDP, because in DDP, your cost-to-go could be indefinite.

- Regularization should not be required for iLQR but can be necessary due to floating-point error.

DDP Notes:

- Can be very fast (both iterations and wall-clock time).
- One of the most efficient methods due to exploitation of DP structure.
- Stays dynamically feasible due to forward rollout (may be sub-optimal but feasible), so it can always execute on a robot.
- Comes with a time-varying LQR tracking controller for free.
- Does not natively handle constraints
- Does not support infeasible initial guesses for state trajectory due to forward rollout.
  Note: This can be problematic for"maze" or"bug-trap" problems.
- Like Riccati recursion, DDP can suffer from ill-conditioning on long trajectories because of floating-point error.

3.5.2 Handling Constraints in DDP

There are many options of handling constraints depending on the type of constraint. Torque limit can be handled with a 'squashing' function e.g. $\tanh(u)$

$$\tilde{u} = u_{max} \tanh\left(\frac{u}{u_{max}}\right)$$

This is effective but adds non-linearity to the dynamic, and may need more iterations. Better option is to solve a box-constrined QP in the backward pass:

$$\Delta u = \arg\min_{\Delta u} s(x + \Delta x, u + \Delta u) \quad s.t. \quad u_{min} \leq u + \Delta u \leq u_{max}$$

State constraints are harder to achieve and often penalties are added to cost function. Do not do this as it can cause ill-conditioning. A better option is to use Augmented-Lagrangian i.e. wrap entire DDP algorithm in an Augmented Lagrangian Method.

3.6 Minimum/Free-time Problems

$$\min_{x(t),u(t),T_F} J = \int_0^{T_F} 1\,dt \quad \text{s.t.} \quad \dot{x} = f(x,u), \quad x(T_F) = x_{\text{goal}}, \quad u_{\min} \le u(t) \le u_{\max}$$

Notes:

- These problems come up fairly often, and in this problem, we're interested in getting to the goal as fast as possible (e.g., drone racing).
- This problem is basically saying to have fewer steps.
- But we don't want to change the number of "knot" points.
- Make h (time step) from Runge-Kutta (RK4) a control input:

$$x_{n+1} = f_{RK}(x_n, \tilde{u}_n) \quad ; \quad \tilde{u}_n = \begin{bmatrix} u_n \\ h_n \end{bmatrix}$$

- Also want to scale the cost by h, e.g.:

$$J(x,u) = \sum_{n=1}^{N-1} h_n\, l(x_n, u_n) + l_N(x_N)$$

- Always nonlinear/non-convex, even if the dynamics are linear.
- Requires constraints on h; otherwise, the solver can "cheat physics" by making h very large or negative to exploit discretization errors.

3.7 Direct Trajectory Optimization

We have seen that indirect methods to solve the optimal control problem require transforming the problem into a Boundary Value Problem (BVP) for a system of differential equations. These methods are closely related to the calculus of variations and Pontryagin's Maximum Principle.

Direct methods take more pragmatic approach by discretizing the control problem directly, transforming it into a finite-dimensional optimization problem that can be solved using standard numerical optimization techniques.

Basic Strategy: Discretize/"transcribe" continuous time optimal control problem into a non-linear program (NLP).

$$\min_x f(x) \quad s.t. \quad c(x) = 0\ ,\ d(x) \le 0$$

where $c(x), d(x)$ are dynamics constraints and other constraints respectively. Here, all functions are assumed to be C^2-smooth. Lots of off-the-shelf solvers are there for large-scale NLP. Most common solvers are IPOPT(Interior-Point), SNOPT(Active-set) and KNITRO(Both).

Common solution strategy: Sequential Quadratic Programming (SQP)
The idea in SQP is to use 2^{nd}-order Taylor expansion of the Lagrangian and linearize $c(x)$ and $d(x)$ to approximate the NLP as a QP.

$$\min_{\Delta x} f(x) + g^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x \quad \text{s.t.} \quad c(x) + C\Delta x = 0 \quad , \quad d(x) + D\Delta x = 0$$

where; $H = \frac{\partial^2 L}{\partial x^2}$, $g = \frac{\partial L}{\partial x}$, $C = \frac{\partial c}{\partial x}$, $D = \frac{\partial d}{\partial x}$ and the Lagrangian

$$\mathcal{L}(x,\lambda,u) = f(x) + \lambda^T c(x) + \mu^T d(x)$$

Now, solve QP to compute primal-dual search direction. $\Delta z = \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta \mu \end{bmatrix}$ and perform line search with merit function. With on the equality constraints, the above problem reduces to Newton's method on KKT conditions:

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -g \\ -c(x) \end{bmatrix}$$

We can think SQP as a generalization of Newton to handle inequalities. We can use any QP solver for sub-problems, but good implementations typically warm start using previous QP iteration. For good performance on trajectory optimization problems, taking advantage of sparsity in KKT systems is crucial. If inequalities are convex, we can generalize SQP to SCP (sequential convex programming) where inequalities are passed directly to the sub-problem solver. SCP is still an active research area.

3.7.1 Direct Collocation

It is a numerical method used in optimal control to discretize and solve continuous time control problems. It is a type of *direct method* that transforms the optimal control problem into a finite-dimensional nonlinear programming (NLP) problem. This approach is particularly powerful for handling complex, nonlinear systems and constraints.

Steps in Direct Collocation

1. **Discretization of Time:** $[0, T]$ is divided into N intervals $(t_0, t_1, \ldots, t_N)$, also called nodes.

2. **Approximation of State and Control:** $x(t)$ and $u(t)$ are approximated by piecewise polynomial functions (Lagrange Polynomials or Splines). For each time interval, $x(t)$ and $u(t)$ are represented as polynomials that pass through the state and control variables at the nodes.

$$x(t) \approx \sum_{j=0}^{m} \phi_j(t) x_j, \quad u(t) \approx \sum_{j=0}^{m} \psi_j(t) u_j$$

   where $\phi_j(t)$ and $\psi_j(t)$ are basis functions, and x_j and u_j are values of the state and control at the collocation points.

3. **Collocation Points:** Within each interval, collocation points are chosen (e.g., Gauss–Lobatto or Gauss–Legendre points) where the system dynamics are enforced. At each collocation point t_c,

$$\dot{x}(t_c) \approx f(x(t_c), u(t_c))$$

4. **Formulation of NLP:**

$$\min \sum_{i=1}^{N-1} J(x_i, u_i) \Delta t_i + \text{terminal cost}$$

   such that:

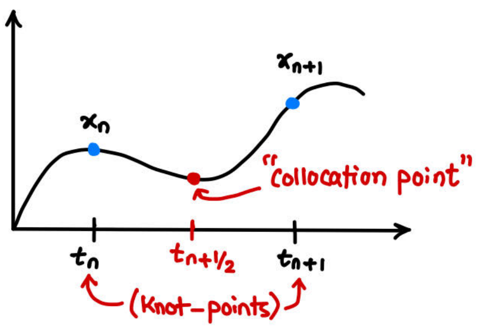
   - Collocation equations (discretized dynamics).
   - State and control constraints at each node.
   - Boundary conditions (initial and terminal states).

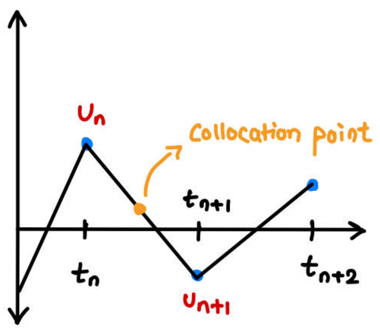
5. **Solution of NLP:** The NLP is solved using numerical optimization solvers like (SQP), interior-point methods, or other techniques. The solution provides the optimal discrete values of the state and control variables at the nodes, which approximates the continuous-time optimal trajectory.

So far, we've used explicit Runge-Kutta methods: $\dot{x} = f(x,u) \Rightarrow x_{n+1} = f(x_n, u_n)$. If we're doing rollouts, this makes sense because we're simulating the dynamics forward in time. In the Direct Collocation method, we're not doing rollouts anymore. However, we're just enforcing dynamics as equality constraints between knot points:

$$C_k(x_k, u_k, x_{k+1}, u_{k+1}) = 0$$

Implicit Integration is "free": Letting the solver do it for free? Collocation methods represent trajectories as polynomial splines and enforce dynamics on spline derivatives. Classic `DIRCOL` algorithm uses cubic splines for states and piecewise linear interpolation for $u(t)$. Very high-order polynomials are sometimes used (e.g., spacecraft trajectories), but this is not common.





How do we go about deriving these things?
Between a pair of knot points:

$$x(t) \approx C_0 + C_1 t + C_2 t^2 + C_3 t^3$$
$$\dot{x}(t) \approx C_1 + 2C_2 t + 3C_3 t^2$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h & h^2 & h^3 \\ 0 & 1 & 2h & 3h^2 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} x_n \\ \dot{x}_n \\ x_{n+1} \\ \dot{x}_{n+1} \end{bmatrix}$$

This gives a good way of interpolating if you know x and $\dot{x}$ at end-points.

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3/h^2 & -2/h & 3/h^2 & -1/h \\ 2/h^3 & 1/h^2 & -2/h^3 & 1/h^2 \end{bmatrix} \begin{bmatrix} x_n \\ \dot{x}_n \\ x_{n+1} \\ \dot{x}_{n+1} \end{bmatrix}$$

Now, we evaluate the spline at the collocation point:

$$x_{n+1/2} = x(t_{n+1/2}) = \frac{1}{2}(x_n + x_{n+1}) + \frac{h}{8}\left(\dot{x}_n + \dot{x}_{n+1}\right)$$

$$= \frac{1}{2}(x_n + x_{n+1}) + \frac{h}{8}\left(f(x_n, u_n) + f(x_{n+1}, u_{n+1})\right)$$

and

$$\dot{x}_{n+1/2} = \dot{x}\left(t_{n+1/2}\right) = -\frac{3}{2h}(x_n - x_{n+1}) - \frac{1}{4}\left(\dot{x}_n + \dot{x}_{n-1}\right)$$

$$= -\frac{3}{2h}(x_n - x_{n+1}) - \frac{1}{4}\left(f(x_n, u_n) + f(x_{n+1}, u_{n+1})\right)$$

also,

$$u_{n+1/2} = u\left(t_{n+\frac{1}{2}}\right) = \frac{1}{2}(u_n + u_{n+1})$$

We can now enforce dynamics constraints:

$$C_i\left(x_n, u_n, x_{n+1}, u_{n+1}\right) = f\left(x_{n+1/2}, u_{n+1/2}\right) - \left[-\frac{3}{2h}(x_n - x_{n+1}) - \frac{1}{4}\left(f(x_n, u_n) + f(x_{n+1}, u_{n+1})\right)\right] = 0$$

- Note that only x_n, u_n are decision variables (not $x_{n+1/2}, u_{n+1/2}$).
- This implicit integration scheme is called **Hermite-Simpson** integration.
- Achieves 3rd order integration accuracy like RK3 would.
- Requires fewer dynamics calls than explicit RK3!
- Since dynamics call often dominate total compute cost; this is a $\sim 50\%$ win.

Explicit RK3 (3 dynamics call per time-step):

$$\begin{aligned} f_1 &= f(x_n, u_n) \\ f_2 &= f(x_n + 1/2\ hf_1, u_n) \\ f_3 &= f(x_n + 2hf_1 - hf_2, u_n) \\ x_{n+1} &= x_n + h/6\ (f_1 + 4f_2 + f_3) \end{aligned}$$

Hermite-Simpson (only 2 dynamics call per time-step):

$$f(x_{n+1/2}, u_{n+1/2}) + \frac{3}{2h}(x_n - x_{n+1}) + \ldots\ldots + \frac{1}{4}\underbrace{\left(f(x_n, u_n) + f(x_{n+1}, u_{n+1})\right)}_{\text{These get re-ued at adjacent steps!}} = 0$$

3.8 Algorithm Recap

Deterministic Optimal Control algorithms
-Linear systems/"local" control: situations where linearization of non-linear problem works.

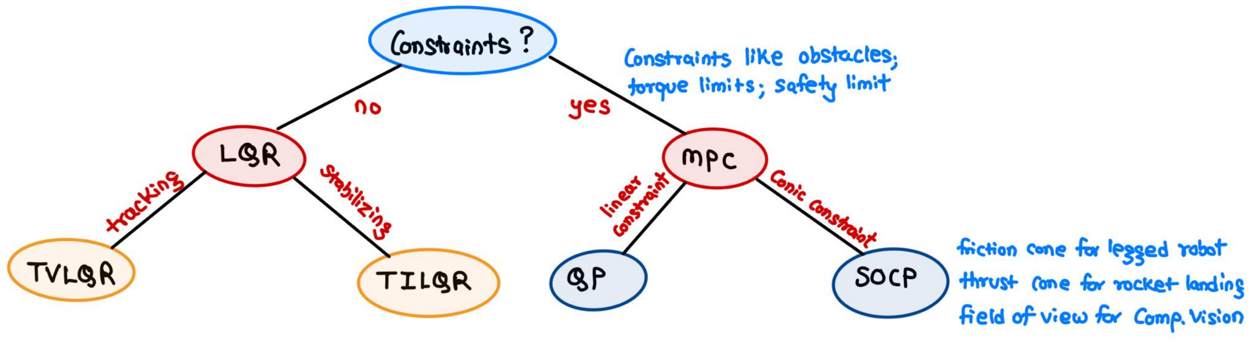

Constraints ?
Constraints like obstacles; torque limits; safety limit
no
yes
LQR
MPC
tracking
Stabilizing
linear constraint
Conic constraint
TVLQR
TILQR
QP
SOCP
friction cone for legged robot
thrust cone for rocket landing
field of view for Comp. Vision

Figure 3: Choosing Control Algorithm

DDP is often a good choice for online or real-time applications where speed is critical and constraint tolerance is not (not ideal for long horizon because Riccati blows-up). `DIRCOL` i often a good choice for offline trajectory design, especially over long horizons and/or with complex constraints.

DIRCOL (direct)	**DDP (Indirect)**
- Only obeys dynamics constraints at convergence.	- Always dynamically feasible.
- Can use infeasible guess so it's useful.	- Can only guess controls; hard to handle constraints.
- Can handle arbitrary constraints.	- Can get TVLQR controller for free.
- Tracking controller must be designed separately.	- Very fast (local) convergence.
- Harder to implement on your own (i.e., large-scale SQP solver).	- Much easier to implement on your own (on embedded systems).
- Numerically robust.	- Not ideally robust (quick and dirty way to implement it yourself)

Table 1: Direct vs Indirect Method

3.9 Control in the case of a Bad system model

Most of the times the model of a system is an approximate and not exact. Simpler model are often preferred even if they are less accurate. Feedback (LQR/MPC) can often compensate for model error. However, sometimes that is not enough, for example in the cases where the constraints are very tight, performance and safety requirements are high.

There exists several options in these cases:

- **(i) Parameter Estimation:** Classical "system identification" / "gray-box" modeling. Fit e.g., masses in your model from data.
  - + Very sample efficient
  - + Generalizes well
  - − Assumes model structure
- **(ii) Learn Model:** Fit a generic function approximator to the full dynamics or residuals. Classical "black-box" modeling / Sys. identification.
  - + Doesn't assume model structure
  - + Generalizes well
  - − Not sample efficient. Requires lots of data (e.g., Neural Nets)
- **Note:** (i)–(ii) are called **indirect adaptive control** and in RL it's called **model-based RL**.
- **(iii) Learn a Policy:** Standard RL approach. Optimize a function approximator for control policy.
  - + Makes few assumptions
  - − Doesn't generalize well
  - − Not sample efficient. Requires lots of "rollouts"
- **(iv) Improve a Trajectory:** Assume we have a reference computed with a nominal model. Improve it with data from the real system. (Like a transfer learning)
  - + Makes few assumptions
  - − Doesn't generalize (task-specific)
  - + Very sample efficient
  - − Assumes a decent prior model

Note: (iii)–(iv) improve the Controller (skips the model and tries to 'dial-in' the controller). In RL, this would be called **policy optimization** or **model-free RL**. Also called **direct adaptive control**.

3.9.1 Iterative Learning Control (ILC)

Imagine a robot that moves an object from 'A' to 'B'. The first time it tries, it makes a mistake and doesn't exactly get it right. **ILC works by letting the robot learn from this mistake**. After each attempt, the robot adjusts its actions slightly based on what went wrong the last time, so it gets closer to perfect the next time. ILC is like a human practicing a skill.
Can think of this as a very specialized policy gradient method on the policy class:

$$u_{n+1}(t) = \underbrace{u_n(t)}_{\text{feed-forward reference input}} - \underbrace{k_n\left(x(t) - x_{\text{ref}}(t)\right)}_{\text{feed-back tracking controller}}$$

where we are updating $u_n(t)$: feed-forward reference (policy parameters).

We can also think of this as SQP (like direct collocation) where we get the R.H.S. vector from a rollout on the real system.

- **Details:** Assume we have a reference trajectory $\bar{x}, \bar{u}$ that we want to track:

$$\min_{x_{1:N}, u_{1:N-1}} J = \sum_{n=1}^{N-1} \frac{1}{2}(x_n - \bar{x}_n)^T Q (x_n - \bar{x}_n) + \frac{1}{2}(u_n - \bar{u}_n)^T R (u_n - \bar{u}_n) + \frac{1}{2}(x_N - \bar{x}_N)^T Q_N (x_N - \bar{x}_N)$$

subject to: $\quad x_{n+1} = f(x_n, u_n)$.

The KKT system for this problem looks like:

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta Z \\ \lambda \end{bmatrix} = \begin{bmatrix} -\nabla J \\ -C(Z) \end{bmatrix}$$

where:

$$\Delta Z = \begin{bmatrix} \Delta x_1 \\ \Delta u_1 \\ \vdots \\ \Delta x_N \end{bmatrix}, \quad C(Z) = \begin{bmatrix} \vdots \\ f(x_n, u_n) - x_{n+1} \\ \vdots \end{bmatrix}, \quad C = \underbrace{\frac{\partial C}{\partial Z}}_{\text{Jacobian}}, \quad H = \underbrace{\begin{bmatrix} Q & & & \\ & R & & \\ & & \ddots & \\ & & & Q_N \end{bmatrix}}_{\text{Gauss-Newton Hessian}}$$

Observations:

1. If we do a rollout on the real system, $C(Z) = 0$ always (for the free dynamics).
2. Since we know J; given x_n, u_n from a rollout, we can compute ∇J.

Now we have RHS vector for the KKT system.

- We also know H from the cost.
- We can compute C or $\partial C / \partial Z$ using x_n, u_n and the nominal model.
- However, since our nominal model is approximate and assuming x_n, u_n is already close to $\bar{x}, \bar{u}$, we can just use $C = \partial C / \partial Z|_{\bar{x},\bar{u}}$ which can be computed offline.
- Now just solve KKT system for ΔZ, update $\bar{u} \leftarrow \bar{u} + \Delta u$ and repeat.
- Can easily add inequality constraints (e.g., torque limits) and solve a QP.

ILC Algorithm

Algorithm 4 Iterative Learning Control (ILC)

```
1: Given: Nominal trajectory $\bar{x}, \bar{u}$
2: while $\|\bar{x}_N - x_N\| >$ tolerance do
3:     Perform Rollout: Execute $\bar{x}_{1:N}, \bar{u}_{1:N-1}$ on the real system
4:     Compute error $\Delta x, \Delta u$ by solving:
           $\arg\min J(\Delta x, \Delta u) \quad \text{s.t. } \Delta x_{n+1} = A_n \Delta x_n + B_n \Delta u_n$
       with constraints: $u_{\min} \leq u_n \leq u_{\max}$
5:     Update control input: $\bar{u} \leftarrow \bar{u} + \Delta u$
6: end while=0
```

Why should ILC work?

We've alredy seen approximations in Newton's method (e.g, Gauss-Newton). In general, these are called 'inexact' and/or 'quasi-newton' methods. Many variants (BFGS, Newton-CG); well developed theory. For a generic root-finding problem:

$$f(x + \Delta x) \approx f(x) + \frac{\partial f}{\partial x} \Delta x = 0$$

As long as Δx satisfies:

$$||f(x) + J(\Delta x)|| \leq \eta ||f(x)||$$

for some $\eta \leq 1$ on inexact Newton method will converge. Convergence ks slower than exact Newton. This means we can use $J \approx \partial d/\partial x$ to compute Δx.

4 Stochastic Optimal Control

So far we assumed we know the system's state perfectly. What happens when we all have are noisy measurements of quantities related to the state? $y = g(X)$ where $g(X)$ is a measuremnt model. In deterministic case, the state is simply X but in stochastic case, the state is conditioned on the measurement i.e. $\mathbb{P}(X|Y)$.

The stochastic optimal control problem is:

$$\min_u \mathbb{E}\left[J(x, u)\right]$$

In principle, we can solve with DP but it suffers from curse of dimensionality and it is hard to write a value function over a high-dimensional space. All of Reinforcement Learning is about solving the above problem approximately.

4.1 Linear Quadratic Gaussian (LQG)

It is a special case of stochastic control problem where you have a closed-form solution.

L - Linear Dynamics, Q - Quadratic Cost, G - Gaussian Noise

Dynamics:

$$x_{n+1} = Ax_n + Bu_n + w_n$$
$$y_n = Cx_n + v_n$$

where $w_n \sim \mathcal{N}(0, W)$ is process-noise and $v_n \sim \mathcal{N}(o, V)$ is measurement-noise.

NOTE: LQG provides a local approximation to nastier non-linear problems. In particular, Kalman Filter comes out of this.

The pdf of a multivariate-Gaussian is given by:

$$p(x) = \frac{1}{\sqrt{(2\pi)^n |S|}} \exp\left[-\frac{1}{2}(x-\mu)^T S^{-1}(x-\mu)\right]$$

where the mean $\mu = \hat{x} = \mathbb{E}[x] \in \mathbb{R}^n$ and the covariance $S = \mathbb{E}[(x-\mu)(x-\mu)^T] \in S^n_{++}$
The cost function in LQG is:

$$J = \mathbb{E}\left[\frac{1}{2}\sum_{n=1}^{N-1}\left(x_n^T Q x_n + u_n^T R u_n\right) + \frac{1}{2}x_N^T Q_N x_N\right] \tag{1}$$

Then, Dynamic Programming Recursion:

$$V_N(x) = \mathbb{E}\left[x_N^T Q x_N\right] = \mathbb{E}\left[x_N^T P_N x_N\right] \tag{2}$$

Now,

$$\begin{aligned}
V_{N-1}(x) &= \min_u \mathbb{E}\left[\frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u_{N-1}^T R u_{N-1} + \frac{1}{2}(Ax_{N-1} + Bu_{N-1} + \omega_{N-1})^T P_N (Ax_{N-1} + Bu_{N-1} + \omega_{N-1})\right] \\
&= \min_u \mathbb{E}\left[\frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u_{N-1}^T R u_{N-1} + \frac{1}{2}(Ax_{N-1} + Bu_{N-1})^T P_N (Ax_{N-1} + Bu_{N-1})\right] + \\
&\quad + \mathbb{E}\left[(Ax_{N-1} + Bu_{N-1})^T P_N w_{N-1} + w_{N-1}^T P_N (Ax_{N-1} + Bu_{N-1}) + w_{N-1}^T P_N w_{N-1}\right]
\end{aligned}$$

where the noise terms contribute only to the cost, not the control policy i.e. noie sample drawn at time n has nothing to do with state (or control) at time n. x_n depends on w_{n-1} (and all past w) but not on w_n or future w. Noise term has no impact on the controller design...you just get a higher cost.

Certainty-Equivalence Principle The optimal LQG controller is essentially the LQR controller with x replaced by its expected value $\mathbb{E}[x]$.

Separation Principle In LQG, you can design an optimal feedback controller and an optimal estimator separately. The combined policy remains optimal.

Neither of these holds in general, but are still frequently used in practice to design sub-optimal policies.

4.2 Optimal State Estimation

Say we have a random noisy measurement y of the system state x and we want to know where we are based on the measurement. In this case, what should we be optimizing?

Maximum a-posteriori (MAP): $\arg\max \mathbb{P}(x|y)$
Minimum Mean Square Error (MMSE): $\arg\min \mathbb{E}\left[(x-\hat{x})^T(x-\hat{x})\right]$

MAP/MMSE are the same for a Gaussian!. Also;

$$\begin{aligned}
\mathbb{E}\left[(x-\hat{x})^T(x-\hat{x})\right] &\equiv \mathbb{E}\left[tr\left((x-\hat{x})^T(x-\hat{x})\right)\right] \\
&= \mathbb{E}\left[tr\left((x-\hat{x})(x-\hat{x})^T\right)\right] \\
&= tr\left(\mathbb{E}[(x-\hat{x})(x-\hat{x})^T]\right) \\
&= tr(\Sigma)
\end{aligned}$$

4.2.1 Kalman Filter

It is a recursive linear MMSE estimator. Assume an estimate of the state that includes all measurements up to the current time:

$$\hat{x}_{k|k} = \mathbb{E}[x_k|y_{1:k}]$$

Assume we also know the error covariance:

$$\Sigma_{k|k} = \mathbb{E}[(x_k - \hat{x}_{k|k})(x_k - \hat{x}_{k|k})^T]$$

We want to update $\hat{x}$ and Σ to include a new measurement at t_{k+1}.

Kalman Filter can be broken into two steps:

1. **Prediction:**

$$\begin{aligned}
\hat{x}_{k+1|k} &= \mathbb{E}[Ax_k + Bu_k + w_k|y_{1:k}] = A\hat{x}_{k|k} + Bu_k \\
\Sigma_{k+1|k} &= \mathbb{E}[(x_{k+1} - \hat{x}_{k+1|k})(x_{k+1} - \hat{x}_{k+1|k})^T] \\
&= \mathbb{E}[(Ax_k + Bu_k w_k - A\hat{x}_{k|k} - Bu_k)(\cdots)^T] \\
&= A\mathbb{E}[(x_k - \hat{x}_{k|k})(x_k - \hat{x}_{k|k})^T]A^T + \mathbb{E}[w_k w_k^T] \\
&= A\Sigma_{k|k}A^T + W
\end{aligned}$$

   x_k and w_k are uncorrelated.

2. **Measurement Update:**
   Define Innovation:

$$\begin{aligned}
z_{k+1} &= y_{k+1} - C\hat{x}_{k+1|k} \quad \text{surprise-error estimate} \\
z_{k+1} &= Cx_{k+1} + \underbrace{\nu_{k+1}}_{measurement-noise} - C\hat{x}_{k+1|k}
\end{aligned}$$

   Innovation Covariance:

$$\begin{aligned}
S_{k+1} &= \mathbb{E}[z_{k+1}z_{k+1}^T] \\
&= \mathbb{E}[(Cx_{k+1} + \nu_{k+1} - C\hat{x}_{k+1|k})(\cdots)^T] \\
&= C\mathbb{E}[(x_{k+1} - \hat{x}_{k+1})(\cdots)^T]C^T + \mathbb{E}[\nu_{k+1}\nu_{k+1}^T] \\
&= C\Sigma_{k+1}C^T + V
\end{aligned}$$

   Innovation is the error signal we feed back into the estimator.

State Update:

$$\hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} + \underbrace{L_{k+1}}_{Kalman\ Gain} z_{k+1}$$

If we are doing old-school pole-placement and not optimizing it; it is called *Luenberger* observer. If we pick the optimal then it is called Kalman gain.

Covariance Update:

$$\begin{aligned}
\Sigma_{k+1|k+1} &= \mathbb{E}[(x_{k+1} - \hat{x}_{k+1})(x_{k+1} - \hat{x}_{k+1})^T] \\
&= \mathbb{E}[(x_{k+1} - \hat{x}_{k+1} - L_{k+1}(cx_{k+1} + \nu_{k+1} - c\hat{x}_{k+1|k}))(\cdots)^T] \\
&= \underbrace{(I - L_{k+1}c)\Sigma_{k+1|k}(I - L_{k+1}c)^T + L_{k+1}VL_{k+1}^T}_{\text{Joseph Form}}
\end{aligned}$$

Kalman Gain:
MMSE $\rightarrow \min \mathbb{E}[(x_{k+1} - \hat{x}_{k+1|k+1})^T(\cdots)] = \min tr(\Sigma_{k+1|k+1})$

set

$$\frac{\partial tr(\Sigma_{k+1|k+1})}{\partial L_{k+1}} = 0$$

and solve for L_{k+1}. Which gives;

$$L_{k+1} = \Sigma_{k+1|k} C^T S_{k+1}^{-1}$$

Kalman Filter Algorithm

Algorithm 5 Kalman Filter Algorithm

```
1: Initialize: $\hat{x}_{0|0}$, $\Sigma_{0|0}$, $W$, $V$ {Gaussian prior, process noise $W$, measurement noise $V$}
2: for $k = 1, 2, \dots$ do
3:     Prediction:
4:     $\hat{x}_{k+1|k} \leftarrow A\hat{x}_{k|k} + Bu_k$
5:     $\Sigma_{k+1|k} \leftarrow A\Sigma_{k|k}A^T + W$
6:     Calculate Innovation and Covariance:
7:     $z_{k+1} \leftarrow y_{k+1} - C\hat{x}_{k+1|k}$
8:     $S_{k+1} \leftarrow C\Sigma_{k+1|k}C^T + V$
9:     Calculate Kalman Gain:
10:    $L_{k+1} \leftarrow \Sigma_{k+1|k}C^T S_{k+1}^{-1}$
11:    Update:
12:    $\hat{x}_{k+1|k+1} \leftarrow \hat{x}_{k+1|k} + L_{k+1}z_{k+1}$
13:    $\Sigma_{k+1|k+1} \leftarrow (I - L_{k+1}C)\Sigma_{k+1|k}(I - L_{k+1}C)^T + L_{k+1}VL_{k+1}^T$
14: end for=0
```

There are variants of Kalman Filter e.g. Extended Kalman Filter (EKF) where we linearize about $\hat{x}$ and proceed as in standard KF. There is also Unscented Kalman Filter (UKF).

4.2.2 Probabilistic Interpretation of Kalman Filter

Consider the following state space model:

$$\begin{aligned} X_t &= AX_{t-1} + q_{t-1} \\ Y_t &= HX_t + r_t \end{aligned}$$

where q_t is the process noise with covariance matrix Q_t and r_t is the sensor noise. Here, $q_t \sim \mathcal{N}(0, Q_t)$ and $r_t \sim \mathcal{N}(0, R)$and are independent of each other.

Assume that the state of the system is normally distribution i.e . $X_{t-1}|Y_{0:t-1} \sim \mathcal{N}(\hat{x}_{t-1}, P_{t-1})$ Let, $Y_{0:t} = y_o, y_1, \dots, y_t$. Furthermore, let X_0 be he initial state of the system such that $X_0 \sim \mathcal{N}(\hat{x}_0, P_0)$.

NOTE: $(\ \cdot\ )_t^-$ denotes a quantity that is calculated at time t without taking the current measurement y_t into consideration.

From a Bayesian perspective, we are interested in the distribution of X_t conditioned on the current available measurements, $\mathbb{P}(X_t|Y_{0:t-1})$ where $Y_{0:t-1} = y_0, y_1, \dots, y_{t-1}$. This distribution can only be Gaussian if $(X_{t-1}, q_{t-1}|Y_{0:t-1})$ is jointly Gaussian.

$$\begin{aligned} \mathbb{P}(x_{t-1}, q_{t-1}|Y_{0:t-1}) &= \frac{\mathbb{P}(x_{t-1}, q_{t-1}, Y_{0:t-1})}{\mathbb{P}(Y_{0:t-1})} \\ &= \frac{\mathbb{P}(x_{t-1}, Y_{0:t-1})\mathbb{P}(q_{t-1})}{\mathbb{P}(Y_{0:t-1})} \\ &= \mathbb{P}(x_{t-1}|Y_{0:t-1})\mathbb{P}(q_{t-1}) \end{aligned} \tag{1}$$

Since, both the terms in the R.H.S of (1) are Gaussian, the L.H.S is also Gaussian. Therefore, the $\mathbb{P}(X_t|Y_{0:t-1})$ is Gaussian too.
Now, we look for the mean and covariance of $(X_t|Y_{0:t-1})$.
Mean is given by the following:

$$\begin{aligned}\mathbb{E}[X_t|Y_{0:t-1}] &= \mathbb{E}[(AX_{t-1}+q_{t-1})|Y_{0:t-1}]\\ &= A\ \mathbb{E}[X_{t-1}|Y_{0:t-1}] + \mathbb{E}[q_{t-1}|Y_{0:t-1}]\\ &= A\hat{x}_{t-1} + \mathbb{E}[q_{t-1}]\\ &= \hat{x}_t^-\end{aligned} \tag{2}$$

Similarly, the covariance for $(X_t|Y_{0:t-1})$ is given by the following:

$$\begin{aligned}P_t^- :=\ & \mathbb{E}[(X_t-\hat{x}_t^-)(X_t-\hat{x}_t^-)^T|Y_{0:t-1}]\\ &= \mathbb{E}[(AX_{t-1}+q_{t-1}-\hat{x}_t^-)(AX_{t-1}+q_{t-1}-\hat{x}_t^-)^T|Y_{0:t-1}]\\ &= A\ \mathbb{E}[(X_{t-1}-\hat{x}_{t-1})(X_{t-1}-\hat{x}_{t-1})^T]\ A^T + 2A\mathbb{E}(X_{t-1}-\hat{x}_{t-1})\mathbb{E}[q_{t-1}]^T + \mathbb{E}[q_{t-1}q_{t-1}]^T\\ &= AP_{t-1}A^T + Q\end{aligned} \tag{3}$$

Now, we would like to know the distribution of X_t conditioned on all the available measurements up to time t i.e. $\mathbb{P}(X_t|Y_{0:t})$
Here,

$$\begin{aligned}\mathbb{P}(x_t|Y_{0:t}) &= \frac{\mathbb{P}(x_t, Y_{0:t})}{\mathbb{P}(Y_{0:t})}\\ &= \frac{\mathbb{P}(x_t, y_t, Y_{0:t-1})}{\mathbb{P}(y_t, Y_{0:t-1})}\\ &= \frac{\mathbb{P}(y_t|(x_t, Y_{0:t-1}))\ \ \mathbb{P}(x_t, Y_{0:t-1})}{\mathbb{P}(y_t|Y_{0:t-1})\ \ \mathbb{P}(Y_{0:t-1})}\\ &= \frac{\mathbb{P}(y_t|(x_t, Y_{0:t-1}))\ \ \mathbb{P}(x_t|Y_{0:t-1})\ \mathbb{P}(Y_{0:t-1})}{\mathbb{P}(y_t|Y_{t-1})\ \ \mathbb{P}(Y_{0:t-1})}\\ &= \frac{\mathbb{P}(y_t|(x_t, Y_{0:t-1}))\ \ \mathbb{P}(x_t|Y_{0:t-1})}{\mathbb{P}(y_t|Y_{0:t-1})}\end{aligned} \tag{4}$$

$\mathbb{P}(X_t|Y_{0:t})$ is Gaussian if every terms in the R.H.S of (4) is Gaussian. We have already proved above that the second term in numerator is Gaussian.

Now, consider the first term.

$$\begin{aligned}\mathbb{E}[Y_t|(X_t, Y_{t-1})] &= \mathbb{E}[(HX_t + r_t)|X_t = x_t, Y_{t-1}]\\ &= \mathbb{E}[(Hx_t + r_t)|X_t = x_t, Y_{t-1}]\\ &= Hx_t\end{aligned} \tag{5}$$

And its covariance is given by;

$$\begin{aligned}\mathbb{E}[(Y_t - Hx_t)(Y_t - Hx_t)^T|Y_{t-1}] &= \mathbb{E}[r_t r_t^T|Y_{t-1}]\\ &= R\\ &= \sigma_n^2 I\end{aligned} \tag{6}$$

Consider the denominator $\mathbb{P}(y_t|Y_{t-1})$. Recall that y_t is a linear combination of X_t and r_t. Therefore, $y_t|Y_{t-1}$ is Gaussian if $(X_t, r_t)|Y_{t-1}$ is jointly Gaussian.

i.e. $\mathbb{P}(X_t, r_t|Y_{t-1}) = \mathbb{P}(r_t|X_t, Y_{t-1})\ \mathbb{P}(X_t|Y_{t-1}) = \mathbb{P}(r_t)\ \mathbb{P}(X_t|Y_{t-1})$. All terms in R.H.S are Gaussian so L.H.S is also Gaussian.

We now look into the statistics (mean and covariance) of this Gaussian distribution:

$$\begin{aligned}\mathbb{E}[y_t|Y_{t-1]}] &= \mathbb{E}[(HX_t + r_t)|Y_{t-1}] \\ &= H\mathbb{E}[X_t|Y_{t-1}] + \mathbb{E}[r_t|Y_{t-1}] \\ &= H\hat{x}_t^-\end{aligned} \tag{7}$$

Similarly;

$$\begin{aligned}\text{cov} &= \mathbb{E}[(y_t - H\hat{x}_t^-)(y_t - H\hat{x}_t^-)^T|Y_{t-1]}] \\ &= \mathbb{E}[(HX_t + r_t - H\hat{x}_t^-)(HX_t + r_t - H\hat{x}_t^-)^T|Y_{t-1}] \\ &= H\ \ \mathbb{E}[(X_t - \hat{x}_t^-)(X_t - \hat{x}_t^-)^T]\ H^T + 2H\ \mathbb{E}[X_t - \hat{x}_t^-]\mathbb{E}[r_t]^T + \mathbb{E}[r_t r_t^T] \\ &= HP_t^- H^T + R \\ &= HP_t^- H^T + \sigma_n^2 I\end{aligned} \tag{8}$$

$$\therefore \quad \mathbb{P}(y_t|y_0, y_1, \ldots, y_{t-1}) \sim \mathcal{N}(H\hat{x}_t^-, HP_t^- H^T + \sigma_n^2 I)$$

4.3 Brief Overview of Game-Theoretic Trajectory Optimization

Assume that are multiple agents and all of them are solving an optimal control. The idea here is to solve a joint optimization problem for all cars simultaneously. One version of this idea is the *Nash Equilibrium*.

Stack all states and inputs together i.e. $\bar{x} = \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^N \end{bmatrix}$ and $\bar{u} = \begin{bmatrix} u^1 \\ u^2 \\ \vdots \\ u^N \end{bmatrix}$ For each agent, we ge a problem of the form:

$$\min_{\bar{x}, u^i} J^i(\bar{x}, u^i) \qquad \text{s.t.} \qquad D(\bar{x}, \bar{u}) = 0, C(\bar{x}, \bar{u}) = 0$$

where cost is associated for each agent and the constraints are global. In this setting we have n number of these problem that we must solve simultaneously. This type of optimization can be interpreted as "No player can unilaterally improve their cost". This is a good model of non-cooperative behavior. Cost functions can capture behavior like aggressiveness etc.

Solution strategy: (AL Games)
Form augmented Lagrangian for each player's first-order necessary conditions.

$$\mathcal{L}^i(\bar{x}, \bar{u}, \lambda^i, u^i) = J^i(\bar{x}, u^i) + \frac{\rho}{2}||D(\bar{x}, \bar{u})||_2^2 + \lambda^{iT} D(\bar{x}, \bar{u}) + \frac{\rho}{2}||C(\bar{x}, \bar{u})||_2^2 + \mu^{iT} C(\bar{x}, \bar{u})$$

$$\implies \nabla u^i \mathcal{L}^i = 0$$

Now, stack first-order necessary conditions for all agents/players. $\begin{bmatrix} \nabla_{u^1}\mathcal{L}^1 \\ \nabla_{u^2}\mathcal{L}^2 \\ \vdots \\ \nabla_{u^N}\mathcal{L}^N \end{bmatrix} = 0$. Then solve with Newton's method and apply standard Augmented Lagrangian update to λ, u.

4.4 PDE & BSDE for Stochastic Optimal Control

An agent controls the state process x through an action u taking values in $\mathbb{R}^d$ and $\mathcal{U}$ respectively, where the dynamics of x are given by the stochastic differential equation (S.D.E.):

$$dx(t) = f(x_t, u_t)\,dt + \sigma(x_t, u_t)\,dW_t \quad \text{with} \quad x(0) = x_0$$

where $f(\cdot)$ and $\sigma(\cdot)$ are the drift term and diffusion (coefficient) term respectively. They are Borel measurable functions. This agent aims to minimize the expected cost:

$$\min_{u \in \mathcal{U}} \mathbb{E}\left[\int_0^T L(x_s, u_s)\,ds + \Phi(x_T)\right]$$

where $L(\cdot,\cdot)$ is the running cost and $\Phi(x_T)$ is the terminal cost. The above problem can be tackled by PDEs or (F)BSDEs.

PDE Approach

When considering Markovian controls (closed-loop), we can define a value function:

$$V(x,t) = \inf_{u \in \mathcal{U}} \mathbb{E}\left[\int_t^T L(x_s, u_s)\,ds + \Phi(x_T)\Big| x_t = x\right]$$

Now employ the dynamic programming principle (DPP):

$$V(x_t, T) = \Phi(x_T) \quad \text{and} \quad V(x_t, t) = \inf_{u \in \mathcal{U}} \mathbb{E}\left[\int_t^\tau L(x_s, u_s)\,ds + V(x_\tau, \tau)\Big| x_t = x\right]$$

Then, we can derive the Hamilton-Jacobi-Bellman (HJB) equation, which describes the evolution of the value function. To derive the HJB equation, we use Itô's lemma to express the infinitesimal change in the value function $V(x_t, t)$ as:

$$dV(x_t,t) = \left(\frac{\partial V}{\partial t} + \nabla_x V \cdot f(x_t, u_t) + \frac{1}{2}\operatorname{Tr}(\sigma\sigma^T \nabla_x^2 V)\right) dt + \nabla_x V \cdot \sigma(x_t, u_t)\,dW_t$$

Taking the expectation and recalling that the expectation of the stochastic integral vanishes, we have:

$$\mathbb{E}\left[dV(x_t,t)\right] = \left(\frac{\partial V}{\partial t} + \nabla_x V \cdot f(x_t, u_t) + \frac{1}{2}\operatorname{Tr}(\sigma\sigma^T \nabla_x^2 V)\right) dt$$

Thus,

$$\frac{\partial V}{\partial t} + \nabla_x V \cdot f(x_t, u_t) + \frac{1}{2}\operatorname{Tr}(\sigma\sigma^T \nabla_x^2 V) + L(x_t, u_t) = 0$$

Define the Hamiltonian as:

$$H(x_t, \nabla_x V, \nabla_x^2 V, u_t) = \nabla_x V \cdot f(x_t, u_t) + \frac{1}{2}\operatorname{Tr}(\sigma\sigma^T \nabla_x^2 V) + L(x_t, u_t)$$

The HJB equation can thus be written as;

$$\frac{\partial V}{\partial t} + \inf_{u \in \mathcal{U}} H(x, u, \nabla_x V) = 0 \qquad s.t. \qquad V(x_T, T) = \Phi(x_T) \tag{*}$$

And the optimal control u^* minimizes the Hamiltonian:

$$u^*(x_t) = \arg\inf_{u \in \mathcal{U}} \nabla_x V \cdot f(x_t, u_t) + \frac{1}{2}Tr(\sigma\sigma^T \nabla_x^2 V) + L(x_t, u_t)$$

BSDE Approach

The value function in the stochastic control problem can often be represented as the solution to a BSDE, providing probabilistic interpretation of the value function and its derivatives.

1. **When volatility is uncontrolled i.e. $\sigma(x_t, u_t)$ is free of u_t**
   In this case u^* does not depend on $\nabla_x^2 V$ and the PDE in $(*)$ is semi-linear. Suppose there exist functions $\tilde{f}(x_t, u_t)$ and $h(x_t, Z_t)$ such that:

$$\nabla_x V \cdot \tilde{f}(x_t, u_t) + h(x_t, \sigma^T(x_t)\nabla_x V) = f(x_t, u_t^*) \cdot \nabla_x V + L(x_t, u_t^*)$$

   Then the non-linear Feynman-Kac formula gives the following BSDE interpretation of $V(x_t, t)$:

$$\begin{cases} d\mathcal{X}_t = \tilde{f}(x_t)\, dt + \sigma(x_t)\, dW_t, & \mathcal{X}_0 \sim \mu_0 \\ d\mathcal{Y}_t = -h(x_t, Z_t)\, dt + Z_t\, dW_t, & \mathcal{Y}_T = \Phi(\mathcal{X}_T) \end{cases}$$

   through the relation:
$$Y_t = V(\mathcal{X}_t, t) \quad \text{and} \quad Z_t = \sigma^T(\mathcal{X}_t)\nabla_x V$$

   Solution Procedure

   - The Hamiltonian in this case is:

$$H(x_t, u_t, \nabla_x V) = \nabla_x V \cdot f(x_t, u_t) + L(x_t, u_t)$$

   - Compute $\nabla_x V$: from BSDE representation, $\nabla_x V(x_t) = \left[\sigma^T(x_t)\right]^{-1} Z_t$
   - Minimize the Hamiltonian:
     - Substitute $\nabla_x V(x_t)$ into the Hamiltonian and solve the minimization problem over u to find $u^*(x_t)$.

2. **In the controlled volatility case**
   The PDE $(*)$ is fully non-linear and its solution is connected to a solution of the second order BSDE. If we choose $\tilde{f}(x_t)$ and $\Sigma(x_t)$ s.t. h is determined by

$$H(x_t, \nabla_x V, \nabla_x^2 V, u_t^*) = \tilde{f}(x_t) \cdot \nabla_x V + h(x_t, \nabla_x V, \nabla_x^2 V) + \frac{1}{2} Tr\left(\Sigma(x_t)\Sigma^T(x_t)\nabla_x^2 V\right)$$

   then the solution to 2BSDE:

$$\begin{cases} d\mathcal{X}_t = \tilde{f}(\mathcal{X}_t)\, dt + \Sigma(\mathcal{X}_t)\, dW_t, & \mathcal{X}_0 \sim x_0 \\ d\mathcal{Y}_t = -h(\mathcal{X}_t, \mathcal{Y}_t, Z_t)\, dt + Z_t^T \Sigma(\mathcal{X}_t)\, dW_t, & \mathcal{Y}_T = \Phi(\mathcal{X}_T) \\ dZ_t = \mathcal{A}_t\, dt + \Gamma_t \Sigma(\mathcal{X}_t)\, dW_t, & Z_T = \nabla_x \Phi(\mathcal{X}_T) \end{cases}$$

   gives an interpretation of the solution to the HJB $-(*)$ through the relations $\mathcal{Y}_t = V(\mathcal{X}_t)$, $Z_t = \nabla_x V(\mathcal{X}_t)$, $\Gamma_t = \nabla_x^2 V(\mathcal{X}_t)$ and $\mathcal{A}_t = \mathcal{L}_x V(\mathcal{X}_t)$. Here $\mathcal{L}$ denotes the infinitesimal generator of $\mathcal{X}$.

   Solution Procedure

   $\rightarrow$ compute $\nabla_x V$ and $\nabla_x^2 V$ from the solution of the second-order BSDE; $Z_t = \nabla_x V(\mathcal{X}_T)$ and $\Gamma_t = \nabla_x^2 V(\mathcal{X}_t)$

   $\rightarrow$ minimize the Hamiltonian: substitute $\nabla_x V$ and $\nabla_x^2 V$ into the Hamiltonian and solve the minimization problem over u to find $u^*(x_t)$.

   e.g. If $f(x_t, u_t)$ and $\sigma(x_t, u_t)$ are affine in u_t i.e. $f(x_t, u_t) = Ax_t + Bu_t$ and $\sigma(x_t, u_t) = \sigma_0 + \Sigma u_t$; the optimal control might be found by solving:

$$u^*(x_t) = \arg\inf_{u \in \mathcal{U}} \left(\nabla_x V \cdot (Ax_t + Bu_t) + \frac{1}{2} \operatorname{Tr}\left((\sigma_0 + \Sigma u)(\sigma_0 + \Sigma u)^T \nabla_x^2 V\right) + \frac{1}{2} u^T R u \right)$$

Summary

1. PDE (HJB) is deterministic; BSDE is probabilistic.
2. PDE focuses on local properties (derivatives) of the value function; BSDE focus on the global evolution of the value function along stochastic paths.
3. PDE relies on analytical or numerical techniques; BSDE uses probabilistic methods, often involving simulations?
4. PDE is traditional and simpler for lower-dimensional, deterministic-like problems. BSDE is better suited for high dimensional, complex, or fully stochastic problems.

5 Deep Learning from a Dynamical System Perspective

(Contents in this section are adapted from [Liu and Theodorou, 2019])

5.1 Information Propagation inside DNN (Fully Connected)

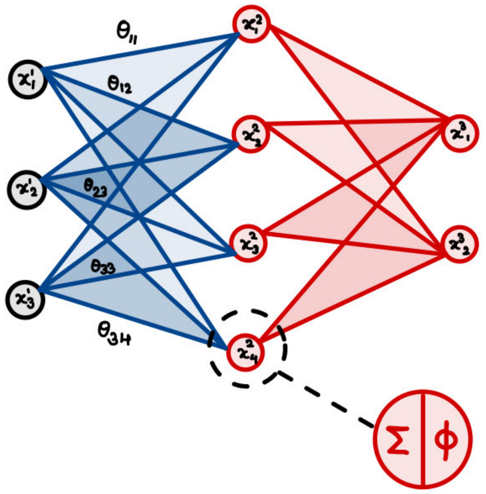


Figure 4: Fully Connected NN

In the figure above, $\Sigma := x_1\theta_{14} + x_2\theta_{24} + x_3\theta_{34} + b = w^T x$ and $\phi = \sigma(w^T x)$ is an activation function. Let $h_l = W(x_l;\theta_l)$ and $x_{l+1} = \phi(h_l)$.

The dynamics at time-step t (or at $t-$th layer of the network):

$$h_t = W(x_t;\theta_t) := W_t x_t + b_t; \qquad W_t^{(i,j)} \sim \mathcal{N}(0, \sigma_w^2/N_t) \quad \& \quad b_t^{(i)} \sim \mathcal{N}(0, \sigma_b^2)$$

Denote $\theta_t \equiv (W_t, b_t)$ and N_t is the dimension of pre-activation at t. Furthermore, as $N_t >> 1$; then each node in the layer is normally distributed i.e. $h_t^{(i)} \sim \mathcal{N}(0, q_t)$. The variance q_t can be estimated by matching the second moment of the empirical distribution of $h_t^{(i)}$ across all N_t i.e. $q_t := \frac{1}{N_t}\sum_{i=0}^{N_t}(h_t^{(i)})^2$ when q_t is propagated forward:

$$q_{t+1} = \sigma_w^2 \mathbb{E}_{h_t^{(i)} \sim \mathcal{N}(0,q_t)}[\phi^2(h_t^{(i)})] + \sigma_b^2$$

with initial condition given by $q_0 = \frac{1}{N_0} x_0 \cdot x_0$

NOTE:

$$h_{t+1} = W_{t+1} x_{t+1} + b_{t+1} = W_{t+1} \phi(h_t) + b_{t+1}$$

then;

$$q_{t+1} = Var(W_{t+1} \cdot \phi(h_t) + b_{t+1}) = \sigma_w^2 \mathbb{E}[\phi^2(h_t)] + \sigma_b^2$$

For any bounded ϕ and finite σ_w and σ_b; there exits a fixed point $q* := \lim_{t \to \infty q_t}$ regardless of initial state q_0

Now consider we have two input pairs $x^{(\alpha)}$ and $x^{(\beta)}$. The variance at the activation is given by

$$q_{t+1}^{(\alpha,\beta)} = \sigma_w^2 \mathbb{E}_{[h_t^{(\alpha)}, h_t^{(\beta)}]^T \sim \mathcal{N}(0, \Sigma_t)}[\phi(h_t^{(\alpha})\phi(h_t^{(\beta)})] + \sigma_b^2$$

where; $\Sigma_t = \begin{bmatrix} q_t^{\alpha} & q_t^{(\alpha,\beta)} \\ q_t^{(\alpha,\beta)} & q_t^{\beta} \end{bmatrix}$ with initial condition $q_0 = \frac{1}{N_0} x_0^{(\alpha)} \cdot x_0^{(\beta)}$. For q^* to exist there must be fixed point for covariance and correlation i.e. $q*_{(\alpha,\beta)}$ and $c^* = \frac{q^*_{(\alpha,\beta)}}{q^*}$. The fixed point covariance is given by;

$$\Sigma^* = \begin{bmatrix} q^* & q^* c^* \\ q^* c^* & q^* \end{bmatrix}$$

or; $\Sigma^*_{(\alpha,\beta)} = q^*(\delta_{(\alpha,\beta)} + (1 - \delta_{(\alpha,\beta)})c^*)$

NOTE: q_{t+1} admits a deterministic process and depends only on σ_w, σ_b and h_t. The pre-activation $h_t^{(i)}$ is indeed sampled from $\mathcal{N}(0, q_t)$. However, the equation deals with the expectation over this distribution (average behavior over the distribution of $h_t^{(i)}$ aka Mean Field Approximation)

The statistics of the distribution follows a deterministic dynamic as propagating through layers. This statistic can be treated as the information signal.

5.2 Stability Analysis and Implications

Normally in a dynamical system; stability analysis is concerned with determining whether small perturbations to the system's state will grow, decay or remain constant over time. Stability is typically analyzed around equilibrium points (also called fixed-point or steady-state); which ar points where the system does not change unless perturbed.

Steps in traditional stability analysis:

1. Identify equilibrium points of a system e.g. $\dot{x} = f(x)$ and $f(x^*) = 0$
2. Linearize the system about the equilibrium point. This is done by taking the Jacobian of the system equations at the equilibrium point e.g. $J(x^*) = \frac{\partial f(x)}{\partial x}|_{x=x^*}$. This matrix describes how small perturbations evolve near x^*.
3. Analyze the eigenvalues of $J(x^*)$.
   - $\lambda < 0$: asymptotically stable (all eigenvalues: $\text{Re}(\lambda) < 0$)
   - $\lambda > 0$: unstable (if any eigenvalue has positive real part)
   - $\lambda = 0$: marginally stable (if all eigenvalues have $\text{Re}(\lambda) < 0$ and at least one λ has 0 real part but no positive real part). In this case stability depends on higher-order terms or non-linearity and the system might stay near the equilibrium but not necessarily return to it.

Example: Consider a 2D-system:

$$\dot{x} = f_1(x,y)$$
$$\dot{y} = f_2(x,y)$$

Find x^*, y^* by solving $\dot{x} = 0$ and $\dot{y} = 0$. Then linearize the system at the equilibrium point:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}_{|(x^*,y^*)}$$

- If $\text{Re}(\lambda_1) < 0$ and $\text{Re}(\lambda_2) < 0$: equilibrium is stable.
- If $\text{Re}(\lambda_1) > 0$ or $\text{Re}(\lambda_2) > 0$: equilibrium is unstable.

In the case when the system is highly non-linear, local linearization may not provide enough insight into stability. The Lyapunov method can be used by finding a function $V(x)$ behaving like an energy function. If this function decreases or remains constant along the system trajectories, then the system is stable.

Consider a system $\dot{x} = f(x)$ where $x \in \mathbb{R}^n$ and x^* s.t. $f(x^*) = 0$ i.e. $\dot{x} = 0$. Lyapunov method consists of finding a scalar function $V : \mathbb{R}^n \to \mathbb{R}$ such that:
- $V(x) > 0 \quad \forall x \neq x^*$
- $V(x^*) = 0 \quad \forall x = x^*$
- $\dot{V}(x) < 0 \quad \forall x \neq x^*$

Then;

$$\dot{V}(x) = \frac{\partial V(x)}{\partial t} = \nabla V(x) \cdot \dot{x} = \nabla V(x) \cdot f(x)$$

- If $\dot{V}(x) < 0$ for $\forall x \neq x^*$: asymptotically stable.
- If $\dot{V}(x) \leq 0$ for $\forall x \neq x^*$: Lyapunov stable.
- If $\dot{V}(x) > 0$ for $\forall x \neq x^*$: unstable.

For example, consider a dynamical system:

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -x_1 \\ -x_2^3 \end{bmatrix}$$

A common choice for $V(x)$ is a quadratic function:

$$V(x) = \frac{1}{2} x^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x = \frac{1}{2}(x_1^2 + x_2^2)$$

Clearly, $V(x) > 0$ for $x \neq x^*$ where $x^* = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and $V(x^*) = 0$.

Now,

$$\dot{V}(x) = \frac{\partial V}{\partial x_1}\dot{x}_1 + \frac{\partial V}{\partial x_2}\dot{x}_2 = -x_1^2 - x_2^4$$

Thus,

$$\dot{V}(x) = -(x_1^2 + x_2^4) < 0 \quad \forall x \neq x^*$$

indicating that the system is **asymptotically stable**.

Within the perspective of stability analysis via Jacobian matrix, define a residual system, $\epsilon_t := \Sigma_t - \Sigma^*$ and first order expand it at Σ^*. *Refer to 'Deep Information Propagation - ICLR 2017'.* The Jacobian of this residual system can be decoupled into two sub-systems because of the independent evolution of two signal quantities in q_{t+1}. Their eigenvalues are given by;

$$\mathcal{X}_{q*} = \sigma_w^2 \mathbb{E}_{h \sim \mathcal{N}(0,\Sigma*)} \left[\phi''(h^{(i)})\phi(h^{(i)}) + \phi'(h^{(i)})^2 \right]$$

$$\mathcal{X}_{c*} = \sigma_w^2 \mathbb{E}_{h\sim\mathcal{N}(0,\Sigma*)}\left[\phi'(h^{(i)})\phi'(h^{(j)})\right] \qquad h^{(i)} \neq h^{(j)}$$

For a fixed point to be stable: $\mathcal{X}_{q*} < 1$ and $\mathcal{X}_{c*} < 1$. The log of $\mathcal{X}_{q*}$ and $\mathcal{X}_{c*}$ relate to a well-known Lyapunov exponents in the dynamical system theory i.e.

$$|q_t - q^*| \sim e^{-t/\xi_{q*}} \qquad ; \qquad \xi_{q*}^{-1} = \log \mathcal{X}_{q*}$$
$$|c_t - c^*| \sim e^{-t/\xi_{c*}} \qquad ; \qquad \xi_{c*}^{-1} = \log \mathcal{X}_{c*}$$

where, c_t represents the dynamics of correlation. Recall that for the given DNN, its information propagation rule depends only on σ_w and σ_b. We can now construct a phase diagram as shown below.

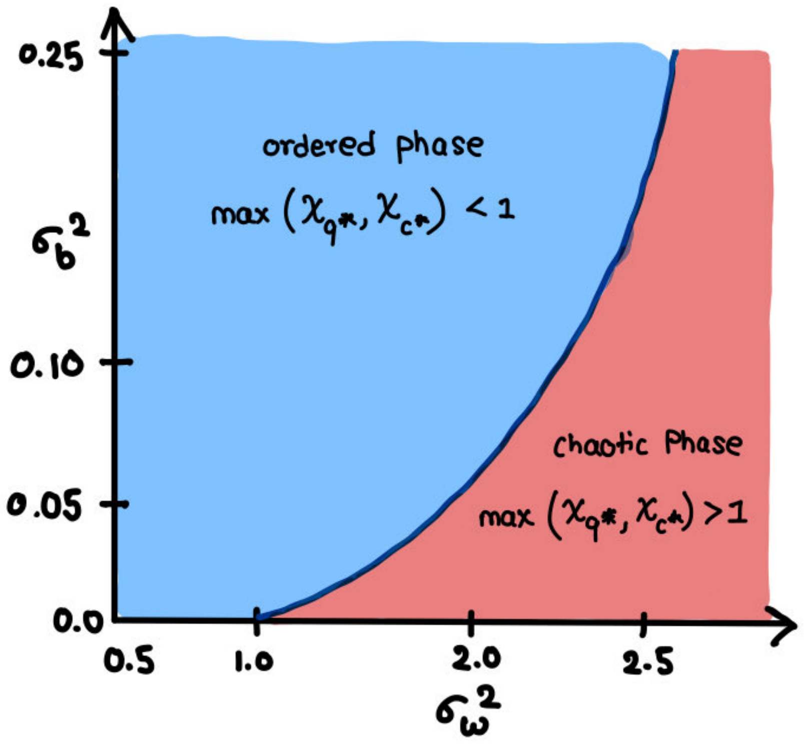

0.25
σ_b^2
0.10
0.05
0.0
0.5
1.0
2.0
2.5
σ_w^2
ordered phase
max $(\mathcal{X}_{q*}, \mathcal{X}_{c*}) < 1$
chaotic Phase
max $(\mathcal{X}_{q*}, \mathcal{X}_{c*}) > 1$

Figure 5: Phase Diagram

Networks initialized in the ordered phase may suffer from saturated information if the depth is sufficiently large i.e. they become untrainable. However, if the network is initialized along the critical line remain marginal stable and the information is able to propagate through arbitrary depth without saturating or exploding.

NOTE: Once the training begins, the iid assumption needed to construct q_t no longer holds and consequently the analysis (stability and eigen-decomposition) also is invalid. However, experimental results suggest that for an over-parameterized DNN each weights will be close to random initialization over training iterations. This means that under certain assumptions the statistical properties derived at initialization can be preserved throughout training process. This has a strong implication and can be used to prove the global convergence and optimality of Gradient Descent.

5.2.1 Deep Information and Gradient Descent

The overarching goal in this section is to prove how deep information is brought to gradient descent. Recall that we have two input pairs $x^{(\alpha)}$ and $x^{(\beta)}$. The variance at the activation is given by

$$q_{t+1}^{(\alpha,\beta)} = \sigma_w^2 \, \mathbb{E}_{[h_t^{(\alpha)}, h_t^{(\beta)}]^T \sim \mathcal{N}(0,\Sigma_t)}[\phi(h_t^{(\alpha)})\phi(h_t^{(\beta)})] + \sigma_b^2$$

with initial condition $q_0 = \frac{1}{N_0} x_0^{(\alpha)} x_0^{(\beta)}$ and $\Sigma_t = \begin{bmatrix} q_t^{(\alpha)} & q_t^{(\alpha,\beta)} \\ q_t^{(\alpha,\beta)} & q_t^{\beta} \end{bmatrix}$

Now construct a Gram matrix $K_t \in \mathbb{R}^{|D|\times|D|}$, where $|D|$ is the size of the dataset and t represents the time, such that $t \in \{0, 1, \dots, T-1\}$ and

$$K_t^{(i,j)} := q_t^{i,j}$$

represents the information quantity between the data points with corresponding indices.

Then consider an MSE objective, i.e., $\frac{1}{2}\|y-u\|^2$, where $y, u \in \mathbb{R}^{|D|}$ and each element $u^{(i)}$ denotes the scalar prediction of each data point $i \in D$, i.e., $u^{(i)} = \mathbf{1}^T x_T^{(i)}/||\mathbf{1}||_2$. The dynamics of the prediction error governed by the GD algorithm takes an analytical form:

$$y - u(k+1) \approx [I - \eta G_n(k)]\,(y - u(k)),$$

where

$$G_n^{(i,j)}(k) := \left\langle \frac{\partial u_i(k)}{\partial \theta_{T-1}^{(k)}}, \frac{\partial u_j(k)}{\partial \theta_{T-1}^{(k)}} \right\rangle,$$

with k-th iteration process and η-learning rate.

Clearly, the equation above shows the linear dynamics of error between two successive iterations. The matrix G at initialization is related to the Gram matrix K by:

$$G_n^{(i,j)}(k) = \frac{1}{\sigma_w^2} K_{T-1}^{(i,j)} \mathbb{E}_{h\sim\mathcal{N}(0,\Sigma_{T-1})}\left[\phi'(h^{(i)})\phi(h^{(j)})\right] := K_T^{(i,j)}.$$

When the width is large, $G(k)$ will be close to K_T ($\|G(k) - K_T\|_2$ bounded) for all iterations $k \geq 0$. This, together with the least eigenvalue of K_T being lower-bounded for non-degenerate datasets, ensures linear convergence to the global minimum.

5.3 Training DNN with Optimal Control

The parameters in DNN can be treated as control variables, and thereby we can draw an interesting connection between deep learning and optimal control problems. In a discrete case, the optimal control problem is:

$$\min_{\{\theta_t\}_{t=0}^{T-1}} J = \mathbb{E}\left[\sum_{t=0}^{T-1} L(x_t, \theta_t) + \Phi(x_T)\right] \quad \text{s.t.} \quad x_{t+1} = f(x_t, \theta_t) \tag{i}$$

Goal of supervised learning is to find a set of optimal parameters at each time-step, i.e., layer, $\{\theta_t\}_{t=0}^{T-1}$ such that when starting from the initial state x_0, its terminal state x_T is close to the target y.

- **Dynamical Constraints** → Characterized by the DNNs
- **Terminal Cost** → Training Cost
- **Control Dependent Intermediate Cost** → Weight Regularization

Now, let's view a batch of data (x_0, y), input-output pairs as a random variable drawn from a space of probability measure. In this paradigm, mean-field formalism can be utilized where the analysis is lifted to distribution spaces. Now, the goal is to find an optimal transport that propagates the input population to the desired target distribution.
The population-risk minimization problem can be regarded as a mean-field optimal control problem.

$$\inf_{\theta_t \in L^\infty} \mathbb{E}_{(x_0,y)\sim\mu_0} \left[\Phi(x_T, y) + \int_0^T L(x_t, \theta_t)\, dt \right] \quad \text{s.t.} \quad \dot{x}_t = f(x_t, \theta_t) \tag{ii}$$

Where $L^\infty = L^\infty([0,T], \mathbb{R}^m)$ denotes the set of essentially bounded measurable functions, and μ_0 is the joint distribution of the initial state x_0 and target y.

Remark: Changed formalism to continuous case because it is mathematically convenient.

This mean-field optimal control formalism allows optimization of DNN training procedure through two perspectives i.e. Pontryagin's Principle and HJB equation.

5.3.1 Mean-Field Pontryagin's Minimum Principle

Pontryagin's minimization principle allows for the necessary conditions of (i). It provides conditions that an optimal state-control trajectory must obey locally. Assume,

1. f is bounded
2. f/L are continuous in θ_t
3. f, L, Φ are continuously differentiable w.r.t x_t
4. μ_0 has bounded support i.e. $\mu_0(\{(x,y) \in \mathbb{R}^n \times \mathbb{R}^d : ||x|| + ||y|| \leq M\}) = 1$ for some $M > 0$

Let $\theta_t^* : t \to \mathbb{R}^m$ be a solution to (ii) that achieves infimum. Then, there exists a continuous stochastic process x_t^* and p_t^* such that $\forall \theta_t \in \mathbb{R}^m$, a.e $t \in [0,T]$

$$\dot{x}_t^* = \nabla_p H(x_t^*, p_t^*, \theta_t^*) \quad ; \quad x_0^* = x_0 \tag{iii}$$

$$\dot{p}_t^* = -\nabla_x H(x_t^*, p_t^*, \theta_t^*) \quad ; \quad p_T^* = \nabla_x \Phi(x_T^*, y) \tag{iv}$$

$$\mathbb{E}_{\mu_0}\left[H(x_t^*, p_t^*, \theta_t^*)\right] \leq \mathbb{E}_{\mu_0}\left[H(x_t^*, p_t^*, \theta_t)\right] \tag{v}$$

where the Hamiltonian function $H : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is given by:

$$H(x_t, p_t, \theta_t) = p_t \cdot f(x_t, \theta_t) + L(x_t, \theta_t) \tag{vi}$$

p_t is the co-state (like Lagrange multiplier) of the adjoint equation. Note that the minimization of Hamiltonian in (v) is now taken over the expectation w.r.t μ_0.

The conditions characterized by (iii) - (v) can be linked to the optimization dynamics in DNN training:

- (iii) - represents feed-forward pass from first layer to last layer
- (iv) - can be viewed as back-propagation

To see this consider the Hamiltonian:

$$H(x_t, p_{t+1}, \theta_t) = p_{t+1} \cdot f(x_t, \theta_t) + L(x_t, \theta_t) \tag{vii}$$

and

$$p_t^* = \nabla_x H(x_t^*, p_{t+1}^*, \theta_t^*) = p_{t+1}^* \nabla_x f(x_t^*, \theta_t^*) + \nabla_x L(x_t^*, \theta_t^*) \tag{viii}$$

where p_t^* is the gradient of the total loss function w.r.t the activation at layer t.

Maximization in (v) can be difficult to solve because the number of parameters can be in millions. So we apply approximated updates iteratively using first-order derivatives i.e.

$$\theta_t^{(i+1)} = \theta_t^{(i)} - \eta \nabla_{\theta_t} \mathbb{E}_{\mu_0} \left[H(x_t^*, p_{t+1}^*, \theta_t^{(i)}) \right] \tag{ix}$$

Superscript (i) denotes the iterative update of the parameters in the outer loop. Update rule (ix) is equivalent to performing gradient descent on the original objective function J in (i) i.e.

$$\nabla_{\theta_t} H(x_t^*, p_{t+1}^*, \theta_t^*) = p_{t+1}^* \cdot \nabla_{\theta_t} f(x_t^*, \theta_t) + \nabla_{\theta_t} L(x_t^*, \theta_t)$$

or:

$$\nabla_{\theta_t} H(x_t^*, p_{t+1}^*, \theta_t) = \nabla_{\theta_t} J \tag{x}$$

It has been shown that when $\mathbb{E}_{\mu_0}[\,\cdot\,]$ in (ix) is replaced with a Sample mean, and if the solution (ii) is stable, then we can find with a high probability a random variable in its neighborhood that is stationary solution of the sampled Pontryagin minimum Principle.

5.3.2 Mean-Field Hamilton-Jacobi-Bellman Equation

HJB analysis has been extended to mean-field by considering probability measures as states.

Consider class of probability measures that are square integrable on Euclidean space with Wasserstein-W_2 metric denoted by $P_2(\mathbb{R})$. Define:

$$J(t, \mu, \theta_t) := \mathbb{E}_{(x_t, y_t) \sim \mu_t} \left[\Phi(x_T, y) + \int_t^T L(x_\tau, \theta_\tau)\, d\tau \right] \quad \text{Cost-to-go}$$

$$v^*(t, \mu) := \inf_{\theta_t \in L^\infty} J(t, \mu, \theta_t) \quad \text{Value function}$$

Note that the expectation is taken over the distribution evolution, starting from μ and propagating through the DNN architecture.

Consider f, L, Φ are bounded and are Lipschitz continuous w.r.t x_t and the Lipschitz constant of f and L are independent of θ_t, $\mu_0 \in P_2(\mathbb{R}^{n+d})$. Then both $J(t, \mu, \theta_t)$ and $v^*(t, \mu)$ are Lipschitz continuous on $[0, T] \times P_2(\mathbb{R}^{n+d})$. $\forall\, 0 \le t \le t' \le T$, the principle of dynamic programming suggests that:

$$v^*(t, \mu) = \inf_{\theta_t \in L^\infty} \mathbb{E}_{(x_t, y_t) \sim \mu_t} \left[\int_t^{t'} L(x_\tau, \theta_\tau) d\tau + v^*(\hat{t}, \hat{\mu}) \right]$$

where $\hat{\mu}$ denotes the terminal distribution at $\hat{t}$. Now, Taylor expand the above value function:
Assume: $\hat{t} = t + \Delta t$ and Δt is very small. Then, $\int_t^{\hat{t}} L(x_\tau, \theta_\tau) d\tau = L(x_t, \theta_t)\Delta t$ and

$$v^*(\hat{t}, \hat{\mu}) \approx v^*(t, \mu) + \frac{\partial v^*(t, \mu)}{\partial t} \Delta t + \left\langle \frac{\delta v^*(t, \mu)}{\delta \mu}, (\hat{\mu} - \mu) \right\rangle$$

Here, $(\hat{\mu} - \mu)$ can be approximated as: $\hat{\mu} - \mu \approx f(x_t, \theta_t)\Delta t$.

$$v^*(t, \mu) = \inf_{\theta_t \in L^\infty} \mathbb{E}_{(x_t, y_t) \sim \mu_t} \left[L(x_t, \theta_t)\Delta t + v^*(t, \mu) + \frac{\partial v^*(t, \mu)}{\partial t} \Delta t + \frac{\delta v^*(t, \mu)}{\delta \mu} f(x_t, \theta_t) \Delta t \right]$$

or,

$$0 = \inf_{\theta_t \in L^\infty} \mathbb{E}_{(x_t, y_t) \sim \mu_t} \left[L(x_t, \theta_t)\Delta t + \frac{\partial v^*(t, \mu)}{\partial t} \Delta t + \frac{\delta v^*(t, \mu)}{\delta \mu} f(x_t, \theta_t) \Delta t \right]$$

$$\therefore \frac{\partial v(t, \mu)}{\partial t} + \inf_{\theta_t \in L^\infty} \left\langle \frac{\delta v(\mu)(\cdot)}{\delta \mu}, f(\cdot, \theta_t) \right\rangle_\mu + \langle L(\cdot, \theta_t) \rangle_\mu = 0$$

$$\therefore v(T,\mu) = \langle \Phi(\cdot) \rangle_\mu$$

Finally if $\theta^* : (t,\mu) \to \mathbb{R}^m$ is a feedback policy that achieves the infimum in the above equation, then θ^* is an optimal solution of the problem.

REMARK: Due to the curse of dimensionality classical HJB equations can be computationally intractable to solve for high dimensional problem let alone its mean-field approximation. DNN optmization algorithms with flavor from DP (or its approximation) have not been well-explored.

5.4 Stochastic Optimization as a Dynamical System

Stochastic optimization for DNNs viewed as a dynamical system offers insights into the convergence behavior, stability, generalization, and robustness of neural networks. This perspective leverages tools from dynamical systems theory, stochastic calculus, and statistical physics to understand the complex, non-linear, and high-dimensional optimization landscape of deep learning. It highlights that DNN training is not just a minimization task but a process that navigates a stochastic, dynamic system, where noise and stability play critical roles.

5.4.1 Introduction to Stochastic Mini-Batch Gradient Descent

With a slight abuse of notation, the training loss on the dataset $\mathcal{D}$ is:

$$\Phi(\theta, \mathcal{D}) = \Phi = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} J(f(x_i,\theta), y_i) \quad \text{where } J(\cdot,\cdot) \text{ is the training objective}$$

and $f = f_0, f_1, f_2, \ldots$ includes all compositional functions of the DNN.
The gradient of the loss function for each sample is:

$$\nabla\Phi(\theta) = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} g^i(\theta) \quad ; \quad g^i(\theta) \text{ is the gradient on each sample } (x^{(i)}, y^{(i)}).$$

The covariance matrix of $g^i(\theta)$, denoted $\Sigma_{\mathcal{D}}(\theta)$, is a positive-definite matrix which can be computed deterministically given the architecture of DNN, dataset, and parameter:

$$\text{Var}\left(g^i(\theta)\right) := \frac{1}{|\mathcal{D}|} \left[g^i(\theta) - g(\theta)\right] \left[g^i(\theta) - g(\theta)\right]^T \equiv \Sigma_{\mathcal{D}}(\theta)$$

Since it is expensive to access $g(\theta)$ at each iteration, we can only estimate it using mini-batch i.i.d. samples $\mathcal{B} \subset \mathcal{D}$; as $|\mathcal{B}| \gg 1$. C.L.T. implies that the mini-batch gradient

$$g^{\text{mb}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} g^i(\theta) \quad \text{has sample mean and variance as;}$$

$$\text{mean } \left[g^{\text{mb}}(\theta)\right] := \mathbb{E}_{\mathcal{B}}\left[g^{\text{mb}}(\theta)\right] \approx g(\theta)$$

$$\text{var } \left[g^{\text{mb}}(\theta)\right] := \mathbb{E}_{\mathcal{B}}\left[\left(g^{\text{mb}}(\theta) - g(\theta)\right)\left(g^{\text{mb}}(\theta) - g(\theta)\right)^T\right] \approx \frac{1}{|\mathcal{B}|}\Sigma_{\mathcal{D}}(\theta)$$

Now define a two-point noise matrix as:

$$\tilde{\Sigma}_B := \mathbb{E}_{\mathcal{B}}\left[g^{\text{mb}}(\theta) g^{\text{mb}}(\theta)^T\right]$$

The entries of this noise-matrix $\tilde{\Sigma}_B$ are:

$$\tilde{\Sigma}_B(i,j) := \mathbb{E}_{\mathcal{B}}\left[g^{\text{mb}}(\theta_i) g^{\text{mb}}(\theta_j)\right] \quad \text{and}$$

$$\tilde{\Sigma}_{B(i_1,i_2,\ldots,i_k)} := \mathbb{E}_{\mathcal{B}}\left[g^{\text{mb}}(\theta_{i_1})g^{\text{mb}}(\theta_{i_2})\ldots g^{\text{mb}}(\theta_{i_k})\right]$$

Here; $g^{mb}(\theta_i)$ is partial derivative w.r.t. θ on the mini-batch. So we can re-write

$$\text{Var}[g^{mb}(\theta)] = \tilde{\Sigma}_B - g(\theta)g(\theta)^T$$

Also denote the distribution of parameter at training cycle t as:

$$\rho_t(z) := \rho(z,t) \propto \mathbb{P}(\theta_t = z)$$

and the steady-state distribution as $\rho^{ss} := \lim_{t\to\infty} \rho_t$.

5.4.2 Continuous-time Dynamics of SGD

Derivation:

We can write $g^{\text{mb}}(\theta) \sim \mathcal{N}\left(g(\theta), \frac{1}{|\mathcal{B}|}\Sigma_{\mathcal{D}}(\theta)\right)$. Then the update rule of SGD at each iteration can be written as:

$$\theta_{t+1} = \theta_t - \eta g^{\text{mb}}(\theta_t)$$

i.e.

$$\theta_{t+1} \approx \theta_t - \eta\left(g(\theta_t) + \frac{1}{\sqrt{|\mathcal{B}|}}\Sigma_{\mathcal{D}}^{1/2}(\theta_t)Z_t\right)$$

where η is the learning rate and $Z_t \sim \mathcal{N}(0, I)$.
Now consider the following SDE and its Euler discretization:

$$d\theta_t = b(\theta_t)dt + \sigma(\theta_t)dW_t$$

The Euler-Maruyama discretization over a small time step Δt is given by:

$$\theta_{t+\Delta t} \approx \theta_t + b(\theta_t)\Delta t + \sigma(\theta_t)\Delta W_t \quad \text{with } \Delta W_t \sim \mathcal{N}(0, \Delta t)$$

i.e.

$$\theta_{t+1} \approx \theta_t + b(\theta_t)\Delta t + \sigma(\theta_t)\sqrt{\Delta t}\cdot Z_t$$

Note that SGD update and Euler-discretization are equivalent if we set $\Delta t \approx \eta$;

$$b(\theta_t) \approx -\eta g(\theta_t) \quad \text{and} \quad \sigma(\theta_t) \approx \sqrt{\frac{\eta}{|\mathcal{B}|}}\Sigma_{\mathcal{D}}^{1/2}(\theta_t)$$

The continuous time limit of SGD is:

$$d\theta_t = -\eta g(\theta_t) + \sqrt{\frac{\eta}{|\mathcal{B}|}}\Sigma_{\mathcal{D}}^{1/2}(\theta_t)dW_t$$

Let $\beta = \frac{2|\mathcal{B}|}{\eta}$; then:

$$d\theta_t = -\eta g(\theta_t) + \sqrt{2\beta^{-1}\Sigma_{\mathcal{D}}(\theta_t)}dW_t$$

Note: As we approach flat locations in the loss landscape, the drift term approaches zero, and the gradient evolution is dominated by the diffusion term.

Dynamics of Training Loss:

We need Itô's lemma in order to describe the propagation of training loss $\Phi(\theta_t)$ as a function of the stochastic process:

$$d\theta_t = -\eta g(\theta_t) + \sqrt{2\beta^{-1}\Sigma_{\mathcal{D}}(\theta_t)}dW_t$$

Itô's Lemma:

Imagine you have a stochastic process and there is a function that depends on this stochastic process. Itô's lemma provides a way to describe the evolution of that function (both in terms of X_t and t).

Consider a stochastic process $dX_t = b(X_t,t)dt + \sigma(X_t,t)dW_t$. Suppose $b(\cdot,\cdot)$ and $\sigma(\cdot,\cdot)$ follow smooth and appropriate growth conditions then for a given function $V(\cdot,\cdot)$ that is twice continuously differentiable in X_t and t respectively i.e. $V(\cdot,\cdot) \in C^{2,1}(\mathbb{R}^d \times [0,T])$; $V(X_t,t)$ is also a stochastic process.

The evolution of $V(X_t,t)$ is then given by;

$$\begin{aligned} dV(X_t,t) = &\left[\partial_t V(X_t,t) + \nabla_x V(X_t,t)^T b(X_t,t) + \frac{1}{2} Tr\ \sigma(X_t,t)^T \nabla_x^2 V(X_t,t)\sigma(X_t,t)\right] dt \\ &+ \left[\nabla_x V(X_t,t)^T \sigma(X_t,t)\right] dW_t \end{aligned}$$

If $V(\Phi(\theta_t))$ i.e. training loss. Then we apply Itô's lemma, which yields:

$$\begin{aligned} d\Phi(\theta_t) &= \left[\partial_t \Phi(\theta_t,t) + \nabla\Phi(\theta_t,t)^T(-g(\theta_t)) + \frac{1}{2} Tr\ \left(\tilde{\Sigma}_d^{1/2} H_\Phi \tilde{\Sigma}_d^{1/2}\right)\right] dt + \left[\nabla\Phi(\theta_t,t)^T \tilde{\Sigma}_D^{1//2}\right] dW_t \\ d\Phi(\theta_t) &= \left[-\nabla\Phi(\theta_t)^T g(\theta_t) + \frac{1}{2} Tr\ \left(\tilde{\Sigma}_d^{1/2} H_\Phi \tilde{\Sigma}_d^{1/2}\right)\right] dt + \left[\nabla\Phi(\theta_t)^T \tilde{\Sigma}_D^{1/2}\right] dW_t \end{aligned}$$

where $\tilde{\Sigma}_D^{1/2} = \sqrt{2\beta^{-1}\Sigma_D(\theta_t)}$. Recall $\nabla\Phi = g$. Now, take the expectation over the parameter distribution $\rho_t(\theta)$ to simplify the expression above. This gives the dynamics of expected training loss:

$$d\mathbb{E}_{\rho_t}\left[\Phi(\theta_t)\right] = \mathbb{E}_{\rho_t}\left[-\nabla\Phi^T\nabla\Phi + \frac{1}{2} Tr\ \left(H_\Phi \tilde{\Sigma}_d\right)\right] dt$$

When training converges, the expected magnitude of the gradient signal is balanced off by the expected Hessian-covariance product measured in trace norm i.e.

$$\mathbb{E}_{\rho^{ss}}[\nabla\Phi^T\nabla\Phi] \approx \mathbb{E}_{\rho^{ss}}[Tr\ (H_\Phi \tilde{\Sigma}_D)]$$

5.5 Control Inspired Learning Algorithms

Recall that the mean-field Pontryagin's maximum principle is a general necessary condition for optimality in mean-field formulation of deep learning. Hence, it should give rise to training algorithms just like how the condition $\nabla\Phi(\theta^*) = 0$ gives rise to gradient descent. The dynamical structure of the problem also hints at adapting other methodologies in numerical analysis of differential equations to deep learning. In this section, we survey some work in this direction. For simplicity, lets only consider empirical risk (and often we will only consider 1-sample case for the ease of writing). In this case, the mean-field PMP reduces to its classical version.

5.5.1 Method of Successive Approximation

The so-called *method of successive approximations* (MSA)Chernousko and Lyubushin [1982] or the *sweeping method*, which perhaps is the simplest method for finding a solution of the PMP equations. The PMP equations are three coupled equations in three unknowns x^*, p^*, θ^*. Moreover, given any two of them, computing the third is straightforward, at least conceptually. Assuming there are N samples, according to the mean-field formulation (with μ the empirical measure), we would then solve the MSA as follows:

We start with an initial guess θ^0 for the optimal control and at n^{th} iteration we solve;

$$\dot{x}_i^n(t) = f(x_i^n(t), \theta^n(t)) \qquad x_i^n(0) = x_i$$

$$\dot{p}_i^n(t) = -\nabla_x H(x_i^n(t), p_i^n(t), \theta^n(t)) \qquad p_i^n(T) = -\nabla_x \Phi(x_i^n(T), y_i)$$

$$\theta^{n+1}(t) = \arg\max_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} H(x_i^n(t), p_i^n(t), \theta)$$

$\forall i = 1, \cdots, N$. This method includes as a special case the classical back-propagation algorithm for deep learning by discretization argument.

MSA and Back-Propagation

Consider, for simplicity, the running-cost $L \equiv 0$ and sample size $N = 1$. Let x be the state under some control θ. Using forward-Euler discretization of the dynamics, we can write:

$$\hat{x}(k+1) = \hat{x}(k) + \Delta t\ f(\hat{x}(k), \hat{\theta}(k)), \qquad k = 0, \cdots, K-1 \ \ (K\Delta t = T)$$

Now fix $k < K$ and regard $\hat{x}(k)$ as a function of $\hat{x}(K)$, we have by the chain rule (D denotes total derivative)

$$\begin{aligned} D_{\hat{x}(k)}\Phi(\hat{x}(K)) &= [D_{\hat{x}(k)}\hat{x}(k+1)]^T D_{\hat{x}(k+1)}\Phi(\hat{x}(K)) \\ &= [I + \Delta t \nabla_{\mathbf{x}} f(\hat{x}(k), \hat{\theta}(k))]^T D_{\hat{x}(k+1)}\Phi(\hat{x}(K)) \end{aligned}$$

Letting $\hat{p}(k) := -D_{\hat{x}(k)}\Phi(\hat{x}(K))$, we see that

$$\hat{p}_{k+1} = \hat{p}_k - \Delta t[\nabla_x f(\hat{x}(k), \hat{\theta}(k))]^T \hat{p}_{k+1}, \quad k = 0, \ldots, K-2, \quad \hat{p}_K = -\nabla_x \Phi(\hat{x}(K))$$

which one recognizes as the (backward) Euler discretization of the co-state equation. In other words, the co-state here tells us the sensitivity of the loss function with respect to the state.

A further application of the chain rule shows that

$$-D_{\hat{\theta}(k)}\Phi(\hat{x}(K)) = \nabla_\theta H(\hat{x}(k), \hat{p}(k+1), \hat{\theta}(k)).$$

Thus, we have shown that applying gradient descent on the loss function Φ is equivalent to performing gradient ascent on H, i.e., we are approximately maximizing H when we perform gradient descent.

Thus, back-propagation with gradient descent is equivalent to the MSA if we replace the step (3.13) by

$$\theta^{n+1}(t) = \theta^n(t) + \eta \nabla_\theta H(\mathbf{x}^n(t), p^n(t), \theta).$$

In other words, we can view MSA as a generalization of the back-propagation algorithm. This leads to the question of whether we can derive other algorithms from the MSA directly, without replacing this step. It further suggests that it is possible to use MSA to derive the corresponding algorithm for the PMP generally applicable to arbitrary terminal targets Φ. Other applications of the MSA/adjoint method include adversarial training and generative models.

5.5.2 Layer Parallel Training Algorithms

There is another aspect of PMP that leads to other algorithmic innovations, namely layer-parallel training. This refers to learning algorithms that can use many processors to train a deep neural network in parallel, not in terms of data splitting but layer splitting.

Let's take a look at PMP equations which consists of two components: (a) Solving dynamical equations for x^* and p^* and (b) Solving the Hamiltonian maximization/minimization problem for θ^*. A crucial observation is that given component (a), component (b) can be computed in parallel for all t, i.e this step is naturally layer-parallel and each-layer need to talk to one another. Thus, to achieve full layer parallelism, it is enough to make the solution of x^* and p^* parallel. There are two main ways of doing so, each relying on different aspects of the Hamilton's equations.

Exploiting Two-Point BVP Form:

Notice that given θ, the equations

$$\begin{aligned} \dot{x}(t) &= f(x(t), \theta(t)), \quad x(0) = x \\ \dot{p}(t) &= -\nabla_x H(x(t), p(t), \theta(t)), \quad p(T) = -\nabla_x \Phi(x(T), y) \end{aligned}$$

constitute a two-point boundary value problem (2P-BVP), in that the equation for x has an initial condition whereas the equation for p has a terminal condition. Hence, one can think of breaking this down into two sub-problems: let $S = T/2$ and consider

P_1 **(for** $t \in [0, S]$**)**

$$\begin{aligned} \dot{p}^n(t) &= -\nabla_x H(x^{n-1}(t), p^n(t), \theta^n(t)), \quad p^n(S) = p^{n-1}(S) \\ \dot{x}^n(t) &= f(x^n(t), \theta^n(t)), \quad x^n(0) = x \end{aligned}$$

P_2 **(for** $t \in [S, T]$**)**

$$\begin{aligned} \dot{x}^n(t) &= f(x^n(t), \theta^n(t)), \quad x^n(S) = x^{n-1}(S) \\ \dot{p}^n(t) &= -\nabla_x H(x^n(t), p^n(t), \theta^n(t)), \quad p^n(T) = -\nabla_x \Phi(x^n(T), y) \end{aligned}$$

The two problems P_1 and P_2 can now be run in parallel on two processors, and at the end of each run the boundary values are passed between them in a communication round. The main idea of the algorithm proposed in Parpas and Muir [2019], but there are additional correction steps not discussed here in detail. We remark here that the method here requires synchronization between P_1 and P_2, and extension to more than two processors requires additional work.

Exploiting Continuity in time:

An alternative approach is the multi-grid approach directly on the state and the co-state ODEs. This is based on the multi-grid in time idea in numerical analysis Falgout et al. [2014], which is an adaptation of the classical spatial(non-linear) multi-grid method for solving boundary-value problems.

The basic observation in this case is that for a discretized (say forward Euler) ODE (we absorbed $\hat{\theta}(k)$ and $k\Delta t$ into the definition of f)

$$\hat{x}(k+1) = \hat{x}(k) + \Delta t \ f(k, \hat{x}(k)), \qquad k = 0, \cdots, K-1, \qquad \hat{x}(0) = x_0$$

we can define a vector X and a vector-valued function A such that;

$$X := \begin{bmatrix} \hat{x}(0) \\ \hat{x}(1) \\ \vdots \\ \hat{x}(K) \end{bmatrix} \qquad A(X) := \begin{bmatrix} X_0 - x_0 \\ X_1 - f(0, X_0) \\ \vdots \\ X_L - f(K-1, X_{K-1}) \end{bmatrix}$$

Now, we can write the discretized Euler ODE as;

$$A(X) = 0$$

The feed-forward method represented by the discretized Euler ODE solves $A(X) = 0$ by sequential substitution, since $A(X)$ has a 'lower triangular' form, in the sense that k^{th} row of its output does not depend on $X_{K+1}, \cdots, X_K$.

One can also attempt to solve $A(X) = 0$ iteratively, say by minimizing $\min_X ||A(X)||^2$. In this case, one can start withh initial guess X^0 and refine this guess and the crucial point is that the relaxation is now no longer sequential in time and can be parallelized. The multi-grid method build on this idea, where instead of solving this minimizing problem in one go, it solves by iterating a sequence of grids of varying sizes, and transferring the information of the solution across the grids.

For this to be sensible, one should expect some regularity in time in the solution X, so that restriction/interpolation operations between grids of different resolution give meaningful initializations for their respective temporal scales. This is ensured by the ODE structure of the dynamics that naturally builds temporal regularity in the states and the co-states. Recent work based on this approach include Chang et al. [2017], Gunther et al. [2020].

6 Examples: Runge-Kutta (ResNet) and ODENets

There are a number of works Weinan [2017]; Haber and Ruthotto [2017]; Chang et al. [2017]; Weinan et al. [2018]; Li and Hao [2018] that aim to design deep neural network architectures. Particularly along this line of work, the authors propose an interpretation of deep learning by the the popular Residual Network (ResNet) architecture He et al. [2016] as discrete optimal control problems.

Let

$$u^{(j)} := \left(K^{(j)}, \beta^{(j)}\right) \quad \forall j \in [0, N-1]$$

and $u = \left(u^0, \cdots, u^{N-1}\right)$. Here, $K^{(j)}$ is a $n \times n$ matrix of weights, $\beta^{(j)}$ represents biases and N is the number of layers in the network.

For simplicity, the problem in this section is a binary classification problem. The constraint minimization problem is of the form:

$$\min_{y,u,W,\mu} \sum_{i=1}^{m} \left\| \mathcal{C}\left(Wy_i^{(N)} + \mu\right) - c_i \right\|^2 + \mathcal{R}(u) \qquad s.t. \qquad y_i^{(j+1)} = y_i^{(j)} \Delta t f(y_i^{(j)}, u^{(j)})$$

with $y_i^{(0)} = x_i$ and some form of regularization $\mathcal{R}(\cdot)$ on the control/weights of the network. Here, Δt is a parameter which for simplicity can be chosen to be equal to 1 and whose role will become clear in what follows. The constraint in the above optimization problem is the ResNet parametrization of a Neural Network. In contrast, the widely used feed-forward network is given by;

$$y_i^{(j+1)} = f(y_i^{(j)}, u^{(j)})$$

For deep learning algorithms, generally:

$$f(y_i^{(j)}, u^{(j)}) := \sigma\left(K^{(j)} y_i^{(j)} + +\beta^{(j)}\right)$$

where σ is a suitable activation function acting component-wise on its arguments.

In order to view the minimization problem, stated above, as a discretization of an optimal control problem, we need to observe that the constraint equation is the discretization of the ordinary differential equation

(ODE) $\dot{y}_i = f(y_i, u_i)$, $y_i(0) = x_i$, on $[0, T]$, with step-size Δt and with the forward-Euler method. In the continuum, the following optimal control problem is obtained:

$$\min_{y,u,W,\mu} \sum_{i=1}^{m} \|\mathcal{C}\left(Wy_i(T) + \mu\right) - c_i\|^2 + \mathcal{R}(u) \qquad s.t. \qquad \dot{y}_i = f(y_i, u) \quad t \in [0, T], \quad y_i(0) = x_i$$

Assuming that this problem satisfies necessary conditions for optimality, and that a suitable activation function and cost function have been chosen, a number of new deep learning algorithms can be generated. For example, the authors of Chen et al. [2018]; Lu et al. [2018] propose to use accurate approximations of $\dot{y}_i = f(y_i, u)$ obtained by black-box ODE solvers. Alternatively, some of these new strategies are obtained by considering constraint ODEs with different structural properties, e.g. taking f to be a Hamiltonian vector field, and by choosing accordingly the numerical integration methods to approximate the ODE dynamic Haber and Ruthotto [2017]; Chang et al. [2017]. This means augmenting the dimension, e.g. by doubling the number of variables in the ODE, a strategy also studied in Gholami et al. [2019].

An ODE-inspired neural network architecture is uniquely defined by choosing f and specifying a time discretization of $\dot{y}_i = f(y_i, u)$. In this section, lets take a look at two choices for f i.e.

$$f(y, u) = \sigma(Ky + \beta), \qquad u = (K, \beta)$$
$$f(y, u) = \alpha\sigma(Ky + \beta), \qquad u = (K, \beta, \alpha)$$

6.0.1 Runge-Kutta Networks

Here we choose $f(y, u) = \sigma(Ky + \beta), u = (K, \beta)$. For simplicity, focus on the simplest Runge-Kutta methods – the explicit Euler. This corresponds to the ResNet in the machine learning literature.

In this case the network relation (forward propagation) is given by:

$$y^{(j+1)} = y^{(j)} + \Delta t\sigma\left(K^{(j)}y^{(j)} + \beta^{(j)}\right)$$

and the gradients with respect to the controls can be computed by first solving for the adjoint variable (back-propagation)

$$\gamma^{(j)} = \sigma'\left(K^{(j)}y^{(j)} + \beta^{(j)}\right) \odot p^{(j+1)}$$
$$p^{(j+1)} = p^{(j)} - \Delta t K^{(j),T}\gamma^{(j)}$$

and then computing

$$\partial_{K^{(j)}}\mathcal{J}(y^{(N)}) = \Delta t\gamma^{(j)}y^{(j),T}$$
$$\partial_{\beta^{(j)}}\mathcal{J}(y^{(N)}) = \Delta t\gamma^{(j)}$$

where $\mathcal{J}(\cdot)$ is a short-hand representation of the cost function to minimize.

6.0.2 ODENet

We can slightly enlarge the set of control variables and have the function f as: $f(y, u) = \alpha\sigma(Ky + \beta), u = (K, \beta, \alpha)$. The quantity α can be interpreted as varying time-steps for each layers. In this case, the network relation (forward-propagation) is given by;

$$y^{(j+1)} = y^{(j)} + \alpha t\alpha^{(j)}\sigma\left(K^{(j)}y^{(j)} + \beta^{(j)}\right)$$

and the gradient with respect to the controls can be computed by first solving for the adjoint variable (back-propagation)

$$\gamma^{(j)} = \alpha^{(j)}\sigma'\left(K^{(j)}y^{(j)} + \beta^{(j)}\right) \odot p^{(j+1)}$$
$$p^{(j+1)} = p^{(j)} - \Delta t K^{(j),T}\gamma^{(j)}$$

and then computing

$$\partial_{K^{(j)}}\mathcal{J}(y^{(N)}) = \Delta t \gamma^{(j)} y^{(j),T}$$
$$\partial_{\beta^{(j)}}\mathcal{J}(y^{(N)}) = \Delta t \gamma^{(j)}$$
$$\partial_{\alpha^{(j)}}\mathcal{J}(y^{(N)}) = \Delta t \left\langle p^{(j+1)}, \sigma\left(K^{(j)} y^{(j)} + \beta^{(j)}\right)\right\rangle$$

6.1 A Simple Experiment

In this section, let's compare the performance of ResNet, ODENet and traditional feed-forward neural network. For simplicity, we restrict ourselves to relatively small architecture for binary classification problem.

Consider all the models have the same number of nodes at each layer and there are a total of 5 layers – corresponding to total time-steps of 5. A scaling factor $\alpha = 0.15$ is set for both ResNet and ODENet. Note that the time-step at each layer in ODENet is adaptive in the sense that Δt is itself a parameter, unlike in ResNet. Dataset used in the experiment are simple toy-dataset available in `sklearn.datasets.make_moons()` and `sklearn.datasets.make_circles()`. 100 samples are randomly selected to train the models.

ResNet

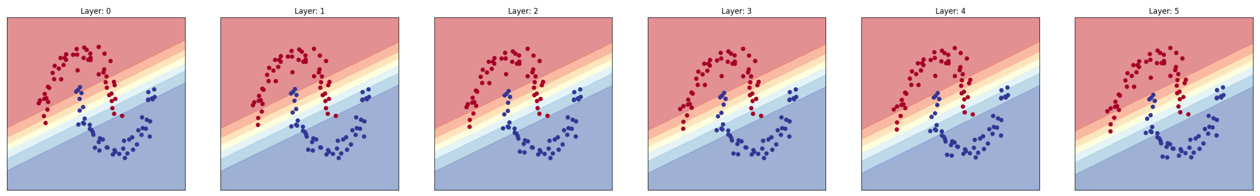


Figure 6: Evolution of data points across each layer of the network (ResNet)

ODENet

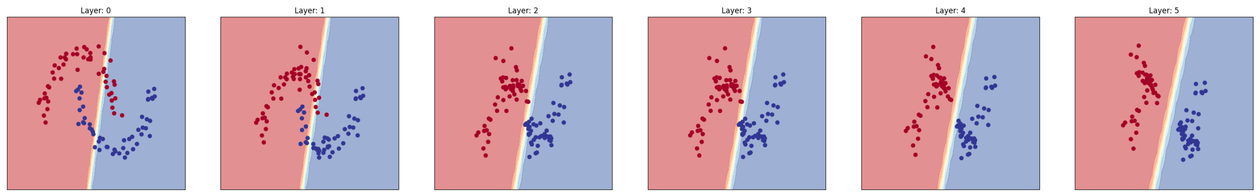


Figure 7: Evolution of data points across each layer of the network (ODENet)

Comparison Decision Boundaries

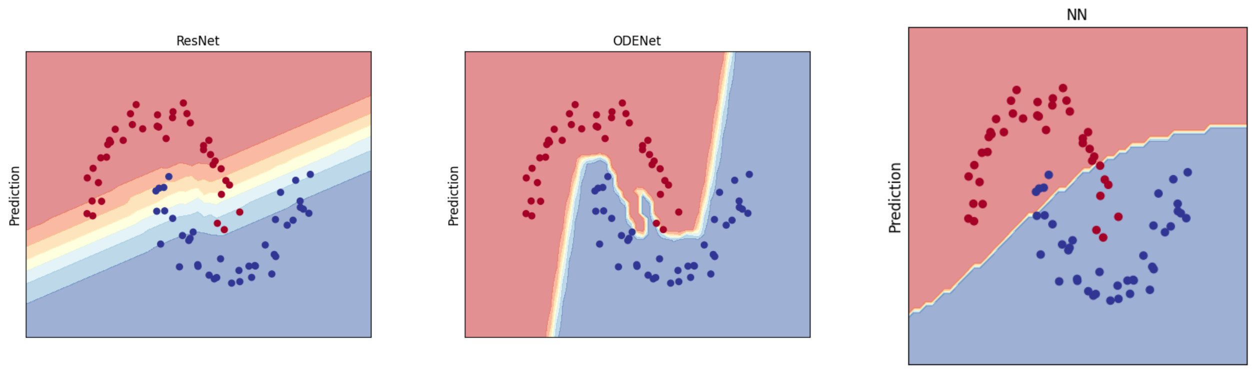


Figure 8: Decision boundaries obtained using ResNet, ODENet and Multi-layer Perceptron (MLP)

	Spiral		Donut	
	Train Time (sec)	Test Accuracy	Train Time (sec)	Test Accuracy
ODENet	29.3	**1.0**	28.5	**1.0**
ResNet	30.7	0.9	24.7	0.55
MLP	248×10^{-3}	0.8625	147×10^{-3}	0.6875

Table 2: Performance Comparison

Notice that the run-time of ODE-based architectures are awfully slow in comparison to the vanilla neural network (MLP). This behavior could be attributed to the fact that ODENets and ResNets requires a numerical ODE solver (e.g. Runge-Kutta, Euler method) to compute the forward pass. Manu high-accuracy solvers (like Runge-Kutta) involve multiple function evaluations per step, which adds to computational overhead. In contrast, traditional NNs compute a simple feedforward pass with a fixed number of operations. In order to highlight this fact, Table 3 shows the computation time for each methods when the number of layers is increased to 15.

	Donut	
	Train Time	Test Accuracy
ODENet	4 min 22 sec	**1.0**
ResNet	5 min 37 sec	0.59
MLP	268×10^{-3}	0.5

Table 3: Vanilla ODENets/ResNets do not scale well with high number of layers in the network (in terms of training time).

Listing 4: ODENet Code

```
def get_cls(control, dimension):
        ix = dimension*(dimension+1)
        jx = -ix+dimension*dimension
        W = control[-ix:jx]
        mu = control[jx:]
        WW = np.array([
            [W[0], W[1]],
            [W[2], W[3]]
        ])
        return WW, mu

activation_function = lambda x: np.tanh(x)

def dynamics_ode(control, X, num_layers, dimension, alpha = 1):
    '''Returns:
        O: array of output for each input sample
        T: list of trajectories for each input sample. It contains all intermediate states
            x_1, x_2, ..., x_final across all layers'''

    O = []
    T = []
    for k in range(X.shape[0]):
        x0 = X[k]
        xn = x0.copy()
        x = [xn]

        # Simulate the forward-pass of a sample input to the NN; as a discrete approximation
            of an ODE:
        for i in range(num_layers):
            ix = (dimension * (dimension + 1) + 1) * i       # 0, 7, 14, 21, 28, 35, ....
            jx = ix + dimension * dimension                  # 4, 11, 18, 25, 32, 39, ....
            zx = jx + dimension                              # 6, 13, 20, 27, 34, 41, ....
            K = control[ix:jx]                               # Current linear-transformation
                params
            bias = control[jx:zx]                          # Bias associated with current
                layer
            Dt = control[zx]                               # Discrete time-step for each
                layer
            KK = np.array([
                [K[0], K[1]],
                [K[2], K[3]]
            ])

            # At each layer the output is obtained as: x_{n+1} = x_n + \alpha * Dt * \phi()
                where \alpha is scaling factor
            xn = (xn + alpha * Dt * activation_function(KK @ xn + bias)).copy()
            xn = xn
            x.append(xn)

        ix = dimension*(dimension+1)
        jx = -ix+dimension*dimension
        W = control[-ix:jx]
        mu = control[jx:]

        # WW maps the final state after all layers to the output space with mu as the output
             layer bias
        WW = np.array([
            [W[0], W[1]],
            [W[2], W[3]]
        ])
        xf = (WW@xn + mu).copy()
        x.append(xf)
        T.append(x)
        sxf = np.exp(xf)
        sxf /= sum(sxf)
        o = sxf[1]
        O.append(o)
    O = np.array(O)

    return O, T


def obj_ode(control, X, y, num_layers, dimension, alpha = 1):
    O, _ = dynamics_ode(control, X, num_layers, dimension, alpha)
    L = y[:O.shape[0]]
    Dts = []

    for i in range(num_layers):
        ix = (dimension*(dimension+1)+1)*i
        jx = ix+dimension*dimension
        zx = jx+dimension
        Dt = control[zx]
        Dts.append(Dt)

    Dts = np.array(Dts)

    return 0.5 * sum((O-L)*(O-L)) + 0.01*sum((1-2*((Dts>0)*(Dts<1)))>0)

dimension = 2
num_layers = 7
dt = 1/num_layers

# Instantiate initial control vector
control_0 = []
for i in range(num_layers):
    K = [1,0,0,1]
    be = [0,0]
    Dt = [dt]
    control_0 += K+be+Dt
W = [1,0,0,1]
mu = [0,0]
control_0 += W+mu
```

```
control_0 = np.array(control_0)


alpha=0.15
X, y = dataset_donut_train
sol_donut = minimize(obj_ode, x0=control_0, args=(X, y, num_layers, dimension, alpha))
optimal_control_donut = sol_donut.x

O_donut, T_donut = dynamics_ode(optimal_control_donut, X, num_layers, dimension, alpha)
WW_donut, mu_donut = get_cls(optimal_control_donut, dimension)
```

6.2 Future Directions

Not sure how feasible these are but some of the possible research directions include:

1. Weight quantization and sparsity constraints?? Use optimal control to dynamically enforce sparsity constraints as the network trains, applying control to minimize the trade-offs between performance and sparsity. Perhaps a framework where quantization levels and sparsity patterns are optimized as part of the control dynamics, ensuring an efficient balance between memory and accuracy

2. Look into faster convergence for control inspired deep architectures?

References

B. Chang, L. Meng, E. Haber, F. Tung, and D. Begert. Multi-level residual networks from dynamical systems view. *arXiv preprint arXiv:1710.10348*, 2017.

R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

F. L. Chernousko and A. Lyubushin. Method of successive approximations for solution of optimal control problems. *Optimal Control Applications and Methods*, 3(2):101–114, 1982.

R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing*, 36(6):C635–C661, 2014.

A. Gholami, K. Keutzer, and G. Biros. Anode: Unconditionally accurate memory-efficient gradients for neural odes. *arXiv preprint arXiv:1902.10298*, 2019.

S. Gunther, L. Ruthotto, J. B. Schroder, E. C. Cyr, and N. R. Gauger. Layer-parallel training of deep residual neural networks. *SIAM Journal on Mathematics of Data Science*, 2(1):1–23, 2020.

E. Haber and L. Ruthotto. Stable architectures for deep neural networks. *Inverse problems*, 34(1):014004, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Q. Li and S. Hao. An optimal control approach to deep learning and applications to discrete-weight neural networks. In *International Conference on Machine Learning*, pages 2985–2994. PMLR, 2018.

G.-H. Liu and E. A. Theodorou. Deep learning theory review: An optimal control and dynamical systems perspective. *arXiv preprint arXiv:1908.10920*, 2019.

Y. Lu, A. Zhong, Q. Li, and B. Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In *International Conference on Machine Learning*, pages 3276–3285. PMLR, 2018.

P. Parpas and C. Muir. Predict globally, correct locally: Parallel-in-time optimal control of neural networks. *arXiv preprint arXiv:1902.02542*, 2019.

E. Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 1(5):1–11, 2017.

E. Weinan, J. Han, and Q. Li. A mean-field optimal control formulation of deep learning. *arXiv preprint arXiv:1807.01083*, 2018.