CMU16-745 Optimal Control Lecture Notes

Zhihai Bi

May 9, 2024

Contents

1 Continuous-Time Dynamics and Stability

1.1 Continuous-Time Dynamics

The most general way of describing smooth dynamical systems is via a dynamics function:

$$\dot{x} = f(x, u) \tag{1}$$

Here, $x \in \mathbb{R}^n$ describes the state of the system, $u \in \mathbb{R}^m$ is the control input provided to the system, and f, the dynamics function, specifies how the system evolves with the application of control inputs. $\dot{x}$ provides the derivatives (with respect to time) of state.

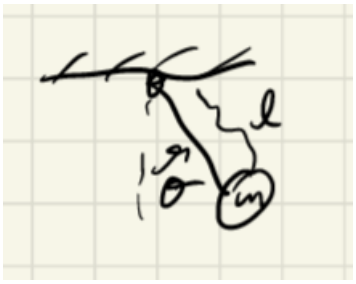

Figure 1: Simple pendulum system

For a mechanical system, the state x is usually described as:

$$x = \begin{bmatrix} q \\ v \end{bmatrix} \tag{2}$$

where q describes the configuration of the system, and v describes the velocity.(q is not always a vector and v is)

1.1.1 Validity of Assumptions

Note that the configuration isn't necessarily a vector, and the velocities are not necessarily derivatives of configuration! Also, remember that this description of the system is valid only when the dynamics are smooth. This is broken when, for instance, the system experiences contacts.

1.1.2 Example

Consider the example of a simple pendulum fig. 1. The system dynamics can be captured in the following equation:

$$ml^2\ddot{\theta} + mgl\sin(\theta) = \tau \tag{3}$$

Here, the configuration q is given by the pendulum angle θ, the velocity v is specified by $th\dot{e}ta$, and the control inputs u are given by the torque applied to the system τ.
Thus the state x is:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} \tag{4}$$

The velocity $\dot{x}$ is:

$$\dot{x} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin\theta + \frac{1}{ml^2}u \end{bmatrix} \tag{5}$$

Here, the system dynamics $f(x, u)$ are given as

$$f(x, u) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin\theta + \frac{1}{ml^2}u \end{bmatrix} \tag{6}$$

The manifold of state here is described as $x \in \mathbb{S}^1 \times \mathbb{R}$, a cylinder.

1.2 Control-Affine Systems

Many systems (mechanical systems in particular) can be defined as a *control-affine system*. This is a specific form of the general dynamics described above, where the control inputs affect the system via an affine matrix G.

$$\dot{x} = f_0(x) + G(x)u \tag{7}$$

For instance, the pendulum system can be described as a control-affine system, where

$$f_0(x) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\theta) \end{bmatrix} \quad G(x) = \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} \tag{8}$$

In driftless systems, $f_0(x) = 0$. One can also convert systems to control-affine systems, by adding original control inputs u to state, and setting the modified control inputs (of the control-affine system) to be the derviatives of u.

1.3 Manipulator Dynamics

Another common form of expressing dynamics is -

$$M(q)\dot{v} + C(q, v) = B(q)u \tag{9}$$

Where $M(q)$ is the mass matrix of the system, $C(q, v)$ is the dynamics bias (including Coriolis terms and gravity), and $B(q)$ is the control input jacobian. As usual, q is the configuration and v is the velocity. Here, the change in configuration $\dot{q}$:

$$\dot{q} = G(q)v \tag{10}$$

describe the kinematics of the system.

$$\dot{x} = f(x, u) = \begin{bmatrix} G(q)v \\ -M(q)^{-1}(B(q)u - C) \end{bmatrix} \tag{11}$$

In the pendulum system, the mass matrix $M(q) = ml^2$, $C(q, v) = mgl\sin(\theta)$, $B = I$, $G = I$.

All mechanical systems can be described in this form; this is because this form is a different way of expressing the Euler-Lagrange equation for: (kinetic energy - potential energy)

$$L = 1/2\ v^T M(q)v - V(q) \tag{12}$$

1.4 Linear Systems

Linear systems are a common way of expressing control problems and designing controllers since we know how to solve linear systems, and they are relatively simple to deal with.

$$\dot{x} = A(t)x + B(t)u \tag{13}$$

when matrices $A(t)$ and $B(t)$ are constant over time, these linear systems are either linearly time-invariant. Otherwise, they are linearly time-varying systems.

We often approximate non-linear systems with linear systems by linearizing the system around the current state.

$$\dot{x} = f(x, u) \tag{14}$$

where $A = \frac{\partial f}{\partial x}$ and $B = \frac{\partial f}{\partial u}$.

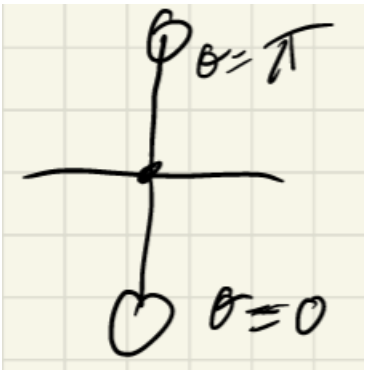

Figure 2: Equilibria points of simple pendulum system

1.5 Equilibria

We also want to understand equilibria - i.e. the point where a system will remain at rest.

$$\dot{x} = f(x, u) = 0 \tag{15}$$

Algebraically, this is described by roots of the dynamics equation.

In the pendulum example (without a control input) fig. 2

$$\dot{x} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\theta) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{16}$$

The roots of this occur when $\theta = 0$ and $\dot{\theta} = 0, \pi$.

1.5.1 First control problem

Lets consider a small control problem. Let's try to move the equilibrium point of the pendulum, by applying an appropriate control input. In this case

$$\dot{x} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\frac{\pi}{2}) + \frac{1}{ml^2}u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{17}$$

Solving for u, we have:

$$\frac{1}{ml^2}u = -\frac{g}{l}\sin(\frac{\pi}{2}) \implies u = mgl \tag{18}$$

In general, we get a root-finding problem in u:

$$f(x^*, u) = 0 \tag{19}$$

1.5.2 Stability of Equilibria

We'd also like to understand when a system will return to its equilibrium point upon being perturbed. Consider a 1-D system, $x \in \mathbb{R}$, as in fig. 3.

In this case, when $\frac{\partial f}{\partial x} < 0$, the system is stable, because it gets pushed back to the equilibrium point by virtue of the dynamics. Conversely, for $\frac{\partial f}{\partial x} > 0$, the system is unstable, because it gets pushed away from the equilibrium point. The region of x such that $\frac{\partial f}{\partial x} < 0$ is called the basin of attraction of a system.

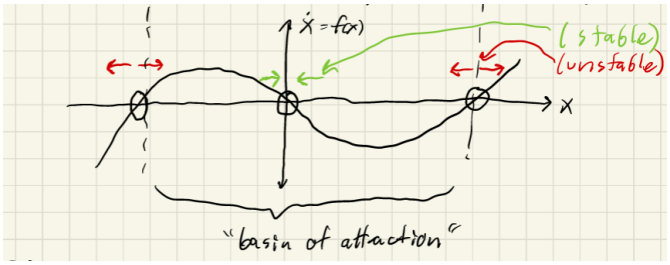

Figure 3: Stability of equilibria of a 1-D system.

In higher dimensions, $\frac{\partial f}{\partial x}$ is a Jacobian matrix. To study stability of such a system, we can take an Eigen decomposition of the Jacobian (decoupling it into n 1-D systems). If the real-component of each of the eigven values of the Jacobian is less than 0, the system is stable. Mathematically, the system is stable if

$$\text{Re}\big[eig(\frac{\partial f}{\partial x})\big] < 0 \tag{20}$$

and unstable otherwise.

In our pendulum example,

$$f(x) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\theta) \end{bmatrix} \implies \frac{\partial f}{\partial x} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos(\theta) & 0 \end{bmatrix} \tag{21}$$

Consider the equilibrium point $\theta = \pi$.

$$\implies \frac{\partial f}{\partial x}\Big|_{\theta=\pi} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} \implies eig(\frac{\partial f}{\partial x}\Big|_{\theta=\pi}) = \pm\sqrt{\frac{g}{l}} \tag{22}$$

Since one eigenvalue is positive, the equibilrium point $\theta = \pi$ is unstable. In contrast, at the equilibrium point $\theta = 0$.

$$\implies \frac{\partial f}{\partial x}\Big|_{\theta=0} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} \implies eig(\frac{\partial f}{\partial x}\Big|_{\theta=\pi}) = 0 \pm i\sqrt{\frac{g}{l}} \tag{23}$$

In this case, the real part of the eigenvalue is 0. In this purely imaginary eigenvalue case, the system is called marginally stable, and experiences undamped oscillations. Adding damping to the systems (such as applying control inputs $u = -K_d\dot{\theta}$, results in strictly negative eigenvalues, leading to a stable system.

1.6 References

Remember to check out [2] and [3] for this course and these topics!

References

[1] J. Nocedal et al. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006. ISBN: 9780387303031. URL: https://books.google.com/books?id=eNlPAAAAMAAJ.

[2] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005. ISBN: 9780471649908. URL: https://books.google.com/books?id=A0OXDwAAQBAJ.

[3] S.H. Strogatz and M. Dichter. *Nonlinear Dynamics and Chaos, 2nd ed. SET with Student Solutions Manual*. Studies in Nonlinearity. Avalon Publishing, 2016. ISBN: 9780813350844. URL: https://books.google.com/books?id=vUWhDAEACAAJ.

2 Discrete-Time Dynamics and Stability

This chapter covers the following topics!

- Continuous ODEs, and how to use discrete time simulation for this.
- More on stability.

2.1 Motivation

- In general cases (i.e. when we don't have convenient forms of f), we can't solve $\dot{x} = f(x)$ for $x(t)$. This implies we can't design appropriate controllers either.
- In these cases, we solve such systems computationally and need to represent $x(t)$ in discrete time.
- Happily, discrete-time models can also capture some effects that continuous ODEs cannot capture, and so in some sense are more general than their continuous counterparts. For example, contacts, and other discontinuous events, can be more easily captured by discrete-time models.

2.2 Discrete-Time Dynamics

Let's consider the explicit form of such discrete-time systems, where the state at the next timestep is determined as:

$$x_{k+1} = f_{\text{discrete}}(x_k, u_k) \tag{1}$$

The simplest discretization that we could use is Forward Euler integration

$$x_{k+1} = x_k + h f_{\text{continuous}}(x_k, u_k) \tag{2}$$

Where h denotes the time step.
Let's go back to the pendulum example. If we use $l = m = 1$, $h = 0.1$, or $h = 0.01$. This blows up! (Check out the notebook in the Lecture). Why does it blow up? To answer this, we need to discuss the stability of discrete-time systems.

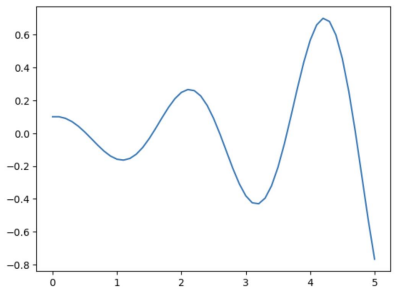

Figure 4: The energy of the system.

2.3 Stability of Discrete-Time Systems

Remember, in continuous time, we have that a system is stable when:

$$\text{Re}\big[eig(\frac{\partial f}{\partial x})\big] < 0 \tag{3}$$

Now in discrete time, these dynamics are an in iterated map:

$$x_N = f_{\text{d}}(f_{\text{d}}(f_{\text{d}} ... f_{\text{d}}(x_0))) \tag{4}$$

If we linearize this and apply the chain rule:

$$\frac{\partial x_N}{\partial x_0} = \frac{\partial f_\mathrm{d}}{\partial x}\frac{\partial f_\mathrm{d}}{\partial x}...\frac{\partial f_\mathrm{d}}{\partial x}\Big|_{x_0} \tag{5}$$

If A_d represents $\frac{\partial f_\mathrm{d}}{\partial x}$ around the point of linearization, this A_d stays the same over iterations, thus the above expression is:

$$\frac{\partial x_N}{\partial x_0} = \frac{\partial f_\mathrm{d}}{\partial x}\frac{\partial f_\mathrm{d}}{\partial x}...\frac{\partial f_\mathrm{d}}{\partial x}\Big|_{x_0} = A_\mathrm{d}^N \tag{6}$$

Let's say the equilibrium point is the origin $x = 0$ (which we can do without loss of generality by a change of coordinates). Thus the stability of this system implies that:

$$\lim_{k\longrightarrow\infty} A_\mathrm{d}^k x_0 = 0 \quad \forall x_0 \tag{7}$$

If this is true $\forall x_0$, this implies A_d^k itself must follow:

$$\lim_{k\longrightarrow\infty} A_\mathrm{d}^k = 0 \tag{8}$$

This implies that the Eigenvalues of A_d must satisfy:

$$|\mathrm{eig}(A_\mathrm{d})| < 1 \tag{9}$$

Equivalently, the eigenvalues of A_d must lie within the unit circle on the complex plane.

2.3.1 Forward Euler integration

Let's try this out on our example of the pendulum with Forward Euler integration. We have:

$$x_{k+1} = x_k + hf(x_k) = f_\mathrm{d}(x_k) \tag{10}$$

We know:

$$A_\mathrm{d} = \frac{\partial f_\mathrm{d}}{\partial x_k} = I + hA_\mathrm{continuous} \tag{11}$$

Remember from last lecture:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos(\theta) & 0 \end{bmatrix} \tag{12}$$

So:

$$A_\mathrm{d} = \frac{\partial f_\mathrm{d}}{\partial x_k} = I + h\begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos(\theta) & 0 \end{bmatrix} \tag{13}$$

What are the Eigenvalues of this A_d?

$$\mathrm{eig}(A_\mathrm{d}\big|_{\theta=0}) = 1 \pm 0.313i \tag{14}$$

Since this doesn't lie in the unit circle, this is unstable! If we create a plot of $\mathrm{eig}(A_\mathrm{d})$ versus different values of h, we observe the plot in fig. 5. We see the system is marginally stable in the limit of $h \to 0$. This means that for any value of timestep h, the system is going to blow up!

Take-Away Messages

- Be careful when discretizing ODEs!
- Do a sanity check based on the equilibrium energy behavior (whether there is conservation and / or dissipation of energy).
- This is an artifact of forward Euler integration - it always overshoots the function it is trying to approximate (in this case, it overestimates the acceleration of the system), so avoid it! Especially for undamped systems!

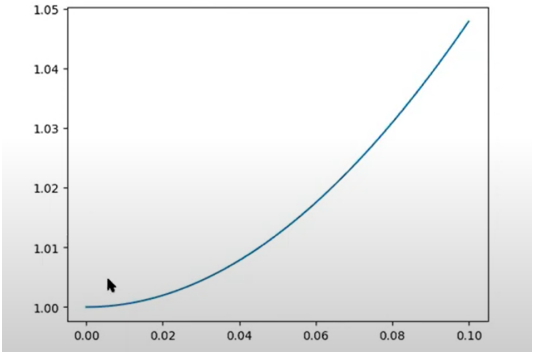

Figure 5: Plot of eigenvalues versus step size.

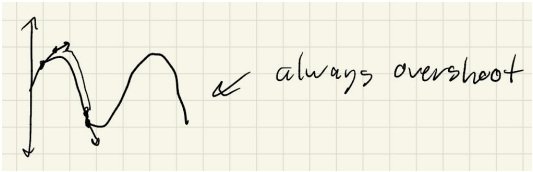

Figure 6: The meaning of overshoot. It's a linear estimation, So it would overshoot when the abs of slope less than 1

2.3.2 A better explicit integrator: RK4

A better explicit integrator to use is the 4th order Runge-Kutta method, the "industry standard". The intuition of this is that Euler Integration fits a line segment over each timestep, whereas the 4th-order Runge-Kutta method (RK4) fits a cubic polynomial, which is much more expressive, and allows for much better accuracy in capturing the underlying function.

4th Order Runge-Kutta Integration

1: $x_{k+1} = f_{RK4}(x_k)$
2: $K_1 = f(x_k)$ ▷ Evaluate at beginning.
3: $K_2 = f(x_k + \frac{1}{2}hK_1)$ ▷ Evaluate at midpoint.
4: $K_3 = f(x_k + \frac{1}{2}hK_2)$ ▷ Evaluate at midpoint.
5: $K_4 = f(x_k + hK_3)$ ▷ Evaluate at end.
6: $x_{k+1} = x_k + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$ ▷ Evaluate as weighted average of intermediate computations.

Let's try this on our Pendulum example. Instead of the explosion we observed before, the eigenvalues are right within the unit circle and reflect the true continuous system.

Take-Away Message

The 4th order Runge-Kutta method provides us with much improved accuracy, that easily outweighs the additional computational cost of 4 function evaluations instead of 1 that forward Euler integration did. That said, even sophisticated integrators have issues, depending on the value of timesteps, etc. So always perform some kind of sanity check!

2.3.3 Backward Euler Integration

Another thing to consider is the implicit form of an ODE. For example, the implicit form of a discrete-time dynamical system is:

$$f_{\mathrm{d}}(x_{k+1}, x_k, u_k) = 0 \tag{15}$$

The simplest example of this is:

$$x_{k+1} = x_k + hf(x_{k+1}) \tag{16}$$

which is *Backward Euler Integration*, where we are evaluating the function f at a future time.
How do we simulate this kind of system? You can write this as:

$$f_{\mathrm{d}}(x_{k+1}, x_k, u_k) = x_k + hf(x_{k+1}) - x_{k+1} = 0 \tag{17}$$

You can then treat this as a root-finding problem in x_{k+1}. (We'll spend time on this next week).
In the pendulum example, we observe the following:

- We see the opposite energy behavior from forward Euler integration.
- Discretization adds artificial damping to the system, instead of adding energy to the system.
- Backward Euler undershoots the function being approximated.
- While this doesn't reflect the physical process (is unphysical), this effect allows simulators to take big steps, and can be convenient sometimes!
- It is very common to use this in quick and easy low-fi simulators in graphics and robotics.

Take-Away Message!

- Implicit methods are often "more stable" than their explicit counterparts.
- For forward simulation, solving the implicit equation can be more expensive.
- In many "direct" trajectory optimization methods, they are *not* any more expsensive to use!

2.4 Discretizing Control Input

So far, we've focused on discretizing the state of the system $x(t)$. We also have the control input to the system, $u(t)$ that we need to discretize! The simplest option here is to set:

$$u(t) = u_k \; t_k \leq t \leq t_{k+1} \tag{18}$$

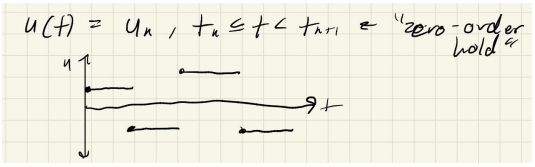

Figure 7: zero-order hold

This is a "zero-order hold", where the control input at time t is simply equal to the control input within the corresponding time interval. This is a piecewise constant control. This is easy to implement, but may require lots of knot points (sample points) to accurately capture a continuous control input $u(t)$. There are possibly better options, such as:

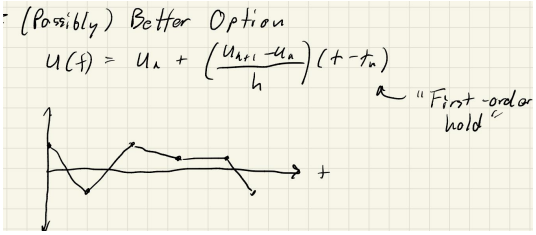

Figure 8: first order hold

$$u(t) = u_k + \frac{u_{k+1} - u_k}{h}(t - t_n) \tag{19}$$

This is a "first-order hold", or a piecewise *linear* control. The control input is a linear function of the control inputs at the endpoints of the time interval. This first-order hold can better approximate continuous $u(t)$ with fewer knot points, thus leading to faster solve times, etc. It's not too much more work than the zero-order hold computationally speaking. If you know your system is continuous, use the first order hold! It's also very common to do this, such as classic DIRCOL (direct collocation).
There are other options too:

- We can progressively move to higher-order holds, using higher-order polynomials.
- In many control applications, $u(t)$ is *not* smooth, such as in bang-bang control methods. In these cases, higher-order polynomials are not the best choice, and are not good approximations.

In general, zero-order and first-order holds are the most common ways to discretize control in practice.

3 Root Finding and Minimization

- This lecture covers some notation that will be useful in representation of matrices.
- We'll also go over root finding
- minimization techniques, with and without equality constraints.

Also, remember to check out the lecture recording to get an overview of Homework 1!

3.1 Notation

Consider a function $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$. For this function, we have its derivatives

$$\frac{\partial f}{\partial x} \in \mathbb{R}^{1\times n} \tag{1}$$

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

$$f(x + \Delta x) \approx f(x) + \frac{\partial f}{\partial x}\ \Delta x \tag{2}$$

Notice that $\Delta x \in \mathbb{R}^{n\times 1}$.

Similarly, given a function $g(y) : \mathbb{R}^m \longrightarrow \mathbb{R}^n$, we have:

$$\frac{\partial g}{\partial y} \in \mathbb{R}^{n\times m} \tag{3}$$

because $g(y + \Delta y) \approx g(y) + \frac{\partial g}{\partial y}\ \Delta y$. Notice that $\Delta y \in \mathbb{R}^{m\times 1}$.

This is important because it makes the chain rule work:

$$f(g(y + \Delta y)) \approx f(g(y)) + \frac{\partial f}{\partial x}\Big|_{g(y)} \frac{\partial g}{\partial y}\Big|_{y}\ \Delta y \tag{4}$$

For convenience, we will also define the gradient:

$$\nabla f(x) = (\frac{\partial f}{\partial x})^T \in \mathbb{R}^{n\times 1} \tag{5}$$

which is a column vector. Further, the second derivative (or the Hessian) is

$$\nabla^2 f(x) = (\frac{\partial^2 f}{\partial x^2})^T = \frac{\partial}{\partial x}(\nabla f(x)) \in \mathbb{R}^{n\times n} \tag{6}$$

3.2 Root Finding

Now that we have our notation fixed, let's consider how we can find a "root". Consider a function $f(x)$. We want to find a "root" x^* such that

$$f(x^*) = 0. \tag{7}$$

Example: equilibrium point of a continuous-time dynamics

This is closely related to finding the fixed point of a system:

$$f(x^*) = x^* \tag{8}$$

Fixed points are equilibrium points of *discrete* time versions of these ODEs.

3.2.1 Fixed Point Iteration

The simplest solution method, If the fixed point is stable, just iterate the "dynamics" until you settle into the fixed point

$$x \longleftarrow f(x) \tag{9}$$

It only works for stable fixed points and has slow convergence.

3.2.2 Newton's Method

One way to go about root finding is to use Newton's method. Here, we use a Taylor approximation of our function around our initial guess and solve for x around this guess.

$$f(x+\Delta x) \approx f(x) + \frac{\partial f}{\partial x}\Big|_x \Delta x = 0 \implies \Delta x = -(\frac{\partial f}{\partial x})^{-1} f(x) \tag{10}$$

We can then find the root of the system by repeatedly updating our estimate of the root with $x \leftarrow x + \Delta x$, until the root converges.

How to cal the jacobian ($\frac{\partial f}{\partial x}$) ? x is the x(n) in a discrete function? How to do in Julia?

3.2.3 Example: Backward Euler

We can do this using Backward Euler Newton's method; Newton's method thus offers very fast convergence when compared to fixed point iteration.

Take-Away Messages

- Quadratic convergence rate.
- We can achieve machine precision.
- Most expensive part of this operation is solving a linear system, which is an $O(n^3)$ operation.
- It's possible to improve upon the time complexity by taking advantage of the structure of the problem. (We'll touch on this later!)

3.3 Minimization

Consider finding a value x for which the function $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$ attains its minimum value. If f is smooth, $\frac{\partial f}{\partial x}\big|_{x^*} = 0$ at local minima. This means we can go about minimizing f by applying Newton's root finding to $\frac{\partial f}{\partial x} = 0$. Consider:

$$\nabla f(x+\Delta x) \approx \nabla f(x) + \nabla^2 f(x)\Delta x = 0 \implies \Delta x = -(\nabla^2 f(x))^{-1}\nabla f(x) = 0 \tag{11}$$

This means we can find our minimum by repeatedly updating our estimate x as $x \leftarrow x + \Delta x$ until convergence.
The intuition for this is that we are fitting a quadratic approximation to $f(x)$, using a Taylor expansion at the current guess of the solution. We can exactly minimize quadratic approximations.
Remember to look at the Jupyter notebook in the lecture video for an example $f(x) = x^4 + x^3 - x^2 - x$.

Take-Away Messages

- Newton's root finding is a **local** method. It will find the **closest** fixed point to the initial guess, which could be a maxima, minima, or a saddle point.

3.4 Sufficient Conditions

In the example, we saw that Newton's method doesn't really care about whether it is maximizing or minimizing the function at hand. So how do we know which we're doing? Let's think about the scalar case:

$$\Delta x = -(\nabla^2 f)^{-1}\nabla f \tag{12}$$

Here, the $-$ signifies performing descent on the function value, the term $(\nabla^2 f)^{-1}$ is analogous to a learning rate, and ∇f is simply the gradient.

$$\nabla^2 f > 0 \implies \text{Descent (minimization)} \tag{13}$$

$$\nabla^2 f < 0 \implies \text{Ascent (maximization)} \tag{14}$$

In the $\mathbb{R}^n$ case, the equivalent conditions are whether the second derivative Hessians are positive definite (for descent / minimization) or vice versa.
If $\nabla^2 f > 0$ everywhere in the function domain, we have that f is strongly convex. We can always find a global minimum by Newton's method. Unfortunately, this doesn't really hold true for hard / nonlinear problems.

3.5 Regularization(Hessian is not positive)

What do we do if we don't have a strongly convex function? The practical solution is to make sure we are actually always minimizing while executing Newton's method.

$$H \leftarrow \nabla^2 f \tag{15}$$

Now while $H \not\succ 0$ (while the Hessian is not positive definite), we can iteratively set $H \leftarrow H + \beta I$, where β is a scalar hyper-parameter.

Regularization / Damped Newton's Method

1: $H \leftarrow \nabla^2 f$
2: **while** $H \not\succ 0$ **do** ▷ Not positive definite.
3: $H \leftarrow H + \beta I$ ▷ β is a scalar hyperparameter.
4: $\Delta x = -H^{-1}\nabla f$
5: $x \leftarrow x + \Delta x$

After modifying the Hessian in this way, we can then take a "Newton" step with $\Delta x = -H^{-1}\nabla f$, $x \leftarrow x + \Delta x$. This modified Newton's method is called "Damped Newton's Method". The trick of modifying the Hessian is called regularization. It guarantees we are performing descent (minimization), and shrinks the step size of the step taken. Remember to check out the lecture video for the behavior in our example!

3.6 Line Search(overshooting problem)

One problem that we ran into, is that the step Δx is often too big, and overshoots the minimum of the function. To fix this, we can check the function value at $f(x + \Delta x)$ and "backtrack" until we get a "good" reduction.
There are many strategies that exist to do this. One such strategy that is both simple and effective, is the Armijo rule:

Armijo Rule

1: $\alpha = 1$ ▷ Step Length
2: **while** $f(x + \alpha\Delta x) > f(x) + b\alpha\nabla f(x)^T \Delta x$ **do** ▷ $\alpha\nabla f(x)^T\Delta x$ is the expected reduction from gradient
3: $\alpha \leftarrow c\alpha$ ▷ c is a scalar < 1.

The intuition for this is to make sure the step size agrees with linearization(taylor expansion) within some tolerance b, if it not agree, it means that I might take a big step that I don't want to do, so decline the α to limit the step. Typical values of the parameters above are: $c = 0.5$, $b = 10^{-4}$ to 0.1.

Take-Away Messages

- Newton's method with simple and cheap modifications or globalization strategies (such as regularization), is extremely effective at finding local minima.

3.7 References

Remember to check out [1] for more details on this.

References

[1] J. Nocedal et al. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006. ISBN: 9780387303031. URL: https://books.google.com/books?id=eNlPAAAAMAAJ.

[2] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005. ISBN: 9780471649908. URL: https://books.google.com/books?id=A0OXDwAAQBAJ.

[3] S.H. Strogatz and M. Dichter. *Nonlinear Dynamics and Chaos, 2nd ed. SET with Student Solutions Manual*. Studies in Nonlinearity. Avalon Publishing, 2016. ISBN: 9780813350844. URL: https://books.google.com/books?id=vUWhDAEACAAJ.

4 Constrained Minimization

4.1 Equality Constraints

Let's consider the following minimization problem, subject to some *equality constraints*:

$$\min_x f(x) \quad ; \quad f(x) : \mathbb{R}^n \longrightarrow \mathbb{R} \tag{1}$$

$$s.t. \;\; c(x) = 0 \quad ; \quad c(x) : \mathbb{R}^n \longrightarrow \mathbb{R}^m \tag{2}$$

4.1.1 First order necessary conditions

The first order necessary conditions for the x at which the function attains its minimum:

- Need $\nabla f(x) = 0$ in the unconstrained / free directions.
- Need $c(x) = 0$

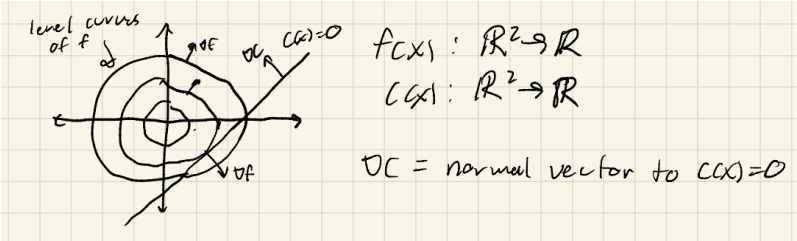

Figure 9: Level curves of $f(x)$

If c(x) is a straight line, then surely it is tangent to c(x) that f(x) achieves its minimum (fig. 9). The same is true if c(x) is N-dimensional. The fact that they minimize when they are tangent means that their normal vectors (derivatives) have to be collinear. Thus any non-zero component of ∇f must be normal to the constraint surface/manifold.

$$\nabla f + \lambda \nabla c = 0 \text{ for some } \lambda \in \mathbb{R} \tag{3}$$

where λ is a Lagrange multiplier or "Dual variable". In general, consider the multi-dimension situation, we have that:

$$\frac{\partial f}{\partial x} + \lambda^T \frac{\partial c}{\partial x} = 0 \quad \lambda \in \mathbb{R}^m \tag{4}$$

Based on this gradient condition, we define:

$$L(x, \lambda) = f(x) + \lambda^T c(x) \tag{5}$$

where $L(x, \lambda)$ represents the Lagrangian. This is subject to:

$$\nabla_x L(x, \lambda) = \nabla f + (\frac{\partial c}{\partial x})^T \lambda = 0 \tag{6}$$

$$\nabla_\lambda L(x, \lambda) = c(x) = 0 \tag{7}$$

together, these conditions represent the KKT conditions. We can now solve this jointly in x and λ as a root finding problem, using Newton's method.

$$\nabla_x L(x + \Delta x, \lambda + \Delta\lambda) \approx \nabla_x L(x, \lambda) + \frac{\partial^2 L}{\partial x^2}\Delta x + \frac{\partial^2 L}{\partial x \partial \lambda}\Delta\lambda = 0 \tag{8}$$

$$\nabla_\lambda L(x + \Delta x, \lambda) \approx c(x) + \frac{\partial c}{\partial x}\Delta x = 0 \tag{9}$$

where $\frac{\partial^2 L}{\partial x \partial \lambda} = (\frac{\partial c}{\partial x})^T$. This can be written as the following matrix system:

$$\begin{bmatrix} \frac{\partial^2 L}{\partial x^2} & (\frac{\partial c}{\partial x})^T \\ \frac{\partial c}{\partial x} & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x L(x, \lambda) \\ -c(x) \end{bmatrix} \tag{10}$$

The first matrix here is the Hessian of the Lagrangian, and describes the KKT system.

4.1.2 Gauss-Newton Method

$$\frac{\partial^2 L}{\partial x^2} = \nabla^2 f + \frac{\partial}{\partial x}\left[(\frac{\partial c}{\partial x})^T \lambda\right] \tag{11}$$

Here, $\frac{\partial}{\partial x}\left[(\frac{\partial c}{\partial x})^T \lambda\right]$ is expensive to compute. We often drop this term, ignoring the "constraint curvature". Dropping this term is called Gauss-Newton method. This is equivalent to linearizing the system first, and then performing Newton's method.
Let's see what's the difference between Full Newton and Gauss-Newton. First, we set the initial guess at [-1,-1] using newTon and Guass-Newton, both find the optimal value:

```
In [40]: xnew, λnew = newton_step(xguess[:,end], λguess[end])
         xguess = [xguess xnew]
         λguess = [λguess λnew]
         plot_landscape()
         plot(xguess[1,:], xguess[2,:],
             "rx")
```

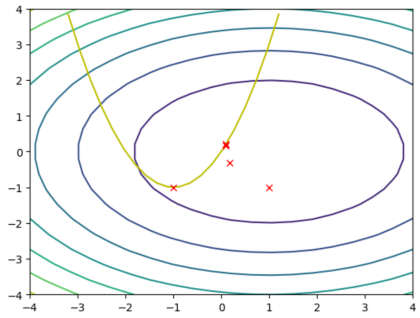

Figure 10: Full Newton's method in initial guess [-1,-1]

But, once we set the initial guess at [-3,2] using full Newton's method, it fails:

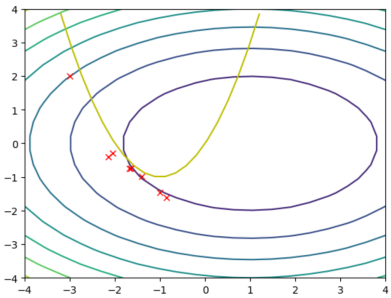

Figure 11: Full Newton's method in initial guess [-3,2]

Because the hessian matrix led to the wrong direction, we can see the hessian matrix right now:

```
In [17]: H = ∇2f(xguess[:,end]) + ForwardDiff.jacobian(x -> ∂c(x)'*λguess[end], xguess[:,end])
Out[17]: 2×2 Matrix{Float64}:
          -2.71881   0.0
           0.0       1.0
```

Figure 12: Hessian matrix after several iterations in initial guess [-3,2]

We can handle it with Gauss-Newton by dropping the $\frac{\partial^2 L}{\partial x^2}$ term or regularize the $\frac{\partial^2 L}{\partial x^2}$ term in full Newton's method,let's see the performance of Gauss-Newton's method

Take-Away Message

- May need to regularize $\frac{\partial^2 L}{\partial x^2}$ in Newton's method, even if $\nabla^2 f > 0$.

```
In [33]: xnew, λnew = gauss_newton_step(xguess[:,end], λguess[end])
         xguess = [xguess xnew]
         λguess = [λguess λnew]
         plot_landscape()
         plot(xguess[1,:], xguess[2,:], "rx")
```

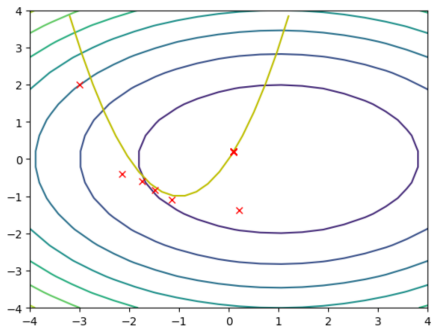

Figure 13: Gauss-Newton's method in initial guess [-3,2]

- Gauss-Newton is slightly slower convergence than full newton (more iterations), but much cheaper per iteration.
- Gauss-Newton is often used in practice

4.2 Inequality constraints

More generally, we have to reconcile with inequality constraints as well. Consider a minimization problem subject to inequality constraint:

$$\min_x f(x) \tag{12}$$

$$s.t. \quad c(x) \geq 0 \tag{13}$$

Let's look at the case of purely inequality constraints for now. In general, these methods are combined with the equality constraint methods above, to handle general inequality / equality constraints in the same problem.

4.2.1 First-Order Necessary Conditions

As before, the first order conditions specify that -

- Need $\nabla f(x) = 0$ in the unconstrained / free directions.
- Need $c(x) \geq 0$ Note the inequality here.

Mathematically,

$$\nabla f - (\frac{\partial c}{\partial x})^T \lambda = 0 \leftarrow \text{"Stationarity"} \tag{14}$$

$$c(x) \geq 0 \leftarrow \text{"Primal Feasibility"} \tag{15}$$

$$\lambda \geq 0 \leftarrow \text{"Dual Feasibility"} \tag{16}$$

$$\lambda^T c(x) = 0 \leftarrow \text{"Complementarity"} \tag{17}$$

Together, these conditions specify the full set of KKT conditions. Notes that whatever the sign of c(x) or the λ, it depends on what you are used to it, and one principle is that it always prevents the cost change.

4.2.2 Intuition

The intuition for these conditions is as follows:

- If constraint is active, (we are on the constraint manifold), we have $c(x) = 0 \implies \lambda = 0$. This is the same as the equality case.
- If the constraint is inactive, we have $c(x) > 0 \implies \lambda = 0$, which is the same as the *unconstrained* case.
- Essentially complementarity ensures either one of λ and $c(x)$ will be 0, or "on /off switching" of constraints.

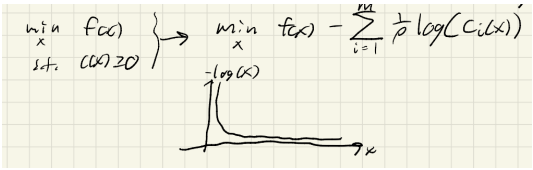

Figure 14: Barrier Function

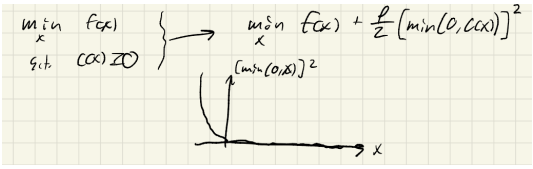

Figure 15: Penalty Method Objective

4.2.3 Algorithms

The implementation of optimization methods here is much harder than the equality case; we can't directly apply Newton's method to the KKT conditions. There are many options we can use, with different trade-offs.

Active-Set Method

- Applies when you have a good way of knowing which constraints are active / inactive.
- Solves equality constrained problem (when it is active).
- Very fast if you have a good heuristic.
- Can be very (combinatorially) bad if you don't know which constraints are active.

Barrier / Interior-Point Method

- We replace the inequalities with a "barrier function", in the objective, that blows up at the constraint boundary fig. 14.
- This is the gold standard for small to medium *convex* problems.
- Requires a lot of hacks and tricks to get working for non-convex problems.

Penalty Method

- Replace the inequality constraint with an objective term that penalizes violations fig. 15.
- It's easy to implement.
- Has issues with being ill-conditioned (the Hessian has a huge spread of eigenvalues, that causes Newton's method to struggle).
- Difficult to achieve high accuracy.

So, the difference between the penalty and barrier method is that the barrier function prevents you to cross the boundary, but the penalty function only works when you are out of the boundary.

Augmented Lagrangian

- Add a Lagrange multiplier estimate to the penalty method, to fix issues that penalty method faced.

$$\min_x f(x) - \tilde{\lambda}^T c(x) + \rho/2\big[\min(0, c(x))\big]^2 \tag{18}$$

where $f(x) - \tilde{\lambda}^T c(x) + \rho/2\big[min(0, c(x))\big]^2 = L_\rho(x, \lambda)$ is the augmented lagrangian. Remember, here we want to penalize the constraints being violated, that's why we have a $-\tilde{\lambda}$ instead of $+\lambda$. We first minimize with respect to x (with a fixed $\tilde{\lambda}$, and then we update $\tilde{\lambda}$ by "offloading" penalty term at each iteration:

$$\frac{\partial f}{\partial x} - \tilde{\lambda}\frac{\partial c}{\partial x} + \rho c(x)^T \frac{\partial c}{\partial x} = \frac{\partial f}{\partial x} - \big[\tilde{\lambda} - \rho c(x)\big]^T \frac{\partial c}{\partial x} = 0 \implies \tilde{\lambda} \leftarrow \tilde{\lambda} - \rho c(x) \text{ for active constraints.} \tag{19}$$

Algorithmically, this can be expressed as:

Augmented Lagrangian method

while Not Converged **do**
  $\min_x L_\rho(x, \tilde{\lambda})$ ▷ Minimize w.r.t x.
  $\tilde{\lambda} \leftarrow \tilde{\lambda} - \max(0, \tilde{\lambda} - \rho c(x))$ ▷ Update the multipliers, clamping to guarantee non-negativity.
  $\rho \leftarrow \alpha\rho$ ▷ Increase penalty. Typically, $\alpha \approx 10$

- This fixes the ill-conditioning that the penalty method suffers.
- It converges fast (super-linearly) to moderate precision.
- Works well on non-convex problems too!

Quadratic Program Example

Consider the problem:

$$\min_x \ \frac{1}{2}x^T Q x + q^T x \ \ , Q > 0 \text{ (Convex)} \tag{20}$$

$$s.t. \ \ Ax \leq b \tag{21}$$

$$Cx = d \tag{22}$$

Here have a quadratic objective, and linear constraints. This type of problem is very common and useful in control, and can be solved very fast online (in the order of KHz). Remember to check out the example in the lecture video!

4.3 Regularization and Duality

Consider the following problem:

$$\min_x f(x) \tag{23}$$

$$s.t. \ \ c(x) = 0 \tag{24}$$

We can express this as:

$$\min_x f(x) + P_\infty(c(x)) \tag{25}$$

$$P_\infty(x) = \begin{cases} 0, & x = 0 \\ +\infty, & x \neq 0 \end{cases} \tag{26}$$

Practically, this is terrible, but we can get the same effect by solving:

$$\min_x \max_\lambda f(x) + \lambda^T c(x) \tag{27}$$

Whenever $c(x) \neq 0$, the inner problem gives $+\infty$. Similarly for inequalities:

$$\min_x \ f(x) \tag{28}$$
$$s.t. \ \ c(x) \geq 0 \tag{29}$$
$$\implies \tag{30}$$
$$\min_x \ f(x) + P_\infty^+(c(x)) \tag{31}$$

Here, we have:

$$P_\infty^+(x) = \begin{cases} 0, & x \geq 0 \\ +\infty, & x < 0 \end{cases} \implies \tag{32}$$
$$\min_x \max_{\lambda \geq 0} f(x) - \lambda^T c(x) \tag{33}$$

where $f(x) - \lambda c(x)$ is our Lagrangian $L(x, \lambda)$.

For convex problems, one can switch the order of the min and max, and the solution does not change; this is the notion of "dual problems"! This isn't true in general, however, i.e. for non-convex problems.

The interpretation of this is that the KKT conditions define a saddle point in the space of (x, λ).

The KKT system should have $\dim(x)$ positive eigenvalues and $\dim(\lambda)$ negative eigenvalues at an optimum. Such a system is known as a "Quasi-definite" linear system.

4.3.1 Take-Away Messages:

When regularizing a KKT system, the lower-right block should be negative!

$$\begin{bmatrix} H + \alpha I & C^T \\ C & -\alpha I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x L \\ -c(x) \end{bmatrix}, \alpha > 0 \tag{34}$$

This makes the system a quasi-definite one.
Remember to check out the example of this in the lecture video. We observe that we still overshoot the solution, so we need a line search!

4.4 Merit Functions(for line search)

How do we a do a line search on a root-finding problem? Consider:

$$\text{find} \ \ x* \ \ s.t. \ c(x*) = 0 \tag{35}$$

First, we define a scalar "merit function" $P(x)$, that measure distance from a solution. A few standard choices for this merit function include:

$$P(x) = \frac{1}{2} c(x)^T c(x) = \frac{1}{2} ||c(x)||_2^2 \tag{36}$$
$$\text{or} \ \ P(x) = ||c(x)||_1 \tag{37}$$
$$\tag{38}$$

Note: We could use any norm. Now we could just do Armijo on $P(x)$:

Armijo Rule on $P(x)$

$\alpha = 1$ ▷ Step Length
while $p(x + \alpha \Delta x) > p(x) + b\alpha \nabla p(x)^T \Delta x$ **do** ▷ $\alpha \nabla p(x)^T \Delta x$ is the expected reduction from gradient
    $\alpha \leftarrow c\alpha$ ▷ c is a scalar < 1.
$x \leftarrow \alpha \Delta x + x$

4.4.1 Line search for Constrained Minimization

How about constrained minimization? Here, we want to come up with an option that specifies how much we are violating the constraint, as well as how far off the optimum we are / minimizing the objective funciton. Consider the problem:

$$\min_x f(x) \tag{39}$$
$$s.t. \quad c(x) \leq 0 \tag{40}$$
$$d(x) = 0 \tag{41}$$
$$\implies \tag{42}$$
$$L(x, \lambda, \mu) = f(x) + \lambda^T c(x) + \mu^T d(x) \tag{43}$$

We have lots of options for merit functions. One option is:

$$P(x, \lambda, \mu) = \frac{1}{2}||\nabla L(x, \lambda, \mu)||_2^2 \tag{44}$$
$$\tag{45}$$

Here, the term $\nabla L(x, \lambda, \mu)$ is the KKT residual:

$$\begin{bmatrix} \nabla_x L(x, \lambda, \mu) \\ \min(0, c(x)) \\ d(x) \end{bmatrix} \tag{46}$$

However, this isn't the best option to use, because evaluating the gradient of the KKT condition is as expensive as the newton step solve itself.
Another option is:

$$P(x, \lambda, \mu) = f(x) + \rho \; || \begin{bmatrix} \min(0, c(x)) \\ d(x) \end{bmatrix} ||_1 \tag{47}$$

Here, ρ is a scalar trade off between the objective minimization and constraint satisfaction. Also remember that any norm works here (in place of the 1 norm depicted), but using the 1 norm is the most common. This option gives us flexibility, because we can pick trade off ρ - initially we can set this to be low, to drive us close to the optimum solution, and when we are close to the optimum, we can increase ρ to ensure we satisfy the constraints.

Yet another option is:

$$P(x, \lambda, \mu) = f(x) - \tilde{\lambda}^T c(x) + \tilde{\mu}^T d(x) + \frac{\rho}{2}|| \min(0, c(x)||_2^2 + \frac{\rho}{2}||d(x)||_2^2 \tag{48}$$

which is the augmented Lagrangian itself.
Remember to check out the example in the lecture video!

4.4.2 Take-Away Messages (from the example)

- $P(x)$ based on the KKT residual is expensive.
- Excessively large penalty weights can cause problems.
- Augmented Lagrangian methods come with a merit function for free. So if we're using the Augmented Lagrangian to solve the problem, just use this as a merit function.

References

[1] J. Nocedal et al. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006. ISBN: 9780387303031. URL: https://books.google.com/books?id=eNlPAAAAMAAJ.

[2] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005. ISBN: 9780471649908. URL: https://books.google.com/books?id=A0OXDwAAQBAJ.

[3] S.H. Strogatz and M. Dichter. *Nonlinear Dynamics and Chaos, 2nd ed. SET with Student Solutions Manual*. Studies in Nonlinearity. Avalon Publishing, 2016. ISBN: 9780813350844. URL: https://books.google.com/books?id=vUWhDAEACAAJ.

5 Deterministic Optimal Control

In this chapter, we cover Deterministic optimal control, Pontryagin's Minimum Principle, and Linear-Quadratic Regulator(LQR)

Let's consider the following control problem:

$$\min_{x(t),u(t)} J(x(t),u(t)) = \int_{t_0}^{t_f} L(x(t),u(t))dt + L_F(x(t_f)) \tag{1}$$

$$s.t.\ \dot{x}(t) = f(x(t),u(t)) \tag{2}$$

$$\text{Any other constraints} \tag{3}$$

Here, we minimize across "state" or "input" trajectories. J represents our cost function, $L(x(t),u(t))$ represents our "stage cost", $L_F(x(t_f))$ represents a "terminal cost", $\dot{x}(t) = f(x(t),u(t))$ are the dynamics constraint.

This is an "infinite dimensional" problem in the sense that an infinite amount of discrete-time control points is required to fully specify the control to be applied.

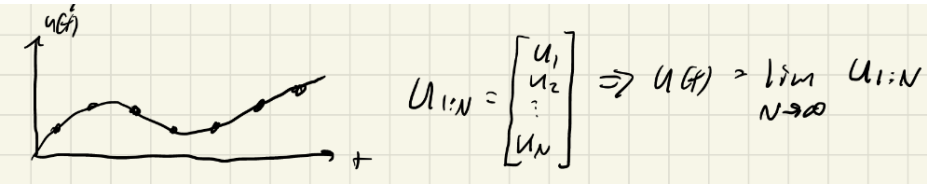

Figure 16: Deterministic optimal control problem

- The solutions to this problem are open-loop trajectories.
- Now, there are a few control problems with analytic solutions in continuous time, but not many.
- We focus on the discrete-time setting, where we have tractable algorithms.

5.1 Discrete Time

Consider a discrete-time version of this problem.

$$\min_{x_{1:N},u_{1:N-1}} J(x_{1:N},u_{1:N-1}) = \sum_{k=1}^{N-1} L(x_k,u_k) + L_F(x_N) \tag{4}$$

$$s.t.\ x_{n+1} = f(x_n,u_n) \tag{5}$$

$$u_{\min} \le u_k \le u_{\max} \quad \text{Torque limits.} \tag{6}$$

$$c(x_k) \le 0\ \forall k \quad \text{Obstacle / collision constraints.} \tag{7}$$

- This version of the problem is now a finite dimensional problem.
- Samples x_k, u_k are often called "knot points".
- We can convert continuous systems to discrete-time problems using integration methods such as the Runge-Kutta method etc.
- Finally, we can convert back from discrete-time problems to continuous problems using interpolation.

5.2 Pontryagin's Minimum Principle

- The Pontryagin's Minimum Principle is also a "maximum principle" if instead of minimizing a cost function, we are maximizing a reward function.
- Essentially provides first order necessary conditions of deterministic optimal control problems.

- In discrete time, it's just a special case of the KKT conditions.

Consider the problem we have before:

$$\min_{x_{1:N}, u_{1:N-1}} J(x_{1:N}, u_{1:N-1}) = \sum_{k=1}^{N-1} l(x_k, u_k) + l_F(x_N) \tag{8}$$

$$s.t.\ x_{n+1} = f(x_n, u_n) \tag{9}$$

In this setting, we will consider torque limits applied to the problem, but it's hard to handle constraints on state (such as collision constraints, like we had before).
We can form the Lagrangian of this problem as follows:

$$L = \sum_{k=1}^{N-1} \Big[l(x_k, u_k) + \lambda_{k+1}^T (f(x_k, u_k) - x_{k+1}) \Big] + l_F(x_N) \tag{10}$$

$$\tag{11}$$

This result is usually stated in terms of the "Hamiltonian":

$$H(x, u, \lambda) = l(x, u) + \lambda^T f(x, u) \tag{12}$$

Plugging in H into the Lagrangian L:

$$L = H(x_1, u_1, \lambda_2) + \Big[\sum_{k=2}^{N-1} H(x_k, u_k, \lambda_{k+1}) - \lambda_k^T x_k \Big] + l_F(x_N) - \lambda_N^T x_N \tag{13}$$

Note the change in indexing of the summation (check it by yourself).
If we take derivatives with respect to x and λ:

$$\frac{\partial L}{\partial \lambda_k} = \frac{\partial H}{\partial \lambda_k} - x_{k+1} = f(x_k, u_k) - x_{k+1} = 0 \tag{14}$$

$$\frac{\partial L}{\partial x_k} = \frac{\partial H}{\partial x_k} - \lambda_k^T = \frac{\partial l}{\partial x_k} + \lambda_{k+1}^T \frac{\partial f}{\partial x_k} - \lambda_k^T = 0 \tag{15}$$

$$\frac{\partial L}{\partial x_N} = \frac{\partial l_F}{\partial x_N} - \lambda_N^T = 0 \tag{16}$$

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

$$u_k = \arg\min_{\tilde{u}} H(x_k, \tilde{u}, \lambda_{k+1}) \tag{17}$$

$$s.t.\ \tilde{u} \in \mathcal{U} \tag{18}$$

Here, $\tilde{u} \in \mathcal{U}$ is shorthand for "in feasible set", for example, $u_{\min} \leq \tilde{u} \leq u_{\max}$.

In summary, we have:

$$x_{k+1} = \nabla_\lambda H(x_k, u_k, \lambda_{k+1}) = f(x_k, u_k) \tag{19}$$

$$\lambda_k = \nabla_x H(x_k, u_k, \lambda_{k+1}) = \nabla_x l(x_k, u_k) + (\frac{\partial f}{\partial x})^T \lambda_{k+1} \tag{20}$$

$$u_k = \arg\min_{\tilde{u}} H(x_k, \tilde{u}, \lambda_{k+1}) \tag{21}$$

$$s.t.\ \tilde{u} \in \mathcal{U} \tag{22}$$

$$\lambda_N = \frac{\partial l_F}{\partial X_N} \tag{23}$$

Now these can be stated almost identically in continuous time:

$$\dot{x} = \nabla_\lambda H(x, u, \lambda) = f_{\text{continuous}}(x, u) \tag{24}$$

$$\dot{\lambda} = \nabla_x H(x, u, \lambda) = \nabla_x l(x, u) + (\frac{\partial f_{\text{continuous}}}{\partial x})^T \lambda \tag{25}$$

$$u = \arg\min_{\tilde{u}} H(x, \tilde{u}, \lambda) \tag{26}$$

$$\ni \ \tilde{u} \in \mathcal{U} \tag{27}$$

$$\lambda_N = \frac{\partial l_F}{\partial X_N} \tag{28}$$

5.3 Notes

- Historically, many algorithms were based on forward / backward integration of the continuous ODEs for $x(t), \lambda(t)$ for performing gradient descent on $u(t)$.
- These are called "indirect" and / or "shooting" methods.
- In continuous time, $\lambda(t)$ is called "co-state" trajectory.
- These methods have largely fallen out of favor as computers and solvers have improved.

6 Linear Quadratic Regulator (LQR)

A very common class of controllers is the Linear Quadratic Regulator. In this setting, we have a quadratic cost, and linear dynamics, specified as:

$$\min_{x_{1:N}, u_{1:N-1}} \sum_{k=1}^{N-1} \left[\frac{1}{2} x_k^T Q x_k + \frac{1}{2} u_k^T R_k u_k\right] + \frac{1}{2} x_N^T Q_N x_N \tag{29}$$

$$s.t.\ x_{k+1} = A_k x_k + B_k u_k \tag{30}$$

Here, we have: $Q \geq 0$ is ok, because 0 means control cost is 0., $R > 0$, because if R=0, u can be infinite, but u has the boundary.

- The goal of this problem is to drive the system to the origin.
- It's considered a "time-invariant" LQR if $A_k = A, B_k = B, Q_k = Q, R_k = R\ \forall\ t$, and is a "time-varying" LQR (TVLQR) otherwise.
- We typically use time-invariant LQRs for stabilizing an equilibrium, and TVLQR for tracking trajectories.
- Can (locally) approximate many non-linear problems, and are thus very commonly used.
- There are also many extensions of LQR, including the infinite horizon case, and stochastic LQR.
- It's been called the "crown jewel of control theory."

6.1 LQR with Indirect Shooting (bad)

As the hamiltonian theory above, we can directly write:

$$x_{k+1} = \Delta_\lambda H(x_k, u_k, \lambda_{k+1}) = Ax_k + Bu_k \tag{31}$$

$$\lambda_k = \Delta_x H(x_k, u_k, \lambda_{k+1}) = Qx_k + A^T \lambda_{k+1},\ \lambda_N = Q_N x_N \tag{32}$$

$$u_k = \arg\min_{\tilde{u}} H(x, \tilde{u}, \lambda) = -R^{-1} B^T \lambda_{k+1} \text{ This gives the gradients of the cost w.r.t.} u_k \tag{33}$$

The procedure for LQR with indirect shooting is:

1. Start with an initial guess $u_{1:N-1}$ and x_0, to rollout all the $x_{1:N}$ using the dynamic function.
2. Backward pass: to get λ_k using: $\lambda_k = Qx_k + A^T \lambda_{k+1},\ \lambda_N = Qx_N$ and Δu_k using $\Delta u_k = (u_k - u_{old})$.
3. Forward pass: Rollout the new state $x_{1:N}$ using the new input $u_{new} = u_{old} + a. * \Delta u$ with line search technique.
4. Go to (2) until Δu convergence.

6.1.1 Example

Check out the example of the double integrator in the lecture. Here, we have -

$$\dot{x} = \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} q \\ \dot{q} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{34}$$

Think of this as a "sliding brick" on ice, without friction. Here, the discrete-time version of this system is -

$$x_{k+1} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \begin{bmatrix} q_k \\ \dot{q}_k \end{bmatrix} + \begin{bmatrix} \frac{1}{2}h^2 \\ h \end{bmatrix} u_k \tag{35}$$

where $\begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix}$ represents the A matrix, and $\begin{bmatrix} \frac{1}{2}h^2 \\ h \end{bmatrix}$ represents the B matrix, h is the time. Note: the detailed processes are shown in the Jupyter code.

6.2 LQR as a QP (good)

Let us assume the initial state x_1 is given (and is not a decision variable). Define z:

$$z = \begin{bmatrix} u_1 \\ x_2 \\ u_2 \\ . \\ . \\ x_N \end{bmatrix} \tag{36}$$

Also define H:

$$H = \begin{bmatrix} R_1 & 0 & ... & 0 \\ 0 & Q_2 & ... & 0 \\ & & . & \\ 0 & 0 & ... & Q_N \end{bmatrix} \tag{37}$$

such that $J = \frac{1}{2} z^T H z$. We also define C and d:

$$C = \begin{bmatrix} B_1 & (-I) & ... & ... & ... & 0 \\ 0 & A & B & (-I) & ... & 0 \\ & & . & & & \\ 0 & 0 & ... & A_{N-1} & B_{N-1} & (-I) \end{bmatrix} \tag{38}$$

$$d = \begin{bmatrix} -A_1 x_1 \\ 0 \\ . \\ 0 \end{bmatrix} \tag{39}$$

such that $Cz = d$. We now have the LQR problem can be written as a standard QP:

$$\min_z \frac{1}{2} z^T H z \tag{40}$$

$$s.t. \quad Cz = d \tag{41}$$

The Lagrangian of this QP is:

$$L(z, \lambda) = \frac{1}{2} z^T H z + \lambda^T \big[Cz - d \big] \tag{42}$$

and the KKT conditions are:

$$\nabla_z L = Hz + C^T \lambda = 0 \tag{43}$$

$$\nabla_\lambda L = Cz - d = 0 \tag{44}$$

This may be written as:

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ d \end{bmatrix} \tag{45}$$

We get the exact answer by solving this one linear system!

6.2.1 Example

Remember to check out the example on the double integrator / sliding brick on ice. We can compare shooting to the QP solution of the LQR problem. Much better than the indirect/shooting method!!!

6.2.2 Riccati recursion: A closer look at the LQR QP

The QP KKT system is very sparse (i.e. lots of zeros in the constituent matrices), and has a lot of structure:

$$\begin{bmatrix} R & & & & & & . & B^T & & \\ & Q & & & & & . & -I & A^T & \\ & & R & & & & . & & B^T & \\ & & & Q & & & . & & -I & A^T \\ & & & & R & & . & & & B^T \\ & & & & & Q_N & . & & & -I \\ . & . & . & . & . & . & . & . & . & . \\ B & -I & & & & & . & 0 & 0 & 0 \\ & A & B & -I & & & . & 0 & 0 & 0 \\ & & & A & B & -I & . & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ x_2 \\ u_2 \\ x_3 \\ u_3 \\ x_4 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -Ax_1 \\ 0 \\ 0 \end{bmatrix} \tag{46}$$

- Consider the last line of upper block of this system (i.e. above the dotted line):

$$Q_N x_4 - \lambda_4 = 0 \tag{47}$$

$$\implies \lambda_4 = Q_N x_4 \tag{48}$$

Now consider the second last line of the upper block of this system:

$$Ru_3 + B^T \lambda_4 = Ru_3 + B^T Q_N x_4 = 0 \tag{49}$$

Plugging the dynamics into this, we have:

$$Ru_3 + B^T Q_N (Ax_3 + Bu_3) = 0 \tag{50}$$

$$\implies u_3 = -(R + B^T Q_N B)^{-1} B^T Q_N A x_3 \tag{51}$$

We can define $K_3 = (R + B^T Q_N B)^{-1} B^T Q_N A$ in the above equation.
Finally, consider the third line of this system.

$$Qx_3 - \lambda_3 + A^T \lambda_4 = 0 \tag{52}$$

We can then manipulate this by first substituting λ_4, then plugging in the dynamics for x_4, and finally using the feedback law:

$$Qx_3 - \lambda_3 + A^T Q_N x_4 = 0 \tag{53}$$

$$\implies Qx_3 - \lambda_3 + A^T Q_N (Ax_3 + Bu_3) = 0 \tag{54}$$

$$\implies Qx_3 - \lambda_3 + A^T Q_N (A - BK) x_3 = 0 \tag{55}$$

$$\implies \lambda_3 = (Q + A^T Q_N (A - BK)) x_3 \tag{56}$$

We can then define $P_3 = (Q + A^T Q_N (A - BK))$.
We now have a recursion for K and P:

$$P_N = Q_N \tag{57}$$

$$K_n = (R + B^T P_{n+1} B)^{-1} B^T P_{n+1} A \tag{58}$$

$$P_n = Q + A^T P_{n+1} (A - BK_n) \tag{59}$$

- This is called a Riccati equation / recursion.
- Thanks to the sparse structure in QP, we can solve the QP by doing a backward Riccati recursion followed by a forward rollout starting from x_1.
- Riccati recursion has complexity $\mathcal{O}(N(n+m)^3$ instead of $\mathcal{O}(N^3(n+m)^3$ for the naive QP solution. This carries over to the general non-linear case.
- Even more importantly, we now have a feedback policy u = -kx instead of an open loop trajectory to execute.

6.2.3 Example

Remember to check out the lecture video for an example on LQR with Riccati open-loop, and closed loop with noise.

6.3 Infinite Horizon LQR

Let's now consider the infinite horizon case.

- For the time-invariant LQR, K matrices converge to constant values over the infinite horizon.
- For stabilization problems we almost always use the constant K.
- Backward recursion for P, we can slove it using the fixed point iteration method or the root-finding method and We can solve for this explicitly in Julia / Matlab / Python dare function does this for you.

6.3.1 finite horizon LQR

$$\min_{x_{1:N}, u_{1:N-1}} \sum_{k=1}^{N-1} \left[\frac{1}{2} x_k^T Q x_k + \frac{1}{2} u_k^T R_k u_k\right] + \frac{1}{2} x_N^T Q_N x_N \tag{60}$$

$$s.t.\ x_{k+1} = A_k x_k + B_k u_k \tag{61}$$

Here, A_k and B_k can be time-variant.

6.3.2 infinite horizon LQR

$$\min_{x_{1:\infty}, u_{1:\infty-1}} \sum_{k=1}^{\infty-1} \left[\frac{1}{2} x_k^T Q x_k + \frac{1}{2} u_k^T R_k u_k\right] \tag{62}$$

$$s.t.\ x_{k+1} = A x_k + B u_k \tag{63}$$

Here, A and B are time-invariant.

6.4 How to solve the policy

Two Solutions:

- $\pi(x)$ = slove QP problem (too slow)
- $\pi(x) = -K_k x$, for IHLQR: it seems that we only have to compute the feedback gain K for only once, and use it to generate all the U_k.

How we should do if we want to drive our robot to any other goal? If we know the goal:

$$x_{goal} = A x_{goal} + B u_{goal} \tag{64}$$

$$\tilde{x} = x - x_g, \tilde{U} = U - U_g \tag{65}$$

Then we have:

$$\tilde{x_{k+1}} = x_{k+1} - x_g = A x_k + B U_k - (A x_{goal} + B u_{goal}) \tag{66}$$

$$\tilde{x_{k+1}} = A(x_k - x_g) + B(U - U_g) \tag{67}$$

Then solve the LQR and get the feedback gain:

$$K = dLQR(A, B, Q, R) \tag{68}$$

$$\tilde{U} = -K\tilde{x} \tag{69}$$

$$U - U_g = -K(x - x_g) \tag{70}$$

$$U = -K(x - x_g) + U_g \tag{71}$$

So it can drive the robot to any goal, and if the A, B, Q, R is time-invariant, K is fixed. If A and B are time-variant, then we should calculate the K each time step (This can be done offline).

6.4.1 Extension to the nonlinear

Taylor expansion in a fixed point. let assume $\bar{x} = f(\bar{x}, \bar{u})$, then we can expand the dynamic function in the fixed point $\bar{x}, \bar{u}$:

$$\begin{aligned} x_{k+1} &\approx f(\bar{x}, \bar{u}) + \left[\frac{\partial f}{\partial x}\Big|_{\bar{x},\bar{u}} \right] (x - \bar{x}) + \left[\frac{\partial f}{\partial u}\Big|_{\bar{x},\bar{u}} \right] (u - \bar{u}) \\ \bar{x} + \Delta x_{k+1} &= f(\bar{x}, \bar{u}) + A\Delta x + B\Delta u \\ \Delta x_{k+1} &= \bar{A}\Delta x + \bar{B}\Delta u \end{aligned} \tag{72}$$

6.5 Controllability

How do we know if LQR will work? For the time-invariant case, there is a simple answer. For any initial state x_0, x_n is given by:

$$x_n = Ax_{n-1} + Bu_{n-1} \tag{73}$$
$$= A(Ax_{n-2} + Bu_{n-2}) + Bu_{n-1} \tag{74}$$
$$= A^n x_0 + A_{n-1}Bu_0 + ... + Bu_{n-1} \tag{75}$$
$$= \begin{bmatrix} B & AB & A^2B & ... & A^{n-1}B \end{bmatrix} \begin{bmatrix} u_{n-1} \\ u_{n-2} \\ . \\ . \\ u_0 \end{bmatrix} + A^n x_0 \tag{76}$$

Our goal is the set the $x_n = 0$. Here, we may define a controllability matrix C as:

$$C = \begin{bmatrix} B & AB & A^2B & ... & A^{n-1}B \end{bmatrix} \tag{77}$$

In order to drive any x_0 to any desired x_n, the controllability matrix must have full row rank:

$$rank(C) = n \tag{78}$$
$$\ni n = dim(x) \tag{79}$$

This is equivalent to solving the following least-squares problem for $u_{0:n-1}$:

$$\begin{bmatrix} u_{n-1} \\ u_{n-2} \\ . \\ . \\ u_0 \end{bmatrix} = \begin{bmatrix} C^T(CC^T)^{-1} \end{bmatrix} \begin{bmatrix} (x_n - A^n x_0) \end{bmatrix} \tag{80}$$

Here, $\begin{bmatrix} C^T(CC^T)^{-1} \end{bmatrix}$ is the pseudo-inverse of controllability C, and requires CC^T to be invertible. We can stop at n time steps because the Cayley-Hamilton theorem says that A^n can be written as a linear combination of smaller powers of A:

$$A^n = \sum_{k=0}^{n-1} \alpha_k A^k \tag{81}$$

for some α_k. Therefore adding more timesteps / columns to C can't increase the rank of C. In the time varying case, QP is better than the method above.

7 Dynamic Programming

7.1 Bellman's Principle

- Optimal control problems have an inherently sequential structure.
- Past control inputs affect future states but future control inputs cannot affect past states.
- Bellman's principle (i.e. The principle of optimality), states the cocnsequences of this for optimal trajectories.
- Sub-trajectories of optimal trajectories have to be optimal for the appropriately defined sub-problem.

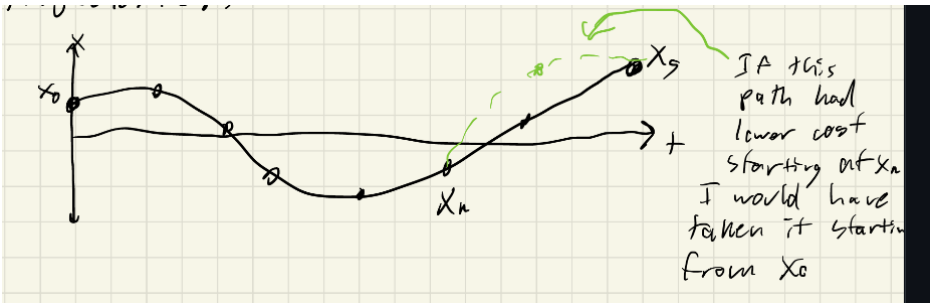

Figure 17: Bellman's Principle

7.2 Dynamic Programming

- Bellman's Principle suggests starting from the end of the trajectory and working backwards.
- We've already seen hints of this in the Riccati equation, and in the co-state / multiplier equation from Pontryagin.
- Define "optimal cost-to-go" a.k.a "value function" $V_N(x)$.
- Encodes the cost incurred starting from state x at time k if we act optimally.
- For the LQR setting, we have:

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

- Now we can back up one time step, and compute $V_{N-1}(x)$:

$$\min_u \frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u^T R u + V_N(A_{N-1}x_{N-1} + B_{N-1}u) \tag{2}$$

$$= \min_u \frac{1}{2}u^T R u + \frac{1}{2}(Ax_{N-1} + B_{N-1}u)^T Q_N (Ax_{N-1} + B_{N-1}u) \tag{3}$$

Since for the optimal action we have ∇_u of this cost $= 0$, we have:

$$u^T R_{N-1} + (Ax_{N-1} + B_{N-1}u)^T Q B_{N-1} = 0 \tag{4}$$

$$\implies u_{N-1} = -(R_{N-1} + B_{N-1}^T Q_N B_{N-1})^{-1} B_{N-1}^T Q_N A_{N-1} x_{N-1} \tag{5}$$

We may define K_{N-1} as $(R_{N-1} + B_{N-1}^T Q_N B_{N-1})^{-1} B_{N-1}^T Q_N A_{N-1}$ for convenience.

- Plugging in $u = -Kx$ into our expression for $V_{N-1}(x)$, we have:

$$V_{N-1}(x) = \frac{1}{2}x^T (Q_{N-1} + K_{N-1}^T R_{N-1} K + (A_{N-1} - B_{N-1}K_{N-1})^T Q_N (A_{N-1} - B_{N-1}K_{N-1}))x \tag{6}$$

We can define $P_{N-1} = (Q_{N-1} + K_{N-1}^T R_{N-1} K + (A_{N-1} - B_{N-1}K_{N-1})^T Q_N (A_{N-1} - B_{N-1}K_{N-1}))$, such that:

$$V_{N-1}(x) = \frac{1}{2}x^T P_{N-1} x \tag{7}$$

- We now have a recursion in K and P that we can iterate until $k = 1$. This is just the Riccati equation again. The k is same as the Riccatic we said above, and P is different from that. In terms of the numerical calculation, this P is symmetry and better.

7.2.1 Dynamic Programming Algorithm

Backward DP Algorithm

$V_N(x) \leftarrow l_N(x)$
$K \leftarrow N$
while $K > 1$ **do**
    $V_{k-1} = \min_u \left[l(x,u) + V_k(f(x,u)) \right]$ ▷ The Bellman Equation.
    $k \leftarrow k - 1$

- If we know $V_k(x)$, the optimal feedback policy is:

$$u_k(x) = \arg\min_u \left[l(x_k, u) + V_{k+1}(f(x_k, u)) \right] \tag{8}$$

- DP equations can be written equivalently in terms of action-value or Q functions:

$$S_k(x,u) = l(x,u) + V_{k+1}(f(x,u)) \tag{9}$$
$$\implies u_k(x_k) = \arg\min_u S_k(x_k, u) \tag{10}$$

- These are usually denoted $Q(x,u)$, here we will use S, since Q is used in the LQR state cost as well.

7.2.2 The Curse

- DP is sufficient for a global optimum.
- It is only tractable for simple problems, such as LQR problems or low-dimensional problems.
- $V(x)$ stays quadratic for LQR problems, but becomes impossible to write down analytically for even simple non-linear problems.
- Even if we could, the $\min_u S(x,u)$ will be non-convex, and possibly hard to solve on its own.
- The cost of DP blows up with state dimension, due to the difficulty of representing $V(x)$.

7.2.3 Why do we care?

- Approximate DP, where $V(x)$, or $S(x,u)$ are reprsented with function approximators can work well.
- Forms the basis of a lot of modern Reinforcement Learning.
- DP generalizes to stochastic problems well, (just wrap everything in expectation operators), whereas Pontryagin's does not.

7.2.4 What are those Lagrange Multipliers?

- Recall Riccati derivation from QPs:

$$\lambda_n = P_n x_n \tag{11}$$
$$P_n = Q + A^T P_{n+1}(A - BK) \tag{12}$$
$$= Q + K^T R K + (A - BK)^T P_{n+1}(A - BK) \tag{13}$$
$$V_n(x) = \frac{1}{2} x^T P_n x \tag{14}$$
$$\implies \lambda_n = \nabla_x V_n(x) \tag{15}$$

- The dynamics multipliers are the cost-to-go gradients!
- Carries over to the general non-linear setting, beyond just LQR.

7.3 Example

Remember to check out the lecture video for an example, where λ_n from the QP matches $\nabla_x V_n(x)$ from DP.

8 Convex Model-Predictive Control

- LQR is very powerful, but often need to explicitly reason about constraints.
- Often these are simple (ex. torque limits), and can be encoded as a convex set.
- Constraints break the Riccati solution, but we can still solve the QP online[Riccati can solve the K offline and apply: u = -kx].
- Convex MPC has gotten extremely popular as computers have gotten faster.

8.1 Background: Convexity

- Convex Set: A line connecting any two points in the set is contained within the set.
- Standard examples:
  - Linear subspaces ($Ax = b$).
  - Half-spaces, boxes, and polytopes ($Ax \leq b$).
  - Ellipsoids ($x^T P x \leq 1$).
  - Cones ($||x_{2:n}||_2 \leq x_1$). This is called the second order cone, it's the usual ice-cream cone you're used to.
- Convex functions: A function $f(x) : R^n \longrightarrow \mathbb{R}$ who's epigraph is a convex set.
- Examples:
  - Linear $f(x) = c^T x$.
  - Quadratic $f(x) = \frac{1}{2} x^T Q x + q^T x, \ni Q \geq 0$.
  - Norms $f(x) = |x|$.
- Convex Optimization Problem: Minimize 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 (QCQP): Quadratic $f(x)$, ellipsoid $c(x)$.
  - Second-order cone program (SOCP): Linear $f(x)$, cone $c(x)$.
- Convex problems don't have spurious local optima that satisfy the KKT. If you find a local KKT solution, you have the global optimum!
- Practically, Newton's method converges really fast and reliably (5-10 iterations at maximum).
- Can bound the solution time for real-time control.

8.2 Convex MPC

- more likely to the "constraint LQR"
- Remember from DP, if we have a cost-to-go function, we can get the control u by solving a one-step problem (a.k.a. we can solve all the k in one time, and so that we can solve it offline)

$$\min_{x_{1:N}, u_{1:N-1}} \sum_{k=1}^{N-1} \left[\frac{1}{2} x_k^T Q x_k + \frac{1}{2} u_k^T R_k u_k \right] + \frac{1}{2} x_N^T Q_N x_N \tag{1}$$

$$s.t.\ x_{k+1} = A_k x_k + B_k u_k \tag{2}$$

- The problem of LQR is that we can not handle the constraints, we can add the constraints on control input u to this one-step problem, but this will perform poorly because V(x) was computed without constraints

- So MPC is the ideas of adding more steps to the one-step problem, which is needed compute online.

$$\min_{x_{1:H},u_{1:H-1}} \sum_{k=1}^{H-1} \left[\frac{1}{2}x_k^T Q x_k + \frac{1}{2}u_k^T R_k u_k\right] + \frac{1}{2}x_H^T Q_H x_H \tag{3}$$

$$s.t.\ x_{k+1} = A_k x_k + B_k u_k \tag{4}$$

- Here, H $\leq$ N is called "horizon"
- with no additional constraints, MPC("receding horizon") exactly matches LQR for any time
- Intuition: explicit constrained optimization over the first H steps gets the state close enough to the reference that the constraints are no longer active and LQR solution/cost-to-go is valid further into the future.

In general, a good approximation of V(x) is important for good performance, if we have better V(x), we only need a shorter horizon, for less reliance on V(x), we need a longer horizon.

8.3 Planar Quadrotor example

- **Results**: for some non-linear problem, MPC is better than LQR since MPC can apply the constraints to stable the systems.
- **Tricks**: When we linear a non-linear problem in MPC, we can add some constraints to make sure that the variables are in the safe region so that the linear approximation works well.

For all details of this example, please see the codes.

8.4 Convex MPC examples

8.4.1 Rocket Landing

Thrust vector constraints are conic (the limit of the fire direction)

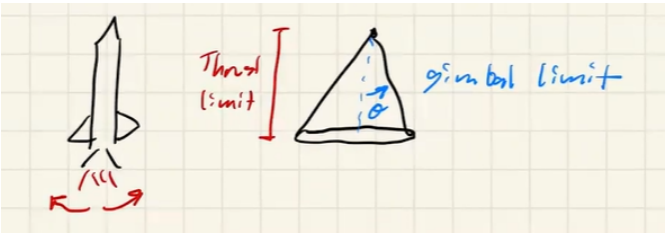

Figure 18: Rocket landing

SpaceX + JPL Mars landing use SOCP-based MPC with linearized dynamics.

8.4.2 Legged Robots

contact forces must obey the "friction cone" constraint

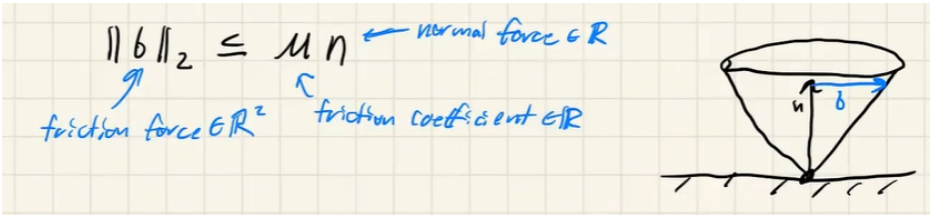

Figure 19: Friction cone

Friction Cone is often approximated as a pyramid so the constraint is linear ($Ax \leq \rho$)

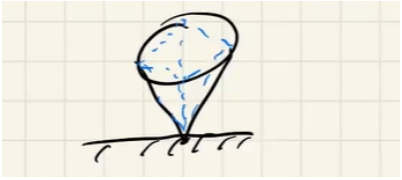

Figure 20: Friction cone

MIT cheetah and other quadrupeds use QP-based MPC with linearized dynamics.
What about Nonlinear Dynamics? Here are several ideas

- Linear stuff often works well, so use it if you can
- Nonlinear dynamic make the MPC problem non-convex, no convergence guarantees.
- can work well in practice with some efforts (like autonomous vehicle).

9 Nonlinear Trajectory Optimization problem

$$min_{x_{1:N},u_{1:N-1}} J = \sum_{k=1}^{N-1} l_k(x_k, u_k) + l_N(x_N) \tag{5}$$

$$\begin{aligned} \text{s.t.} \quad & x_{k+1} = f\left(x_k, u_k\right) \\ & x_k \in X_k \\ & u_k \in U_k \end{aligned} \tag{6}$$

Where the cost is non-convex and the dynamic model is nonlinear dynamics, constraints are non-convex constraints. Usally assume costs and constraints are C^2 continuous.

9.1 Differential Dynamic Programming(DDP/iLQR)

- Nonlinear trajectory optimization method based on approximate DP
- Use 2^{nd}-order Taylor approximations of cost-to-go(V(x) fuanction) in DP and compute Newton steps, while in RL, usually use the neural network to fit the value function.
- Very fast convergence possible

9.1.1 Solving the DDP/iLQR

Taylor expansion of the cost-to-go function $V_k(x)$:

$$\begin{aligned} V_k(x + \Delta x) \approx V_k(x) + p_k^T \Delta x + \tfrac{1}{2} \Delta x^T P_k \Delta x \\ p_N = \nabla_x l_N(x), P_N = \nabla_{xx}^2 l_N(x) \end{aligned} \tag{7}$$

Also, the Taylor expansion of Action-Value function $S_k(x_k, u_k) = l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k))$ is:

$$\begin{aligned} & S_k(x + \Delta x, u + \Delta u) \approx S_k(x, u) + \begin{bmatrix} g_x \\ g_u \end{bmatrix}^T \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix} \\ & + \tfrac{1}{2} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}^T \begin{bmatrix} G_{xx} & G_{xu} \\ G_{ux} & G_{uu} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}, \left(G_{xu} = G_{ux}^T\right) \end{aligned} \tag{8}$$

Why do we have the Action-value function(the Q-function in RL)? Because the Q-function is the DP equation during the backward pass.

According to the Bellman's principle, we can compute backward: from V_N to V_{N-1}:

$$V_{N-1}(x+\Delta x) := \min_{\Delta u}\Big[S(x,u) + g_x^T\Delta x + g_u^T\Delta u + \frac{1}{2}\Delta x^T G_{xx}\Delta x + \frac{1}{2}\Delta u^T G_{uu}\Delta u + \frac{1}{2}\Delta x^T G_{xu}\Delta u + \frac{1}{2}\Delta u^T G_{ux}\Delta x\Big] \tag{9}$$

take the derivative of control input Δu to compute the optimal control input:

$$\begin{aligned}\nabla_u V_{N-1}(x+\Delta x) &= g_u + G_{uu}\Delta u + G_{ux}\Delta x = 0\\ \Rightarrow \Delta u_{N-1} &= -G_{uu}^{-1}g_u - G_{uu}^{-1}G_{ux}\Delta x = -d_{N-1} - K_{N-1}\Delta x\end{aligned} \tag{10}$$

Well, as we can see, besides the feedback term K, it also had a feedforward term d. This is because the original LQR problem is solving the solution in the original point, while now we would like to solve this problem in any initial states. Just like the process of solving the LQR using the Ricatti equation, here we can also insert the optimal input term into the V_{N-1} again and get the new V(x), which is used in iteration:

$$\begin{aligned}V_{N-1}(x+\Delta x) = V_{N-1}(x) &+ g_x^T\Delta x + g_u^T\left(-d_{N-1} - K_{N-1}\Delta x\right) +\\ &\frac{1}{2}\Delta x^T G_{xx}\Delta x + \frac{1}{2}\left(d_{N-1} + K_{N-1}\Delta x\right)^T G_{uu}\left(d_{N-1} + K_{N-1}\Delta x\right) -\\ &\frac{1}{2}\Delta x^T G_{xu}\left(d_{N-1} + K_{N-1}\Delta x\right) - \frac{1}{2}\left(d_{N-1} + K_{N-1}\Delta x\right)^T G_{ux}\Delta x\end{aligned} \tag{11}$$

After arrangement properly, we have the same structure as the "Ricatti" equation:

$$\begin{aligned}P_{N-1} &= G_{xx} + K_{N-1}^T G_{uu}K_{N-1} - G_{xu}K_{N-1} - K_{N-1}^T G_{ux}\\ p_{N-1} &= g_x - K_{N-1}^T g_u + K_{N-1}^T G_{uu}d_{N-1} - G_{xu}d_{N-1}\end{aligned} \tag{12}$$

For the details of the algorithm, check out the Julia code!

9.2 Some math tricks

Actually, we need more math to calculate the g and G above

for the matrix calculs: given f(x) $R^n \Rightarrow R^m$, look at 2^{nd} order Taylor expansion:

if $m = 1$

$$\begin{aligned}f(x+\Delta x) \approx & f(x) + \frac{\partial f}{\partial x}\Delta x + \frac{1}{2}\Delta x^T\frac{\partial^2 f}{\partial x^2}\Delta x\\ &\frac{\partial f}{\partial x} \in \mathbb{R}^{1\times n}\\ &\frac{\partial^2 f}{\partial x^2} \in \mathbb{R}^{n\times n}\end{aligned} \tag{13}$$

if $m > 1$

$$\begin{aligned}f(x+\Delta x) \approx & f(x) + \frac{\partial f}{\partial x}\Delta x + \frac{1}{2}\left(\frac{\partial}{\partial x}\left[\frac{\partial f}{\partial x}\Delta x\right]\right)\Delta x\\ &\frac{\partial f}{\partial x} \in \mathbb{R}^{m\times n}\end{aligned} \tag{14}$$

for m>1, the $\frac{\partial^2 f}{\partial x^2}$ is a 3^{rd}-rank tensor. Think of this as a "3D matrix". We need some tricks to keep track of which dimensions we are multiplying along.

- Kronnecker product

$$\begin{aligned}A\otimes B &= \begin{bmatrix} a_{11}B & a_{12}B & \dots\\ a_{21}B & a_{22}B & \dots\\ \dots & \dots & \dots\end{bmatrix}\\ A\in\mathbb{R}^{l\times m}, B\in\mathbb{R}^{n\times p}, &A\otimes B\in\mathbb{R}^{ln\times mp}\end{aligned} \tag{15}$$

- Vectorization

$$A = \begin{bmatrix} A_1 & A_2 & \dots & A_m \end{bmatrix} \in \mathbb{R}^{l \times m}$$
$$\text{vec}(A) = \begin{bmatrix} A_1 \\ A_2 \\ \dots \\ A_m \end{bmatrix} \in \mathbb{R}^{lm \times 1} \tag{16}$$

- The vec trick

$$\begin{aligned} &\text{vec}(ABC) = \left(C^T \otimes A\right) \text{vec}(B) \quad \text{let } C = I \\ \Rightarrow &\text{vec}(AB) = \left(B^T \otimes I\right) \text{vec}(A) = (I \otimes A)\, \text{vec}(B) \end{aligned} \tag{17}$$

If we want to diff a matric x w.r.t. a vector, vectorize the matrix: (implied whenever we diff a matrix)

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

Back to the Taylor expansion of f(x), we now have:

$$f(x + \Delta x) \approx f(x) + \frac{\partial f}{\partial x}\Delta x + \frac{1}{2}\left(\Delta x^T \otimes I\right)\frac{\partial^2 f}{\partial x^2}\Delta x (Ais \frac{\partial f}{\partial x} and B is \Delta x) \tag{18}$$

By doing so, we can calculate the g and G more intuitively and simply.

Sometimes we also need to diff through a transpose:

$$\begin{aligned} \frac{\partial}{\partial x}\left(A(x)^T B\right) &= \left(B^T \otimes I\right) T \frac{\partial A}{\partial x} \\ T * \text{vec}(A) &= \text{vec}\left(A^T\right) \end{aligned} \tag{19}$$

T is a commutator matrix above.

According to the math above, we can calculate the jacobian of the Action-Value function S(x,u):

$$\begin{aligned} S_k(x,u) &= l_k(x,u) + V_{k+1}(f(x,u)) \\ \Rightarrow \frac{\partial S}{\partial x} &= \frac{\partial l}{\partial x} + \frac{\partial v}{\partial x}\frac{\partial f}{\partial x} \Rightarrow g_x = \nabla_x l + A_k^T p_{k+1} \\ \Rightarrow \frac{\partial S}{\partial u} &= \frac{\partial l}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial f}{\partial u} \Rightarrow g_u = \nabla_u l + B_k^T p_{k+1} \\ G_{xx} &= \frac{\partial g_x}{\partial x} = \nabla^2_{xx} l(x,u) + A_k^T \nabla^2 V_{k+1} A_k + \left(p_{k+1}^T \otimes I\right) T \frac{\partial A_k}{\partial x} \\ G_{uu} &= \frac{\partial g_u}{\partial u} = \nabla^2_{uu} l(x,u) + B_k^T \nabla^2 V_{k+1} B_k + \left(p_{k+1}^T \otimes I\right) T \frac{\partial B_k}{\partial u} \\ G_{xu} &= \frac{\partial g_x}{\partial u} = \nabla^2_{xu} l(x,u) + A_k^T \nabla^2 V_{k+1} B_k + \left(p_{k+1}^T \otimes I\right) T \frac{\partial A_k}{\partial u} \end{aligned} \tag{20}$$

- The last term of G is the tensor terms, if we throw this tensor term, then the method is called iLQR, else is the DDP.
- This handling method is similar to the Newton method and the Guass-Newton method. So, In iLQR, the amount of itteration is larger than the counterpart of the DDP, while the iLQR is generaly faster than the DDP.

9.3 DDP details and extensions

DDP recap

- Solve the unconstrained Trajectory problem:

$$min_{x_{1:N}, u_{1:N-1}} J = \sum_{k=1}^{N-1} l_k(x_k, u_k) + l_N(x_N) \tag{21}$$

$$\text{s.t.} \quad x_{k+1} = f\left(x_k, u_k\right) \tag{22}$$

- Backward pass by the Taylor expansion and the ricatti method:

$$\begin{aligned} V_k(x+\Delta x) &\approx V_k(x) + p_k^T \Delta x + \tfrac{1}{2}\Delta x^T P_k \Delta x \\ p_N &= \nabla_x l_N(x), P_N = \nabla_{xx}^2 l_N(x) \\ V_{N-1}(x+\Delta x) &= min_{\Delta u} S(x+\Delta x, u+\Delta u) \end{aligned} \tag{23}$$

$$\begin{aligned} \Delta u_{N-1} &= -d_{N-1} - K_{N-1}\Delta x \\ P_{N-1} &= G_{xx} + K_{N-1}^T G_{uu} K_{N-1} - G_{xu} K_{N-1} - K_{N-1}^T G_{ux} \\ p_{N-1} &= g_x - K_{N-1}^T g_u + K_{N-1}^T G_{uu} d_{N-1} - G_{xu} d_{N-1} \end{aligned} \tag{24}$$

- Forward Rollout: by the Taylor expansion and the ricatti method:

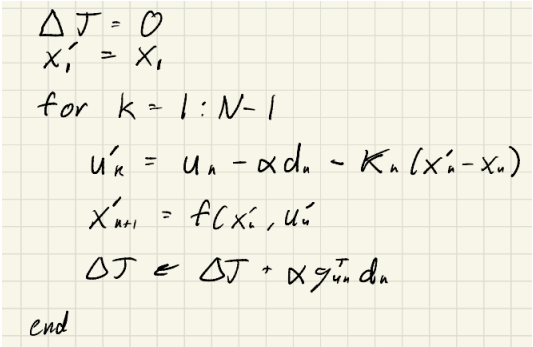

Figure 21: Forward Rollout

- Line search:

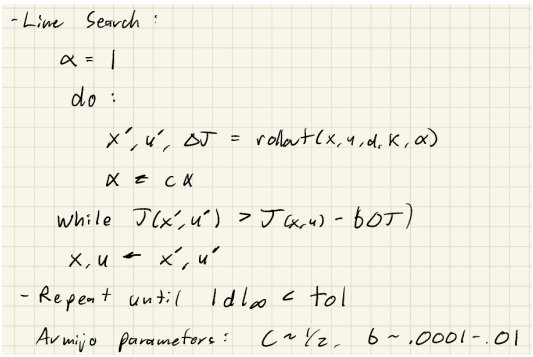

Figure 22: Line Search

9.3.1 Example

Examples shown in the code are the Cart pole and the acrobot swing-up.

- DDP can converge in fewer iterations but iterations are more expensive. iLQR often wins in wall-clock time.
- Problem is nonconvex: can land in different local optima depending on the initial guess.

9.3.2 Regulization

Just like the standard Newton method, here, V(x) and the S(x,u) hessians can become indefinite in the backward pass.

Regulization is Definitely necessary for the DDP, often a good idea for iLQR as well, but it is not necessary for iLQR.

Many options for regularization:

- Add a multiple of identity to $\Delta^2 S(x,u)$, just like what we do in the Newton method.
- Regularize P_n or the G_n as needed in the backward pass.
- Regularize just $G_{uu} = \Delta^2_{uu} S(x,u)$, this is the only matrix you have to invert.

The last method above is a good choice for the iLQR but not for the DDP, because theoretically, the other matrix of iLQR is at least semi-definite, while those of DDP might not.

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

9.3.3 DDP: pros and cons

Merits:

1. can be very fasy (iteration + wall-clock)
2. One of the most efficient methods due to exploitation of DP structure.
3. Always dynamically feasible due to forward rollout ⇒ can always execute on robot
4. Comes with TVLQR tracking controller for free ⇒ can be very effective for online use.

Cons:

1. do not natively handle constraints
2. Does not support infeasible initial guess for state trajectory due to forward rollout. Bad "maze", "bug trap" problems.
3. can suffer from numerical ill-conditioning on long trajectories.

9.4 Handling Constraints in DDP

How to put constraints in DDP? Many options depending on type of constraint.

Torque limits are often handled with a "squashing function", e.g. tanh(u):

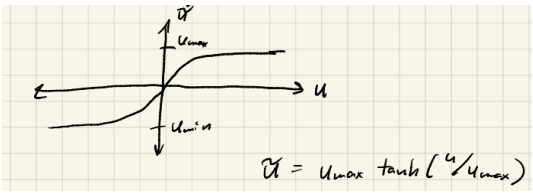

Figure 23: Torque limit

the squashing function is Effective, but adds nonlinearity and may need more iterations

Better option: solve box-constraint QP in the backward pass:

$$\begin{aligned} \Delta u = & \, argmin_{\Delta u} S(x + \Delta x, u + \Delta u) \\ & s.t. u_{min} \leq u + \Delta u \leq u_{max} \end{aligned} \tag{25}$$

- State constraints are harder. Often penalties are added to cost function. Can cause ill-conditioning.
- Better option: Wrap the entire DDP algorithm in an augmented lagrangian method.
- AL method adds linear(multiplier) and quadratic(penalty) terms to the cost ⇒ fits DDP nicely.

9.5 Handling Minimum time problems

$$\begin{aligned} \min_{x(t),u(t),T} J &= \int_0^T 1dt \\ \ni \dot{x} &= f(x,u) \\ x(T) &= x_{\text{goal}} \\ U_{\min} \leq u(t) &\leq U_{\max} \end{aligned} \tag{26}$$

- We don't want to change the number of knot points
- Make h(time step) from RK a control input:

$$x_{k+1} = f(x_k, \tilde{u}_k), \tilde{u}_k = [u_k; h_k] \tag{27}$$

- Also, scale the cost by h e.g.

$$J(x,u) = \sum_{n=1}^{N-1} h_k l(x_k, u_k) + l_N(X_N) \tag{28}$$

- Always nonlinear/nonconvex 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 error.

9.5.1 Code Example

10 Direct Trajectory optimization

Basic strategy: Discretize and "transcribe" continuous-time optimal control problem into a nonlinear control program (NLP)

$$\begin{aligned} \min_x & f(x) \\ \text{s.t.c } & (x) = 0 \\ & d(x) \leq 0 \end{aligned} \tag{29}$$

All functions assumed C^2 smooth. Most common solver: IPOPT(free), SNOPT(commercial), KNITRO(commercial). For those solver, the common solution strategy is Sequential Quadratic programming(SQP).

10.1 Sequential Quadratic programming (SQP)

Strategy: Use 2^{nd}-order Taylor expansion of the Lagrangian and linearize C(x),d(x) using Use 1^{st}-order Taylor to approximate NLP as a local QP problem.

$$\begin{gathered} \min_{\Delta x} \tfrac{1}{2}\Delta x^T H \Delta x + g^T \Delta x \\ \ni c(x) + C\Delta x = 0 \\ d(x) + D\Delta x \leq 0 \\ \text{where } H = \tfrac{\partial L}{\partial x}, g = \tfrac{\partial L}{\partial x}, C = \tfrac{\partial c}{\partial x}, D = \tfrac{\partial d}{\partial x b} \\ L(x,\lambda,\mu) = f(x) + \lambda^T c(x) + \mu^T d(x) \end{gathered} \tag{30}$$

Solve the QP to compute the primal-dual search direction:

$$\Delta z = \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta\mu \end{bmatrix} \tag{31}$$

And then perform a line search with the merit function. But if it only has the equality constraints, reduces to Newton's method on the Lagrangian:(just the KKT system)

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -\frac{\partial L}{\partial x} \\ -c(x) \end{bmatrix} \tag{32}$$

So, as we can see the SQP method is the generalization of Newton's method to inequality-constrained problems.

- Can use any QP solver to solve the sub-problems, but good implementations like warm start are used in previous QP iterations.
- For good performance on trajectory problems, using the advantage of sparsity in KKT system is crucial, SNOPT (sparse Nonlinear Optimization) does well.
- If the inequalities are convex (e.g. conic) 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.

10.2 Direct Collocation

So far, we are used explicit RK methods:

$$\dot{x} = f(x,u) \rightarrow x_{k+1} = f\left(x_k, u_k\right) \tag{33}$$

This makes sense if you are doing rollouts However, in a direct method where dynamics are enforced as equality constraints between knot points, this doesn't matter:

$$c_k(x_k, u_k, x_{k+1}, u_{k+1}) = 0 \tag{34}$$

- Collocation methods represent trajectories with polynomial splines and enforce dynamic on spline derivatives.
- Classic DIRCOL algorithm uses cubic splines for state trajectories and piecewise linear interpolation for u(t).
- Very high-order polynomials are sometimes used (e.g. spacecraft trajectories) but not common.

10.2.1 DIRCOL Spline Approximations

Three-order polynomials (four unknown parameters, which can make sure the parameters of $x_k, x_{k+1}, \dot{x}_k, \dot{x}_{k+1}$): The calculate processes are:

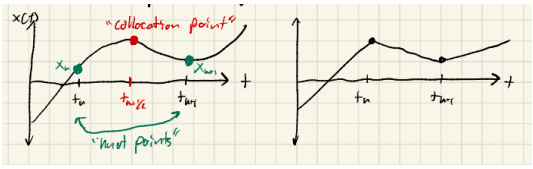

Figure 24: DIRCOL Spline Approximations

$$\begin{gathered}
x(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 \\
\dot{x}(t) = c_1 + 2c_2 t + 3c_3 t^2 \\
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h & h^2 & h^3 \\ 0 & 1 & 2h & 3h^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_k \\ \dot{x}_k \\ x_{k+1} \\ \dot{x}_{k+1} \end{bmatrix} \\
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3/h^2 & -2/h & 3/h^2 & -1/h \\ 2/h^3 & 1/h^2 & -2/h^3 & 1/h^2 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \\ x_{k+1} \\ \dot{x}_{k+1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}
\end{gathered} \tag{35}$$

Once we get the parameters c_0 to c_3, we can calculate the middle point state between two knot points:

$$\begin{gathered}
x_{k+1/2} = x\left(t_k + h/2\right) = \tfrac{1}{2}\left(x_k + x_{k+1}\right) + \tfrac{h}{8}\left(\dot{x}_k - \dot{x}_{k+1}\right) = \tfrac{1}{2}\left(x_k + x_{k+1}\right) \\
+\tfrac{h}{8}\left(f\left(x_k, u_k\right) - f\left(x_{k+1}, u_{k+1}\right)\right) \\
\dot{x}_{k+1/2} = \dot{x}\left(t_k + h/2\right) = -3/2h\left(x_k - x_{k+1}\right) - 1/4\left(\dot{x}_k + \dot{x}_{k+1}\right) = -3/2h\left(x_k - x_{k+1}\right) - 1 \\
/4\left(f\left(x_k, u_k\right) + f\left(x_{k+1}, u_{k+1}\right)\right)
\end{gathered} \tag{36}$$

In this point, we can enforce the system dynamics constraints $\dot{x}_{k+1/2} = f(x_{K+1/2}, u_{K+1/2})$, then we can get the :

$$\begin{gathered}
c_i\left(x_k, u_k, x_{k+1}, u_{k+1}\right) = f\left(x_{k+1/2}, u_{k+1/2}\right) \\
-\left(-3/2h\left(x_k - x_{k+1}\right) - 1/4\left(f\left(x_k, u_k\right) + f\left(x_{k+1}, u_{k+1}\right)\right) = 0\right.
\end{gathered} \tag{37}$$

- Note that only x_n and u_n are decision variables
- The method above is called the "Hermit-Simpson" integration.
- Achieves 3^{rd} order integration accuracy like RK3
- Require fewer dynamics calls than explicit RK3:
  EXplicit RK3: 3 dynamic evals per step:

$$\begin{aligned} k_1 &= f(x_n, y_n), \\ k_2 &= f\left(x_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_1\right), \\ k_3 &= f(x_n + h, y_n - hk_1 + 2hk_2), \\ y_{n+1} &= y_n + \tfrac{h}{6}\left[k_1 + 4k_2 + k_3\right]. \end{aligned} \tag{38}$$

  Hermite-Simpson: only 2 evals per time step!

$$f\left(x_{k+1/2}, u_{k+1/2}\right) - (-3/2h\,(x_k - x_{k+1}) - 1/4\,(f(x_k, u_k) + f(x_{k+1}, u_{k+1})) = 0 \tag{39}$$

- Since the dynamics calls often dominate computational cost, this can be a 50% time-saving.

10.2.2 Code Examples

Acrobot with DIRCOL
Note: A warm start with a dynamically infeasible guess can help a lot!

10.3 Algorithm Recap: deterministic Optimal Control Algorithm

10.3.1 Linear/"local(Linearization works)" Control Problems

Classification by constraint or not.

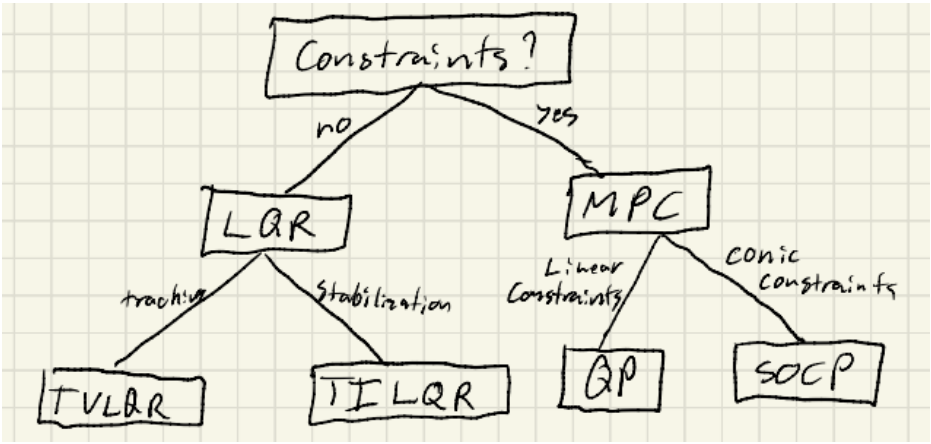

Figure 25: Methods

10.3.2 Nonlinear Trajectory Optimization/Planning

- DDP is often a good choice for online/real-time applications where speed is critical and constraint tolerance is not critical.
- DIROL is often a good choice for offline trajectory design, especially over long horizons and/or with complex constraints.

	DIRCOL	DDP/iLQR(indirect)
Dynamic feasible	Only respect dynamic at convergence (not suitable for real-time control)	Always dynamically feasible (real-time control)
State estimation	Can supply infeasible guess (can combine with some initial guess by RRT/A* or...)	Can only guess controls (not suitable for combining with some rough search A*/RRT)
Constraint handle	Can handle arbitrary constraints	Hard to handle constraints
Efficiency	Typically not so fast	Very fast(local) convergence
Controller	Tracking controller must be designed seperately	TVLQR tracking controller is free
Embedded system	Difficult to implement large scale SQP solver	Easy to implement on embedded system
Numerical problem	Numerical robust	Has issues with ill-conditioning

11 Attitude stuff

Many robot systems undergo large rotations (quadrotors, airplanes, spacecraft, underwater vehicles, legged robots)

Navie angle-based parameterizations (Euler angles) have singularities that cause failures and/or require hacks.

11.1 Rotation matrix

World frame N and body frame B, we have:

$$\begin{bmatrix} {}^N x_1 \\ {}^N x_2 \\ {}^N x_3 \end{bmatrix} = Q \begin{bmatrix} {}^B x_1 \\ {}^B x_2 \\ {}^B x_3 \end{bmatrix} = \begin{bmatrix} {}^B x_1^T \\ --- \\ {}^B n_2^T \\ --- \\ {}^B n_3^T \end{bmatrix} \begin{bmatrix} {}^B x_1 \\ {}^B x_2 \\ {}^B x_3 \end{bmatrix} = \begin{bmatrix} {}^N b_1 & | & {}^N b_2 & | & {}^N b_3 \end{bmatrix} \begin{bmatrix} {}^B x_1 \\ {}^B x_2 \\ {}^B x_3 \end{bmatrix} \tag{1}$$

Each row of the rotation matrix Q represents the value of the axis of N in the B coordinate system, Each column of the rotation matrix Q represents the value of the coordinate axis of B in the N coordinate system. So, the Q is an orthogonal matrix. we have:

$$\begin{aligned} Q^T Q = I \Rightarrow Q^{-1} = Q^T \\ \Rightarrow det(Q) = 1 \\ \Rightarrow Q \in SO(3) \quad \textbf{“Special orthogonal group in 3D”} \end{aligned} \tag{2}$$

11.2 Rotation matrix Kinematic (how to integral a gyro)

if we have a constant vector ${}^B x$ in the body frame (i.e. the IMU in the body frame). Then, the IMU pose in the world frame is:

$${}^N x = Q(t) {}^B x \tag{3}$$

Derive with respect to t, we have:

$${}^N x = Q {}^B x \Rightarrow {}^N \dot{x} = \dot{Q} {}^B x + Q {}^B \dot{x} = \dot{Q} {}^B x \tag{4}$$

When the object rotates in ω, the derivative of ${}^N x$ is:

$${}^N \dot{x} = {}^N \omega \times {}^N x = {}^N_B Q({}^B \omega \times {}^B x) \tag{5}$$

According to two equations above, we have:

$$\dot{Q} {}^B x = Q({}^B \omega \times {}^B x) \tag{6}$$

the fork multiplication can be written as an ordinary matrix multiplication by means of an antisymmetric matrix:

$$\omega \times x = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \hat{\omega} x \tag{7}$$

So, we have:

$$\Rightarrow \dot{Q}^B x = Q\hat{\omega}^B x \Rightarrow \dot{Q} = Q\hat{w} \tag{8}$$

So, we can integrate the rotation matrix Q as:

$$Q_{k+1} = Q_k + \dot{Q}_k \Delta t \tag{9}$$

However, For two rotation matrix R_1 and R_2, we should note that:

$$R_1 + R_2 \notin SO(3) \quad R_1 R_2 \in SO(3) \tag{10}$$

Conclusion: We could do dynamics with a rotation matrix, but a lot of redundancy

11.3 Rotation vector and Euler angle

Intuitively, a rotation can be described as a rotation axis and a rotation angle (Axis-Angle).

$$\boldsymbol{\phi} = \theta \boldsymbol{n} \tag{11}$$

Rotation matrix and the rotation vector can be changed from each other using (Rodrigue's Formula).

For the Euler angle method, it also has the Gimbal lock problem, resulting in the singularity.

11.4 Quaternion

As we can see, the Euler method the rotation vector has the singularity problem, the rotation matrix has the redundant problem. They all are not efficient enough in calculating the rotation. On the contrary, Quaternion is both non-singular and efficient.

In the 2D space, the rotation can be represented as Euler formulation:

$$e^{ix} = \cos x + i \sin x \tag{12}$$

The same principle, in 3D space, the rotation can be described as unit quaternion:

$$\boldsymbol{q} = q_0 + q_1 i + q_2 j + q_3 k \tag{13}$$

Also, quaternion can be described as a scalar part and a vector part:

$$\boldsymbol{q} = [s, v]^T, s = q_0 \in \mathbb{R}, \boldsymbol{v} = [q_1, q_2, q_3]^T \in \mathbb{R}^{\not\vDash} \tag{14}$$

11.4.1 Quaternion calculation

Quaternion multiplication

$$\begin{aligned} q_1 * q_2 &= \begin{bmatrix} s_1 \\ v_1 \end{bmatrix} * \begin{bmatrix} s_2 \\ v_2 \end{bmatrix} = \begin{bmatrix} s_1 s_2 - v_1^T v_2 \\ s_1 v_2 + s_2 v_1 + v_1 \times v_2 \end{bmatrix} \\ &= \begin{bmatrix} s_1 & -v_1^T \\ v_1 & s_1 I + \hat{v}_1 \end{bmatrix} \begin{bmatrix} s_2 \\ v_2 \end{bmatrix} = L(q_1) \begin{bmatrix} s_2 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} s_2 & -v_2^T \\ v_2 & s_2 I - \hat{v}_2 \end{bmatrix} \begin{bmatrix} s_1 \\ v_1 \end{bmatrix} = R(q_2) \begin{bmatrix} s_1 \\ v_1 \end{bmatrix} \end{aligned} \tag{15}$$

Quaternion conjugate

$$q^* = \begin{bmatrix} s \\ -v \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} s \\ v \end{bmatrix} = T \begin{bmatrix} s \\ v \end{bmatrix} \tag{16}$$

Inverse

$$\boldsymbol{q^{-1}} = \frac{\boldsymbol{q}^*}{||\boldsymbol{q}||} \tag{17}$$

Rotate a vector $\boldsymbol{p}$

$$\boldsymbol{p'} = \boldsymbol{q} * \boldsymbol{p} * \boldsymbol{q}^{-1} = q * R(q*) H x = L(q) R^T(q) H x = R^T(q) L(q) H x \tag{18}$$

Quaternion Kinematics

$$\begin{aligned} &\dot{q} \in \mathbb{R}^4, \omega \in \mathbb{R}^3, \\ &\dot{q} = \frac{1}{2} w * q = \frac{1}{2} L(q) H \omega \end{aligned} \tag{19}$$

Now, we can simulate dynamic with quaternion.

11.4.2 Quaternion to other rotation representing method

https://zhuanlan.zhihu.com/p/45404840.

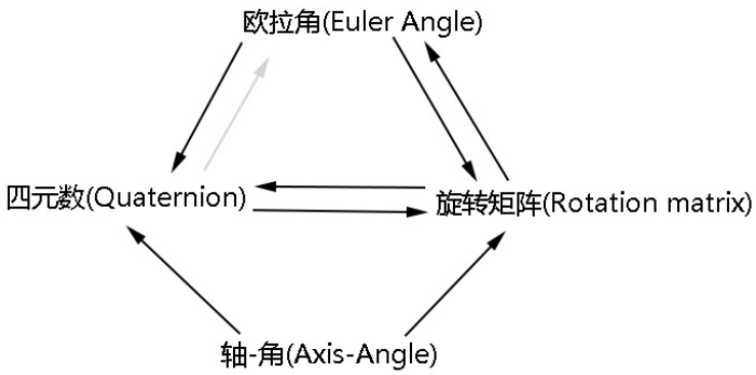

Figure 26: Rotation representative

11.5 Optimization with Quaternion

Geometry of quaternion

11.5.1 Examples: Pose estimation /(Wabba's Problem)

11.6 LQR with Quaternions

With the Quaternions, we can never drive it to zero, because it has the property of norm 1, so we should change the Quaternions to the error 3D thing like gibber parameter, and this 3D parameter can be driven to zero, then we turn this 3D parameter to Quaternions again. That's the whole trick.

Once you do so, just compute the LQR controller as usual. Concretely, Given a reference trajectory, we can linearize it as:

$$\begin{aligned} \bar{x}_{k+1} + \Delta x_{k+1} &= f(\bar{x}_k + \Delta x_k, \bar{u}_k + \Delta u_k) \\ &\approx f(\bar{x}_k, \bar{u}_k) + A_k \Delta x_k + B_k \Delta u_k \end{aligned} \tag{20}$$

Assume that the states of the robot contain Quaternions q, but we just turn it into gibber parameter and downsize the dimension of the system:

$$\begin{bmatrix} \Delta x_{k+1}[1:3] \\ \phi_{k+1} \\ \Delta x_{k+1}[8:n] \end{bmatrix} = \begin{bmatrix} I & & \\ & G(\bar{q}_{k+1}) & \\ & & I \end{bmatrix}^T A_k \begin{bmatrix} I & & \\ & G(\bar{q}_k) & \\ & & I \end{bmatrix} + \begin{bmatrix} I & & \\ & G(\bar{q}_{k+1}) & \\ & & I \end{bmatrix}^T B_k \Delta u_k \tag{21}$$

$$\Delta \tilde{x}_{\tilde{k}+1} = E(\bar{x}_{k+1}) A_k E(\bar{x}_k) \Delta x_k + E(\bar{x}_{k+1}) B_k \Delta u_k$$

Once we have these reduced Jacobians $\tilde{A}, \tilde{B}$:

$$\tilde{A}_k = E(\bar{x}_{k+1}) A_k E(\bar{x}_k), \tilde{B}_k = E(\bar{x}_{k+1}) B_k \tag{22}$$

We can compute the LQR as usual. When we run the controller, we calculate $\Delta\tilde{x}$ before multiplying by K.

$$\begin{aligned} \text{given} \quad x_n, \Delta\tilde{x}_k &= \begin{bmatrix} x_k[1:3] - \bar{x}_k[1:3] \\ \phi(L(\bar{q}_k)^T q_k) \\ x_k[8:n] - \bar{x}_k[8:n] \end{bmatrix} \\ w_k &= \bar{u}_k - K_k \Delta\tilde{x}_k \end{aligned} \tag{23}$$

11.6.1 Eaxmple: 3D Quadrotor

For details, see the code.

12 Contact dynamics

Please refer to : https://zhuanlan.zhihu.com/p/672725906

- P26
- Contact dynamics
- Hybrid systems modeling
- Trajectory for legged systems

We can't write a smooth ODE in the situation above, two options to solve this problem

- Even-based/hybrid formulation: Integrate ODE while checking for contact events using a "guard function" (e.g. $z \geq 0$). When contact happens, execute the "jump map" that models discontinuity then continue integrating ODE.
- Time-stepping / contact-implicit formulation: Slove a constrained optimization problem at each time step that enforces no interpretation between objects ($\phi(x) \geq 0$) by solving for contact forces jointly with the state (HW1 brick problem)

Both are widely used and have pros/cons, for the Hybrid formulation:

- In control, the hybrid formulation is easy to implement with a standard algorithm (e.g. DIRCOL)
- Downside: requires pre-specified "mode sequence" (which contacts are active at which time steps.)
- very successful in locomotion

For the contact-implicit method:

- Don't need the mode sequence pre-specified, but the optimization problems are much harder.

12.1 Example: Solving the Falling Brick problem in two ways

12.1.1 The time-stepping method

12.1.2 The hybrid method

12.2 Hybrid trajectory for legged Robots

One-legged hopper:

13 Collision Avoidance

13.1 Little trick

L1 trick: encourage sparsity slack T: Useful for encouraging sparsity identifying importance min fuel/actuator use

13.2 Collision Avoidance

13.2.1 Search/sample-based method

A*, RRT and so on. These kinds of planners are the only option for really hard problems (finding your way through a maze), but can be slow/inefficient/not the right tool for easier problems.

13.2.2 CBF: control barrier function

We should note that the CBF is not the controller, it just adjusts the input u that is generated by any other controller. CBF says that the output u may be unsafe, but we can adjust it using the CBF with some constraints to make the output u safe.

$$U_{safe} = f_{CBF}(u)$$

Note that maybe the output U is not optimal if we change it using the CBF, So, we have to consider this factor when using the CBF.

How to verify the output after using CBF? (exhausted test).
What about integrating the CBF into the original controller?

13.2.3 Motion planning with collision avoidance

14 Iterative Learning Control

- Reasoning about friction
- Iterative learning control

14.1 Friction in Trajectory Optimization

- For the
  - The constraint is often linearized:
  - The friction pyramid constraint is easy to enforce in a QP-based MPC controller.
  - Hybrid methods generally try to avoid slip. If you want to model stick-slip behavior need to add more modes.
  - Friction is hard to model. Coulomb is only an approximation that typically use a conservatively small value μ

14.2 What happens when our model has errors?

- Models are always approximate
- Simpler models are often preferred even if they are less accurate.
- Feedback (e.g. LQR/MPC) can often compensate for model error.
- Sometimes that is not enough

Several options: 1) Parameter Estimation: Classical "system ID" / "grey box" modeling 2) 3) 4)

14.3 Iterative Learning Control

(Sorry, not understand)

14.3.1 ICL Algorithm

15 Stochastic Optimal control and LQG

16 Kalman Filters and Duality

17 Robust Control and Minimax Optimization