Topics Covered:

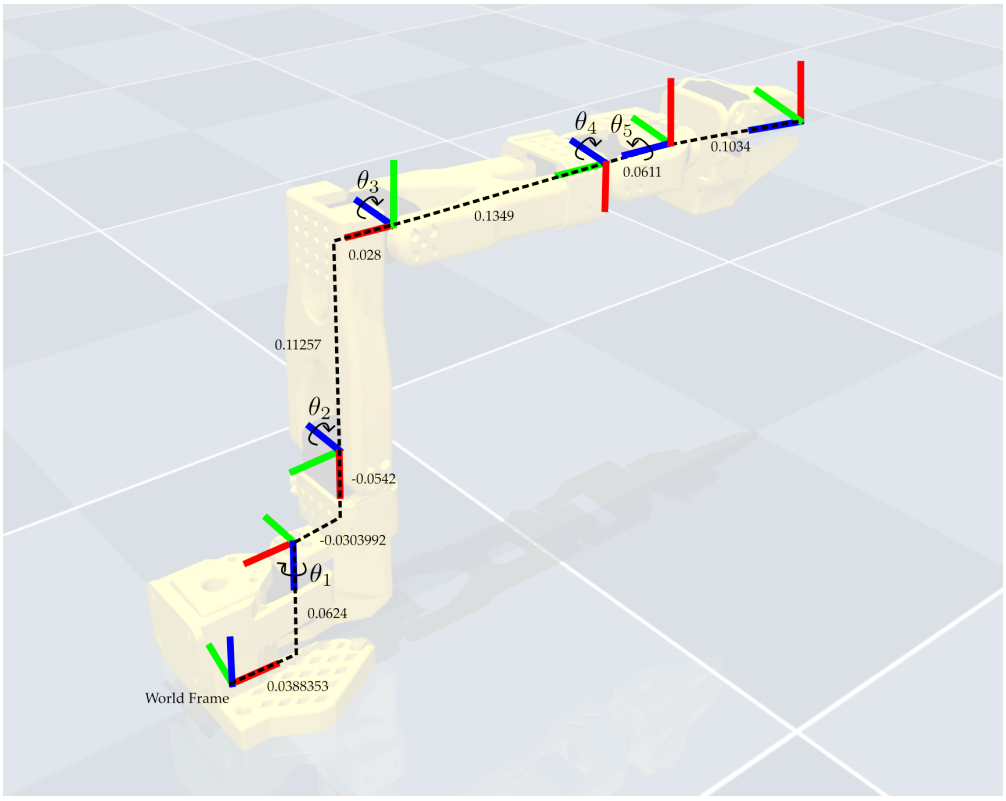
- Review of Geometric Approach
- Numerical Approach to Inverse Kinematics
- Optimization-based Techniques

Additional Reading:

- MLS Chapter 3, Section 3; LP 6.2

Review of Geometric Approach

For lab next week you will be asked to implement inverse kinematics for the SO-101 robot arms. The kinematic structure of these arms is shown below:



Note that this procedure becomes much simpler if we assume that the last two axes of rotation intersect at a common point (i.e. similar to a spherical wrist). Then, we can break the inverse kinematics problem into two parts: positioning the wrist center (center of joint 4) using θ_1, θ_2, and θ_3, and orienting the end-effector using θ_4 and θ_5.

Note that we can also exactly solve for θ_4 and θ_5 if we assume that the desired orientation is strictly about the z-axis of the world frame. This is a reasonable assumption for pick-and-place tasks where the end-effector is always pointing downwards.

However, if we don't want to make this assumption, we could use the numerical appraoach described next with the initial guess provided from our simplified geometric approach.

Numerical Approach

Analytic solutions often rely on simplifying assumptions (such as intersecting joint axes) which are not always valid. In these cases, the analytic inverse kinematic solutions can be used as initial guesses to an iterative numerical procedure.

These iterative approaches are often based on "root finding" methods, where the goal is to find the "roots" of a nonlinear function:

$$x^* \text{ s.t. } g(x^*)$$

In the context of inverse kinematics, our function will be defined as $g(\theta) = x_d - f(\theta)$, where x_d is the desired end-effector pose and $f(\theta)$ is the forward kinematics function. Then, the inverse kinematics problem becomes finding θ_d such that $g(\theta_d) = 0$.

Newton-Raphson Method

The Newton-Raphson method is a fundamental approach to nonlinear root-finding. It solves some equation $g(\theta) = 0$ numerically by iteratively updating some guess $\theta_k + 1$ using the formula:

$$\theta_{k+1} = \theta_k - \left(\frac{\partial g}{\partial \theta}(\theta_k)\right)^{-1} g(\theta_k)$$

where $\theta_k \in \mathbb{R}^n$ is the estimate of the input θ at iteration k. This formula comes from the first-order Taylor expansion of $g(\theta)$ around θ_k:

$$g(\theta) \approx g(\theta_k) + \frac{\partial g}{\partial \theta}(\theta_k)(\theta - \theta_k)$$

and setting $g(\theta) = 0$. The iterations are repeated until some stopping criterion is met. This criterion is typically defined as a small change in θ between iterations, or a small change in $g(\theta)$, i.e. $|g(\theta_k) - g(\theta_{k+1})|/|g(\theta_k)| \leq \epsilon$.

We can apply the Newton-Raphson method to inverse kinematics by defining $g(\theta_d) = x_d - f(\theta)$ and letting θ_0 be some initial guess. Notably, in this expression, we see the Jacobian appear:

$$x_d = f(\theta_d) = f(\theta_0) + \underbrace{\left.\frac{\partial f}{\partial \theta}\right|_{\theta_0}}_{J(\theta_0)} (\theta_d - \theta_0)$$

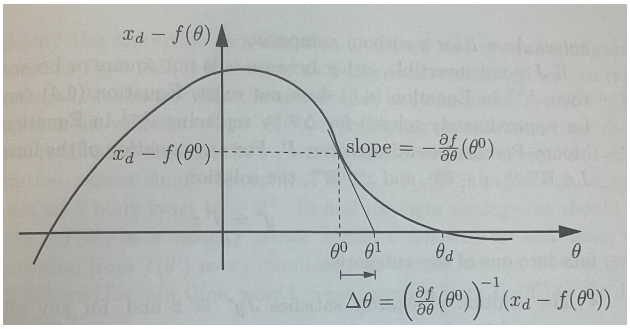
where $J(\theta_0) \in \mathbb{R}^{m \times n}$ is the (coordinate) Jacobian evaluated at θ_0.

By rearranging this expression, we can directly write our update rule as:

$$\delta\theta = J^{\dagger}(\theta_0)(x_d - f(\theta_0))$$

with $J^{\dagger}$ being the pseudoinverse, which we will also cover more in depth later in the course.

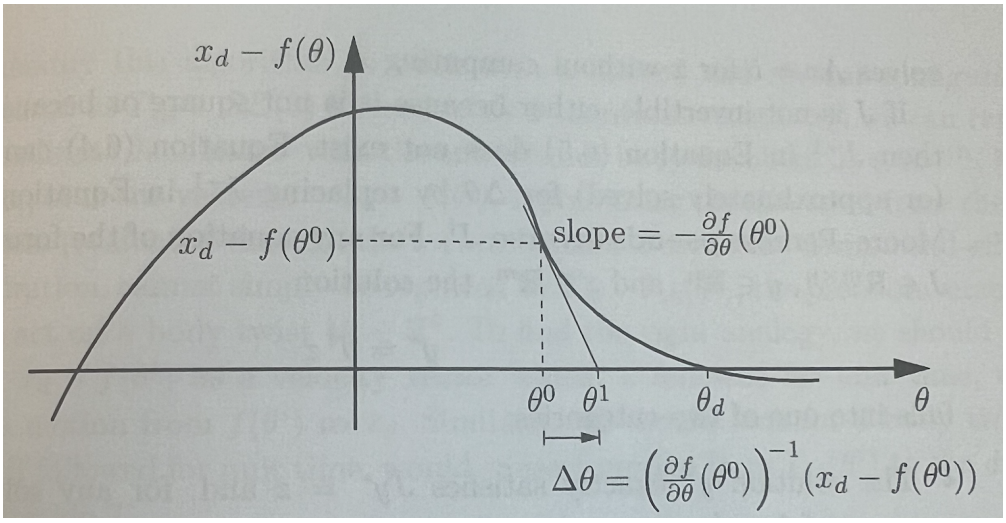
Overall, we can see the effect of θ_0 by considering the following example:



If θ_0 is chosen to be close to the solution, then eventually θ_k will converge to θ_d. However, if θ_0 is chosen to the left of the plateau of $x_d - f(\theta)$, then it will likely converge to the other root of $x_d - f(\theta)$.

How does Newton-Raphson work Graphically?

So how do we implement the Newton-Raphson method visually? Let's consider the same example:



As shown, the gradient step is the same as following the tangent line of the curve (evaluated at the point $g(\theta_k)$) to the point where it intersects the x-axis. This intuition comes from the following derivation:

$$\frac{\partial g(\theta_1)}{\partial \theta} = \frac{g(\theta_2) - g(\theta_1)}{\theta_2 - \theta_1}$$
$$=$$

Since we're trying to find the point where $g(\theta_2) \approx 0$, we can take this to be zero. We will also replace $\theta_2 - \theta_1$ with $\Delta\theta$:

$$\frac{\partial g(\theta_1)}{\partial \theta} = \frac{\partial (x_d - f(\theta))}{\partial \theta} = \frac{-\partial f}{\partial \theta_1} = \frac{0 - (x_d - f(\theta_1))}{\Delta\theta}$$
$$\frac{\partial f}{\partial \theta_1} = \frac{x_d - f(\theta_1)}{\Delta\theta}$$
$$\Delta\theta = \left(\frac{\partial f(\theta_1)}{\partial \theta_1}\right)^{-1}(x_d - f(\theta_1))$$

So this gradient step is the same thing as following the tangent line to the point where it intersects the x-axis.

Modern Optimization Techinques

Optimization-based techinques are advantageous in situations where an exact solution may not exist, or if infinite solutions exist (as is the case for redundant manipulators).

Most modern optimization solvers use some technique similar to Newton-Raphson. However, they often include additional features which allow for the addition of objective functions and constraints. But in all cases, good initial guesses are crucial for the success of the algorithm.

Typically, the optimization problem is set up as:

$$\begin{aligned} \min_{\theta} \quad & f(\theta) \\ \text{s.t.} \quad & c(\theta) = 0 \end{aligned}$$

which can be coded in python as:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

def objective(theta):
    return f(theta)
def constraint(theta):
    return c(theta)

# Example function to minimize
def f(theta):
    return (theta - 3) ** 2 + 4

# Example constraint function
def c(theta):
    return theta - 2

# Initial guess
theta0 = 0.5

# Perform the optimization
result =
    opt.minimize(objective, theta0, constraints={'type': 'eq', 'fun': constraint})

# Print the result
print("Optimal theta:", result.x)
print("Objective function value at optimal theta:", result.fun)
```

or in MATLAB as:

```
% Initial guess
theta0 = 0.5;

% Perform the optimization
[result, fval] = fmincon(@objective, theta0, [], [], [], [], [], [], @constraint);

% Print the result
fprintf('Optimal theta: %f\n', result);
fprintf('Objective function value at optimal theta: %f\n', fval);

% Define the objective function
function val = objective(theta)
    val = f(theta);
```

```
end

% Define the constraint function
function [c, ceq] = constraint(theta)
    c = []; % No inequality constraints
    ceq = theta - 2; % Equality constraint
end

% Example function to minimize
function val = f(theta)
    val = (theta - 3) ^ 2 + 4;
end
```