Robotics Simulation Environments

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

Department of Cybernetics and Robotics
Wrocław Univeristy of Science and Technology

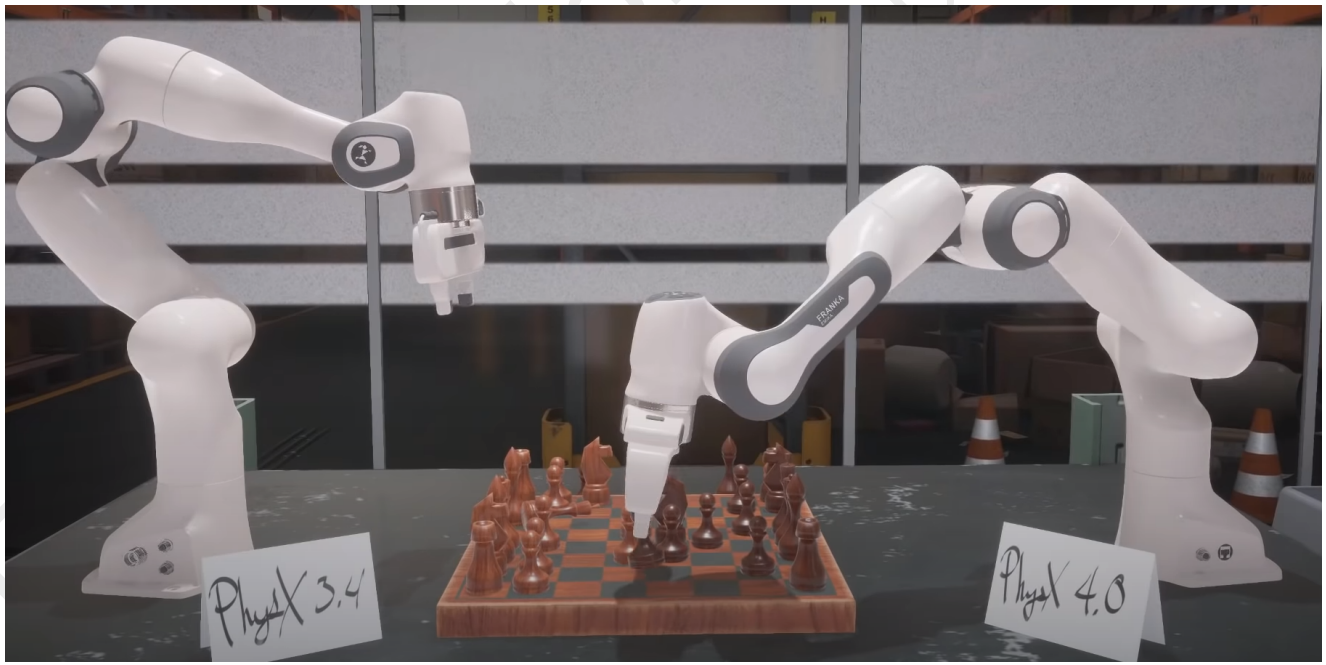
Wrocław University
of Science and Technology

Content

Introduction to Robotics Simulation

What is Simulation in Robotics?

- Simulation imitates the behavior of real systems using models.
- A model represents the behavior and characteristics of a specific process or system.
- Simulation shows how the model changes under various conditions over time.



1

[1] https://blogs.nvidia.com/

Why Robotics Systems are Simulated?

Motivation

Simulation is a fundamental tool in robotics that enables safe and scalable development of complex systems.

Why we use simulation

- Eliminates risk of hardware damage
- Enables fast iteration of algorithms
- Provides fully controlled environments
- Allows repeatable experiments

What simulation enables

- Large-scale data generation
- Testing edge and failure cases
- Training RL agents efficiently
- Long-term behavior analysis

The Reality Gap: Sim-to-Real Challenge

A simulation is a mathematical model, but models are never 100% perfect.

- **Sim to Reality Gap:** The discrepancy between simulated performance and real-world performance.
- **Causes:**
  - Unmodeled friction or "stiction."
  - Sensor noise and latency.
  - Inaccurate mass distributions.

Types of Robotics Simulation

- **Physics-Based Simulation:** Realistic modeling of dynamics, forces, collisions, and sensor behavior (e.g., rigid body dynamics, friction).
- **Kinematic / Low-Level Simulation:** Focus on robot motion without full physics, including joint states, velocities, and actuator-level control.
- **Sensor Simulation:** Emulation of sensors such as LiDAR, cameras, IMU, or sonar, often with noise and distortions.
- **High-Level / Behavioral Simulation:** Abstract representation of robot logic, navigation, and decision-making without detailed physical modeling.
- **Multi-Robot / System-Level Simulation:** Simulation of interactions between multiple robots or entire robotic systems in complex environments.

Popular Robotics Simulation Tools

- **Gazebo:** An open-source simulation tool widely used in the robotics community.
- **Nvidia Isaac Sim**: A simulation environment focused on the physics simulation and acceleration based on GPU computations.
- **V-REP (CoppeliaSim):** Versatile Robot Experimentation Platform with extensive features.
- **CARLA:** Mostly for autonomous vehicles purposes.
- **Unity:** A game development engine increasingly used for robot simulation.
- **MuJoCo (Multi-Joint dynamics with Contact)** is a physics engine used for simulating highly detailed and dynamic robotics and biomechanical systems.

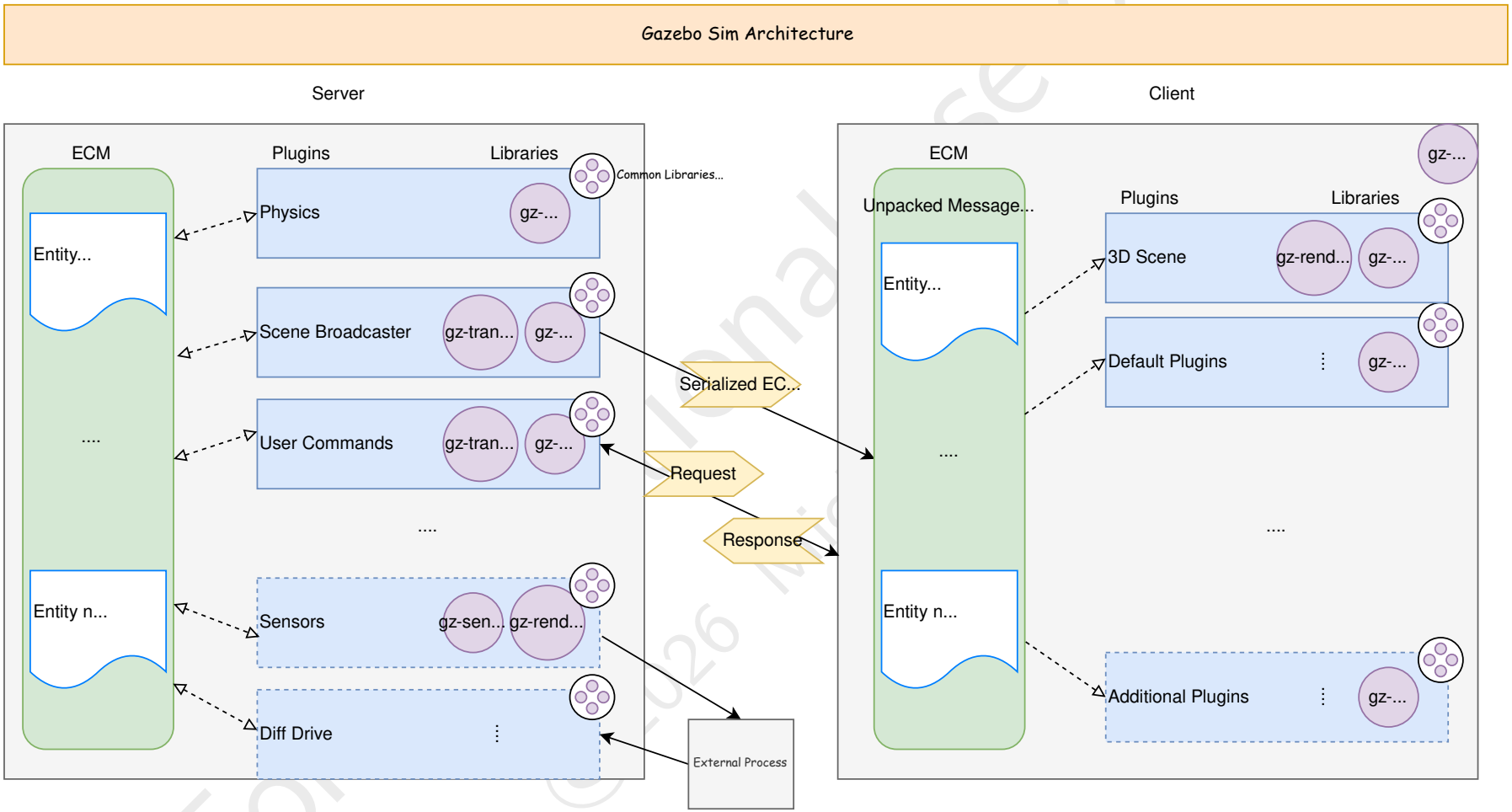
Gazebo

- **Introduction**
  - Gazebo is an open-source robotics simulation software.
  - Developed by the Open Source Robotics Foundation (OSRF).
- **Features**
  - Dynamics simulation: works with multiple high-performance physics engines.
  - 3D visualization for modeling and testing robot designs.
  - Extensive library of sensors (cameras, lidars, RGB-D sensors, IMU, GPS, etc.), actuators models, and environments.
  - TCP/IP Transport: Run simulation on remote servers.

2

[2] https://github.com/gazebosim/gz-sim

Gazebo Architecture



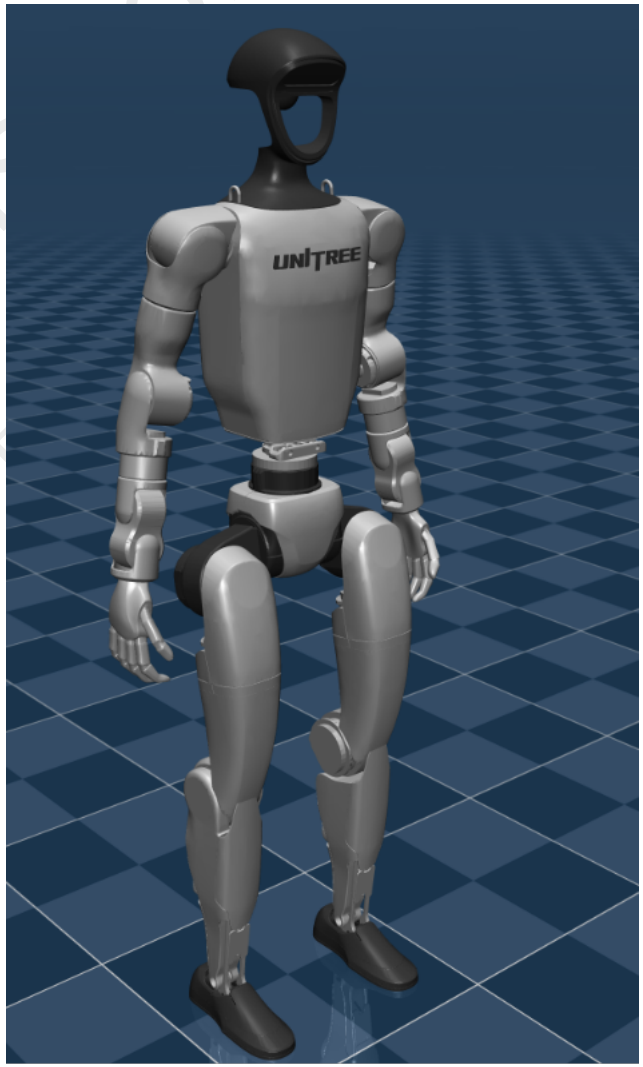
[3]

ECM = Entity Component Manager

[3] https://gazebosim.org/docs/latest/architecture/

MuJoCo (Multi-Joint dynamics with Contact)

- Physics engine for simulating articulated bodies and contacts
- Widely used in robotics, control, and reinforcement learning
- Developed by **Emo Todorov**, now maintained by **DeepMind**
- Focus on:
  - Accurate contact dynamics
  - Efficient computation (real-time simulation)
  - Stable numerical integration
- Supports:
  - Rigid body dynamics
  - Soft contacts and friction modeling
  - Sensors and actuators

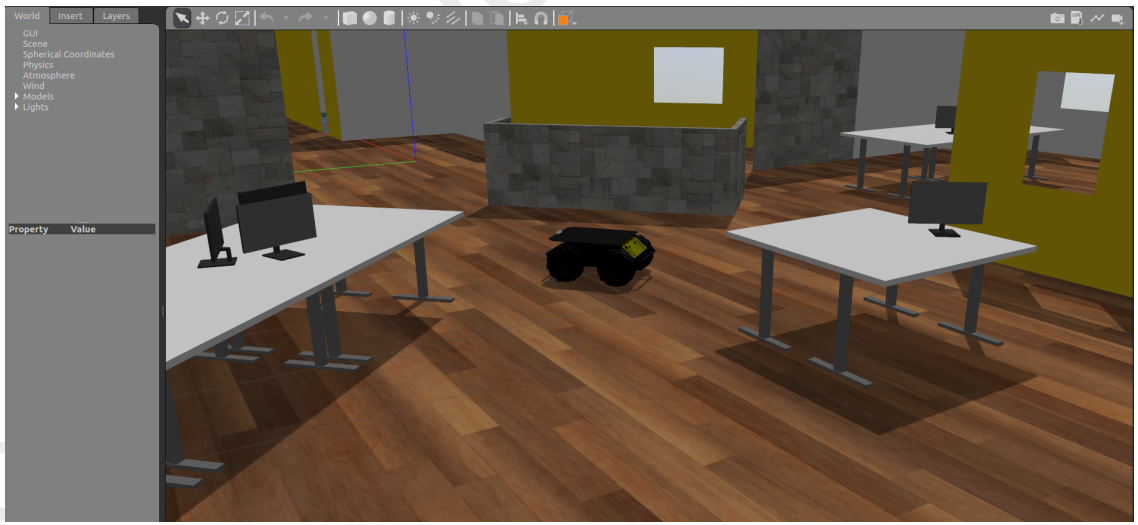


4

[4] https://mujoco.readthedocs.io/en/stable/models.html

Key Components of Physics-Based Robotics Simulation

- **Robot model:** Mathematical representation of the robot.
- **Sensor models:** Emulation of sensors like cameras, lidars, IMUs, etc.
- **Environment:** Virtual representation of the robot's surroundings.
- **Physics engine:** Simulation software that calculates the interactions between objects in the virtual world.



[5]

[5] https://www.clearpathrobotics.com/assets/guides/kinetic/husky/additional_sim_worlds

Physics Engines

Physics engines simulate physical interactions in virtual environments.

Core Components

- Rigid body dynamics
- Collision detection
- Numerical integration

Applications

- Robotics simulation
- Video games
- Engineering analysis

Challenges

- Real-time performance
- Numerical stability
- Simulation accuracy

Popular Engines:

Bullet Physics[6]	Open-source robotics/game physics
NVIDIA PhysX[7]	GPU-accelerated simulation
ODE[8]	Lightweight rigid body engine

[6] https://github.com/bulletphysics/bullet3
[7] https://developer.nvidia.com/physx-sdk
[8] https://www.ode.org/

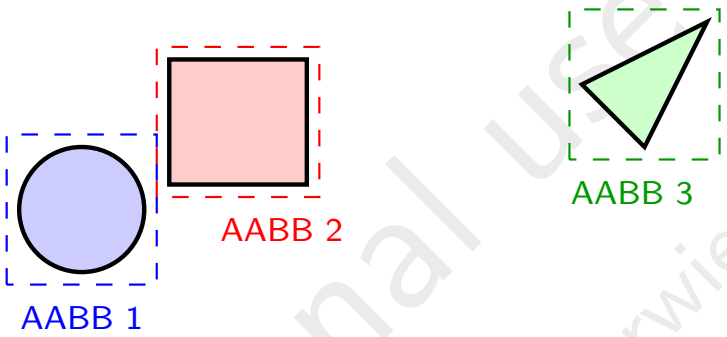
Simulation Loop in Physics Engine

1. **Input Scene Data**
   - Objects, masses, geometry
   - Initial positions and velocities
   - Forces and constraints
2. **Collision Detection**
   - Detect intersections between objects
   - Generate contact points
3. **Physics Computation**
   - Compute forces and torques
   - Solve constraints
   - Apply gravity and friction
4. **Numerical Integration**
   - Update velocities and positions
   - Advance simulation by timestep Δt
5. **Render / Output**
   - Updated object states
   - Sensor and environment feedback

Collision Detection in Robotics Simulation

- Collision detection is the computational problem of detecting when two or more objects intersect or come into contact in a virtual environment.
- **Purpose:**
  - Ensures realistic interactions between robots and the environment.
  - Prevents objects from overlapping in simulation.
  - Critical for path planning, motion control, and safety verification.
- **Techniques:**
  - Bounding volumes (AABB, Sphere, OBB)
  - Mesh-based collision checking
  - Distance queries and proximity tests

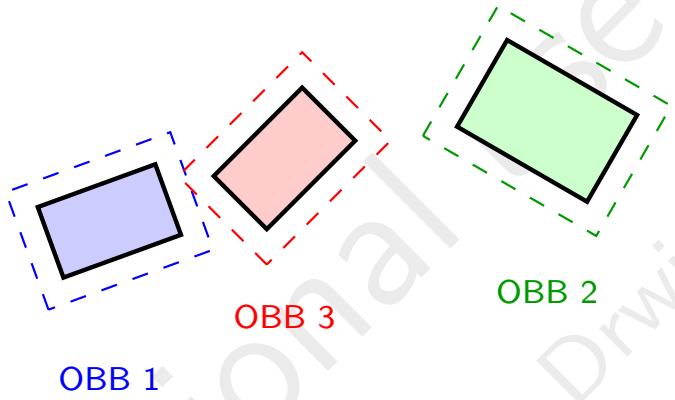
AABB: Axis-Aligned Bounding Boxes



Why AABBs?

Checking if two boxes overlap is a simple comparison of x and y coordinates. If AABBs don't touch, the complex shapes inside definitely don't touch.

OBB: Oriented Bounding Boxes



Why OBBs?

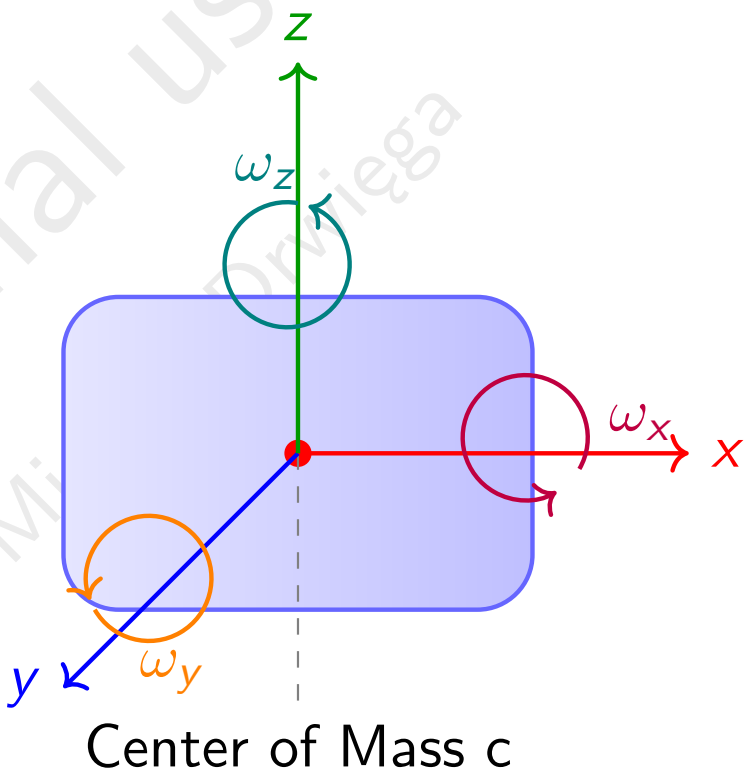
Oriented Bounding Boxes rotate together with objects, providing a tighter fit than AABBs and reducing false collision detections.

Rigid Body Dynamics

Rigid Body

Rigid Body Model

- Body assumed perfectly rigid
- No deformation under applied forces
- State representation:
  - Position x
  - Orientation R
- Motion variables:
  - Linear velocity v
  - Angular velocity ω



What is the Inertia Matrix? (Tensor)

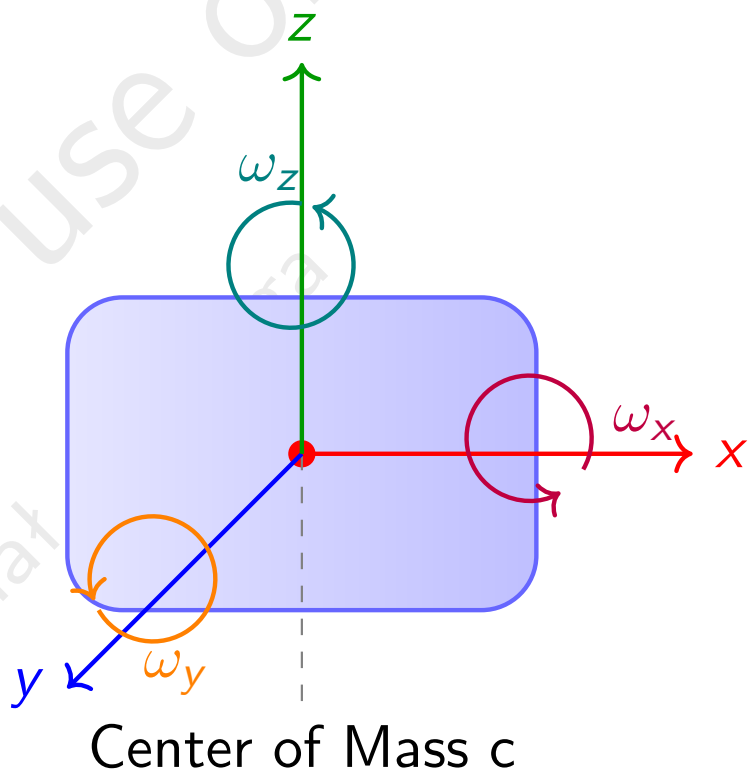
The inertia matrix describes how mass is distributed inside a rigid body and how difficult it is to rotate around different axes.

Properties

- Depends on mass distribution
- Defined relative to a reference frame
- Symmetric matrix
- Units: $\mathrm{kg} \cdot \mathrm{m}^2$

Physical Meaning

- Large inertia $\longrightarrow$ harder to rotate
- Small inertia $\longrightarrow$ easier to rotate



Example Inertia Matrix

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

Interpretation of the Inertia Matrix

Diagonal Terms

$$I_{xx},\ I_{yy},\ I_{zz}$$

- Resistance to rotation around principal axes
- Larger value $\longrightarrow$ harder to accelerate rotationally

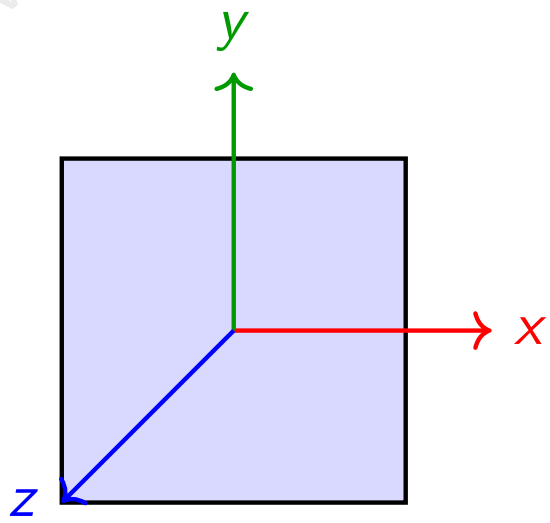
Off-Diagonal Terms

$$I_{xy},\ I_{xz},\ I_{yz}$$

- Represent coupling between axes
- Zero for perfectly symmetric bodies

Example Inertia Matrix

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$



Symmetric body
$\Rightarrow$ small off-diagonal terms

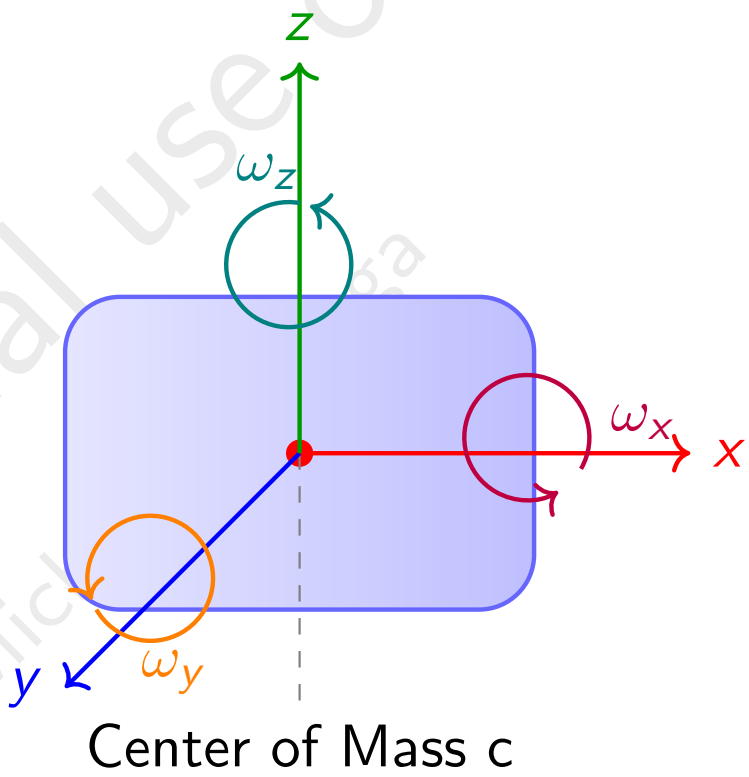
Rigid Body Dynamics - Newton-Euler Equations

Linear Dynamics

$$m\dot{\mathrm{v}} = \mathrm{F}$$

Rotational Dynamics

$$I\dot{\omega} + \omega \times (I\omega) = \tau$$



- m – mass
- v – linear velocity
- F – external force
- I – inertia tensor
- ω – angular velocity
- τ – torque

Forces and Torques in Simulation

Common Forces

- Gravity: $F_g = mg$
- Contact forces (normal + friction)
- Actuator forces (motors, thrusters)
- External disturbances (wind, water currents)

Torques

- Generated by actuators or force offsets

Limitations of Rigid Body Simulation

- No material deformation
- Simplified contact models
- Numerical instability possible
- Approximate friction behavior
- Simulation-reality gap

ODE Solving

What is an ODE?

An **Ordinary Differential Equation (ODE)** relates a function to its derivatives. It describes how a system changes over time.

The Initial Value Problem (IVP)

Find $y(t)$ given:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

- **Analytical:** Exact solution (often impossible).
- **Numerical:** Approximate solution using discrete time steps h.

Simulation: The ODE Engine

Simulation is the numerical execution of an ODE over a discrete timeline.

The Simulation Loop:

1. **State:** Current $y(t)$ (e.g., Robot position).
2. **Physics:** Calculate $\dot{y}$ via ODE.
3. **Step:** Solver computes $y(t + \Delta t)$.
4. **Update:** Advance clock and repeat.

The Trade-off

- **Step Size (h):** Smaller steps $\rightarrow$ Higher accuracy, but slower.
- **Stability:** Large steps in Euler can cause "explosions."

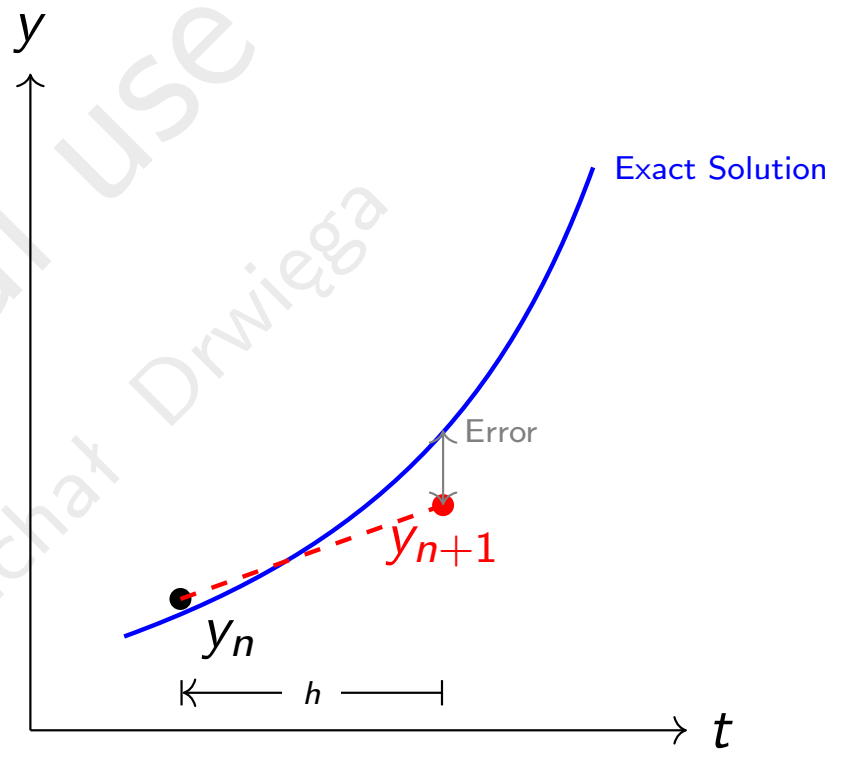
The Starting Point: Euler's Method

The simplest numerical procedure for solving ODEs.

- Uses the derivative at y_n to predict y_{n+1}.
- Linear approximation.

Formula

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$



Visualizing the tangential drift over a single step.

Step-by-Step Euler: The Scenario

Imagine a robot moving with a velocity that depends on time:

The ODE

$$\frac{dx}{dt} = 0.5t \quad \text{(Velocity grows over time)}$$

Initial Condition: At $t = 0$, position $x = 0$.
Goal: Find position at $t = 2$ using step size $h = 1$.

- **Step 1:** Calculate slope at current point.
- **Step 2:** Multiply slope by time step (h).
- **Step 3:** Add to current position to get the next point.

Step 1: From $t = 0$ to $t = 1$

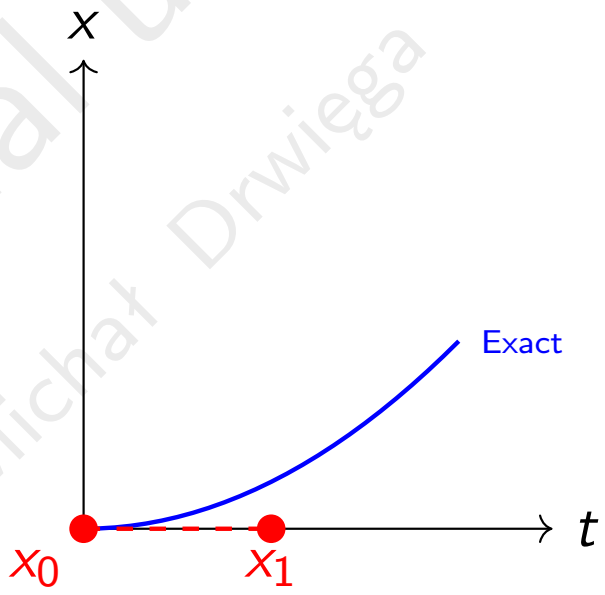
Current State: $t_0 = 0, x_0 = 0$
Step Size: $h = 1$

1. **Slope (k):** $f(0, 0) = 0.5(0) = 0$
2. **Update x:**

$$x_1 = x_0 + h(k)$$

$$x_1 = 0 + 1(0) = 0$$

Note: Because velocity was 0 at the start, Euler predicts we stayed still.



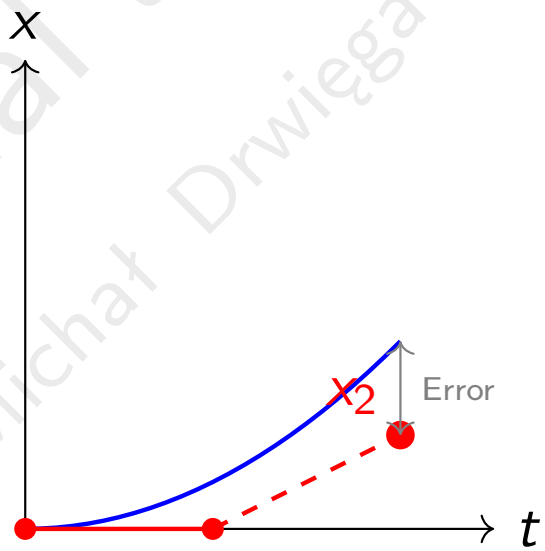
Step 2: From $t = 1$ to $t = 2$

Current State: $t_1 = 1, x_1 = 0$
Step Size: $h = 1$

1. **Slope (k):**
   $f(1, 0) = 0.5(1) = 0.5$
2. **Update x:**

$$x_2 = x_1 + h(k)$$

$$x_2 = 0 + 1(0.5) = 0.5$$



Final Result: $x(2) \approx 0.5$
True Value: $0.25(2)^2 = 1.0$
Error = 50% (Too large!)

The Runge-Kutta Logic

Why settle for one slope when you can use four?

- Euler's method only looks at the "start" of the interval.
- Runge-Kutta methods (RK4) sample slopes at the start, midpoint, and end.
- It calculates a **weighted average** of these slopes to minimize error.

The RK4 Algorithm

For a step size h, calculate four increments:

- $k_1 = f(t_n, y_n)$
- $k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2})$
- $k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2})$
- $k_4 = f(t_n + h, y_n + hk_3)$

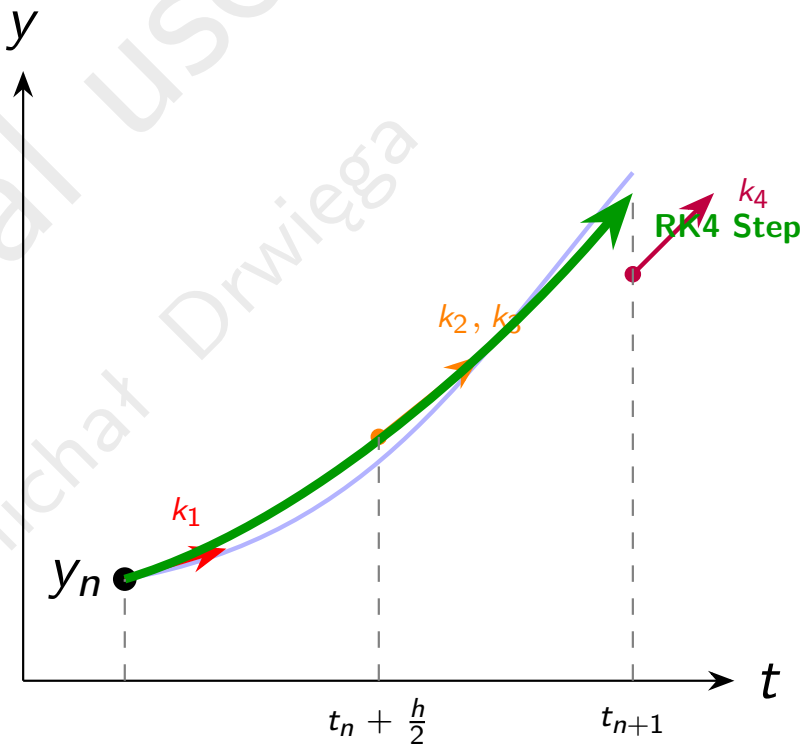
The Final Step

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

The Runge-Kutta Logic: Sampling the Curve

Why settle for one slope when you can use four?

- **Euler:** Uses only the slope at the start.
- **RK4:** Samples the "flow" at the start, midpoints, and end.
- **Result:** A weighted average that cancels out lower-order errors.

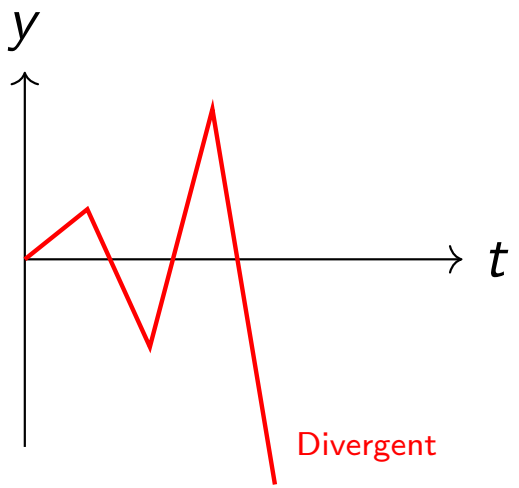


Challenges in ODE Solving

- **Stiffness:** When a system has widely varying time scales (e.g., a very stiff spring vs. a slow-moving heavy mass).
- **Discontinuities:** Sudden changes like collisions or limit switches that break the "smoothness" assumption of the solver.
- **Energy Drift:** In long-term simulations, small numerical errors can accumulate, causing the system to gain or lose energy unnaturally.

Numerical Instability

If the step size h is too large for the system's "stiffness," the solution will oscillate and grow toward infinity—this is a simulation **"Explosion."**



Example: Unicycle Model

System Description

- Planar mobile robot (unicycle model)
- State:
  - Position (x, y)
  - Heading angle θ
- Control inputs:
  - Linear velocity v
  - Angular velocity ω

Assumptions

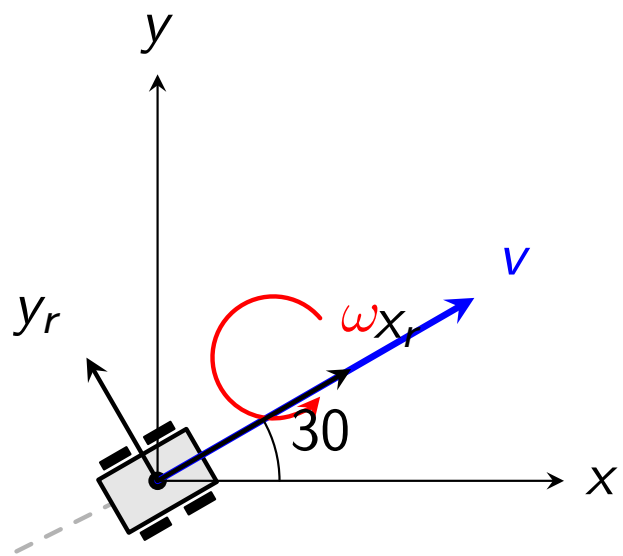
- No lateral slipping
- Motion constrained to heading direction

Kinematic Model

$$\dot{x} = v\cos(\theta)$$

$$\dot{y} = v\sin(\theta)$$

$$\dot{\theta} = \omega$$



Example: Unicycle Model

Continuous model

$$\dot{x} = v\cos(\theta), \quad \dot{y} = v\sin(\theta), \quad \dot{\theta} = \omega$$

Goal: simulate motion over time

We approximate the solution using **Euler integration**:

$$x_{k+1} = x_k + v\cos(\theta_k)\,\Delta t$$

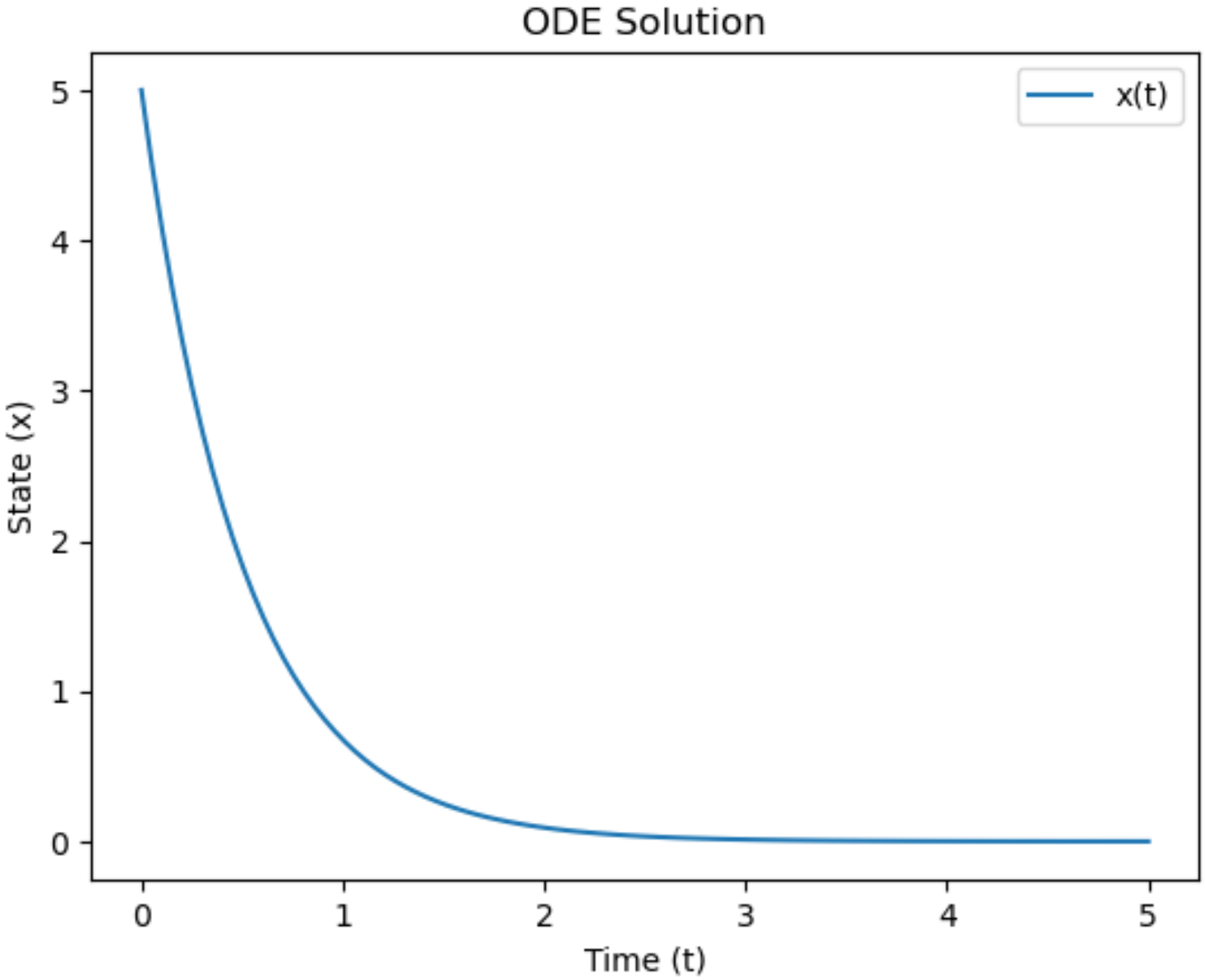
$$y_{k+1} = y_k + v\sin(\theta_k)\,\Delta t$$

$$\theta_{k+1} = \theta_k + \omega\Delta t$$

ODE Solving with Python SciPy

```python
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# ODE dot(x) = f(t, x, p), where
# t -- time, x -- state, p -- parameter
def f(t, x, p):
    return -p * x

T = [0, 5] # Time horizon
x0 = [5] # Initial state
p = 2.0 # Parameter

# Solve
sol = solve_ivp(f, T, x0, args=[p])

# Plot the solution
plt.plot(sol.t, sol.y[0], label='x(t)')
plt.title('ODE Solution')
plt.xlabel('Time (t)'); plt.ylabel('State (x)')
plt.legend(); plt.show()
```

Example - ODE solving with SciPy



Robot modeling – URDF

Introduction to URDF

What is URDF?

- **Definition:** URDF (Unified Robot Description Format) is a standard XML format for describing a robot's structure, kinematics, and dynamics.
- **Purpose:** Used for modeling and visualizing robots, aiding in simulation and control.

Features

- **Standardization:** Provides a standardized format for robot descriptions.
- **Interoperability:** URDF is supported by different ROS tools.
- **Simulation:** Facilitates accurate simulation of robot behavior (URDF is supported by multiple simulation environments).

URDF: Capabilities and Limitations

Capabilities

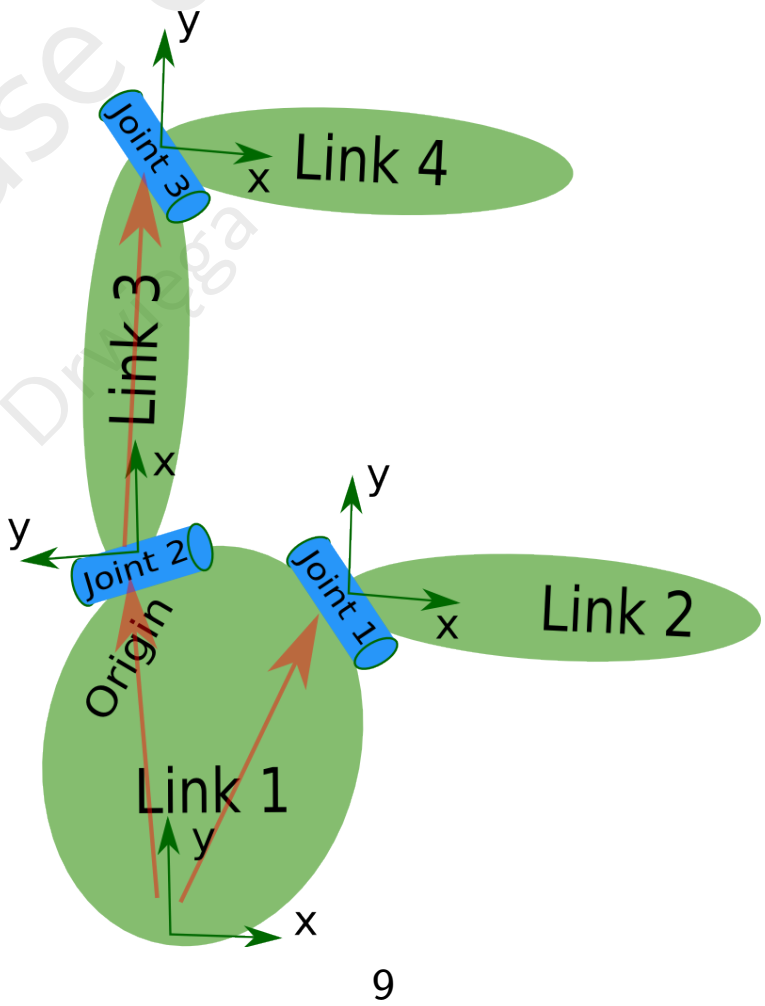
- **Kinematic and dynamic description**: Defines the robot's physical properties, such as mass, center of mass, and inertia for each link.
- **Visual representation**: Specifies the robot's visual appearance using 3D meshes and materials for rendering.
- **Collision model**: Simplifies geometry to define the robot's collision boundaries, useful in simulation and planning.

Limitations

- **Tree structure only**: It cannot represent closed kinematic chains or parallel robots.
- **No support for flexible elements**: Does not support flexible or deformable parts.
- **Limited dynamics**: Lacking advanced dynamics like damping or friction coefficients.
- **Static visual model**: Visual properties cannot change dynamically, limiting use in scenarios where the robot appearance may vary.

Key Components of URDF

- **Link:** Represents a rigid body in the robot.
- **Joint:** Defines the connection between two links and their relative motion.
- **Material:** Specifies visual and collision properties.
- **Transmission:** Describes how joint efforts result in link movements.



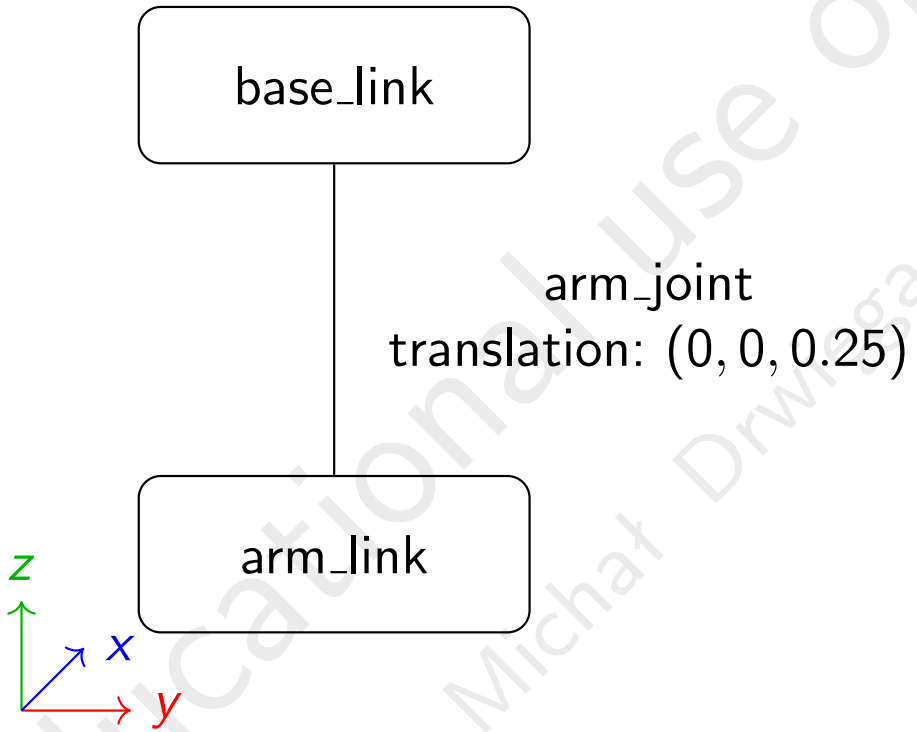
9

[9] https://wiki.ros.org/urdf/XML/model

URDF example

```
<robot name="simple_arm">
  <link name="base_link"/>

  <link name="arm_link">
    <visual>
      <geometry>
        <box size="0.1 0.1 0.5"/>
      </geometry>
    </visual>
  </link>

  <joint name="arm_joint" type="revolute">
    <parent link="base_link"/>
    <child link="arm_link"/>
    <origin xyz="0 0 0.25" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
    <limit lower="-1.57" upper="1.57" effort="10" velocity="
        1.0"/>
  </joint>
</robot>
```

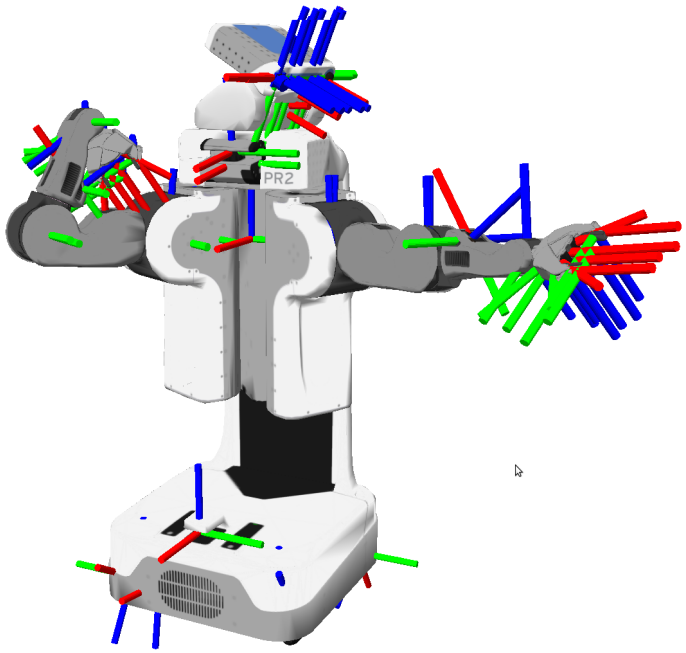
TF Tree of a Simple Arm Robot



- `base_link` is the root frame
- `arm_link` is connected through a revolute joint
- Joint origin offset: $(0, 0, 0.25)$

URDF to TF: Robot Description Publisher

- `robot_state_publisher` is a ROS 2 package used to publish the state of a robot's URDF model to **tf2**.
- It publishes the URDF description and joint states to enable visualization and interaction.
- `robot_state_publisher` uses the URDF specified by the parameter robot_description and the joint positions from the topic joint_states.



10

[10] https://wiki.ros.org/robot_state_publisher

AI in Simulation

AI & Simulation: Synthetic Data

Simulation is now the primary "classroom" for Artificial Intelligence.

- **Reinforcement Learning (RL):** An agent performs millions of trials in simulation to learn complex tasks (e.g., a bipedal robot walking).
- **Safety:** RL often involves thousands of "crashes"—simulation makes this free.
- **Synthetic Data:** Generating millions of labeled images for computer vision from a 3D engine (e.g., Unity/Unreal).

Gym + MuJoCo in Robotics and Reinforcement Learning

Gym (OpenAI Gym)

- Standardized toolkit for reinforcement learning environments
- Provides API: `reset()`, `step()`, `render()`
- Large collection of benchmark tasks

MuJoCo (Multi-Joint dynamics with Contact)

- High-performance physics engine for continuous control

Integration

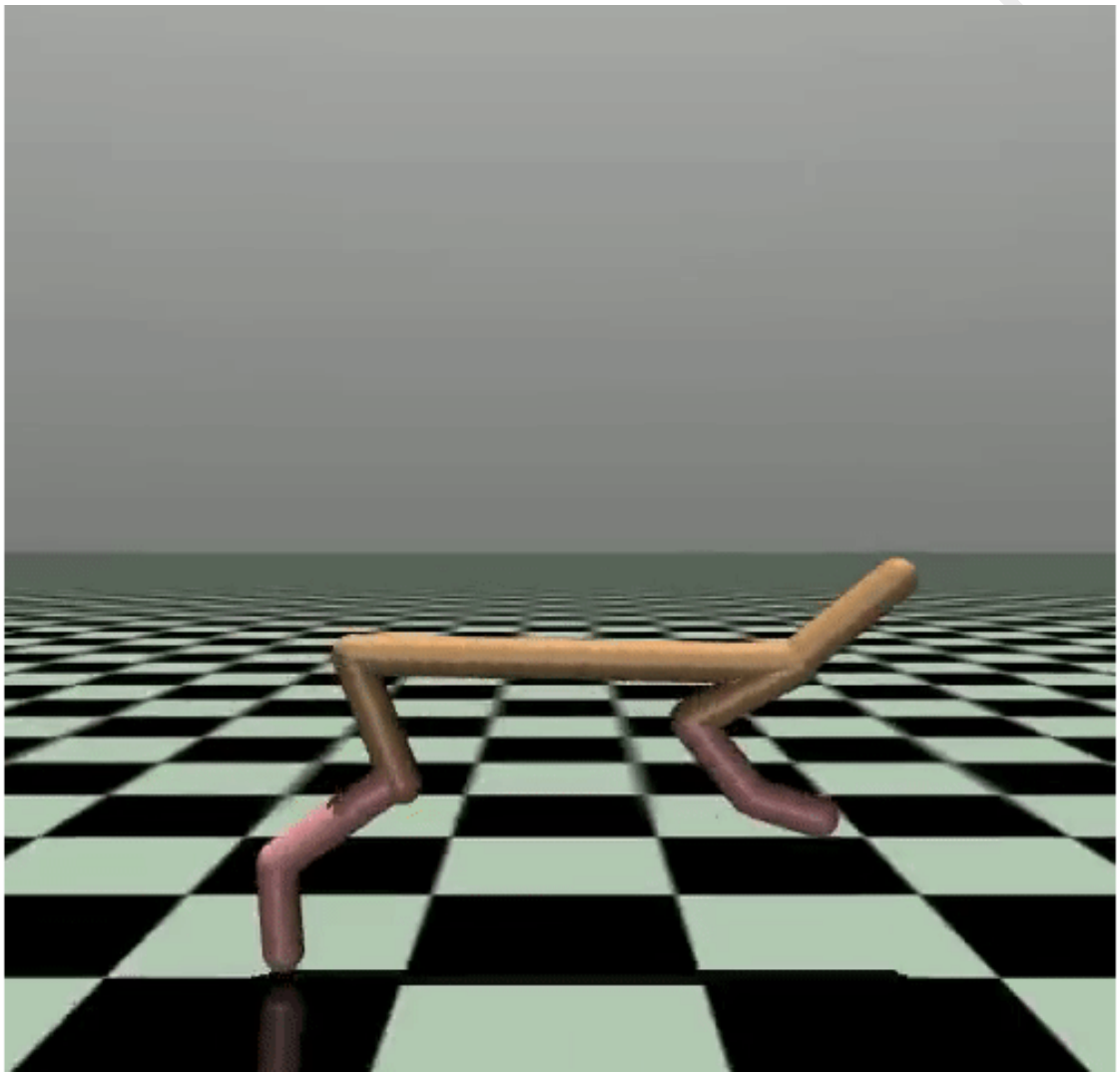
- Gym provides the **interface and environment structure**
- MuJoCo provides the **underlying physics simulation**
- Enables realistic training of agents in continuous action spaces

Practical RL Training Loop with Gym + MuJoCo

```
import gym
env = gym.make("HalfCheetah-v4")
obs = env.reset()
for step in range(max_steps):
  action = policy(obs) # neural network output
  obs, reward, done, info = env.step(action)
  agent.store(obs, action, reward) # collect experience
  if done: obs = env.reset()
  if step % train_freq == 0:
    agent.update() # update policy (PPO, SAC, etc.)
```

- **Observation:** Joint positions, velocities, and contact forces
- **Action:** Torque commands for each actuator (continuous, e.g., -1 to 1)
- **Reward:** Usually custom—forward velocity + energy penalty + survival bonus

HalfCheetah in Mujoco



[11]

[11] https://github.com/NickKaparinos/OpenAI-Gym-Projects/blob/master/MuJoCo/HalfCheetah/results/cheetah.gif

Beyond Basic Simulation: Domain Randomization

A key challenge: Sim-to-Real Transfer (what works in simulation often fails on real hardware).

Domain Randomization

Randomize simulation parameters during training to make the policy robust to reality gaps:

- **Physical parameters:** Mass, friction, damping, actuator strength
- **Latency and noise:** Sensor noise, control delay, observation dropout
- **Environment:** Gravity, ground friction, object positions

Result

A policy trained with sufficient randomization often transfers *zero-shot* to the real robot without fine-tuning.

Synthetic Data for Computer Vision

Modern perception systems are trained on **fully synthetic** datasets generated from 3D engines.

Advantages over real data:

- Perfect ground truth labels (depth, normals, segmentation)
- Infinite variety (lighting, textures, camera angles)
- No privacy or safety concerns
- Generate millions of frames in hours

Popular tools:

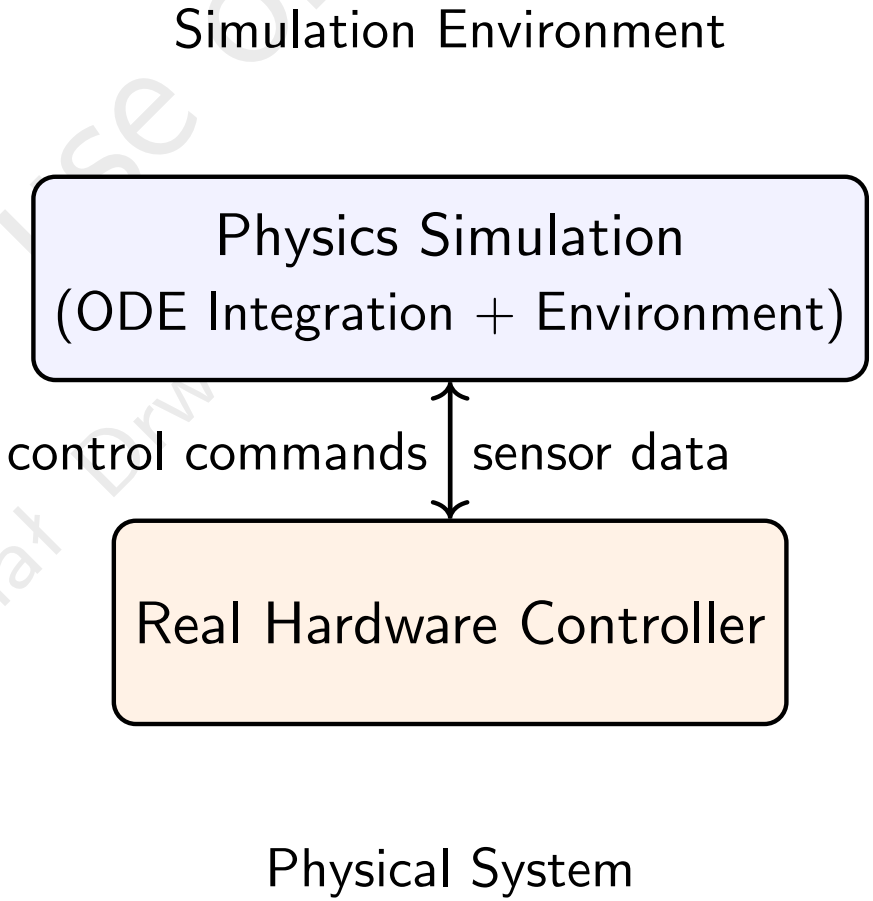
- NVIDIA Isaac Sim
- Microsoft AirSim (drones/cars)
- CARLA (autonomous driving)

HIL and DT

Hardware-in-the-Loop (HIL)

How do we test a flight controller without crashing a plane?

- **Concept:** The controller (real hardware) is connected to a computer running the ODE simulation in *real-time*.
- **The Loop:**
  1. Simulation sends sensor data to the hardware.
  2. Hardware computes control actions.
  3. Simulation updates physics based on actions.

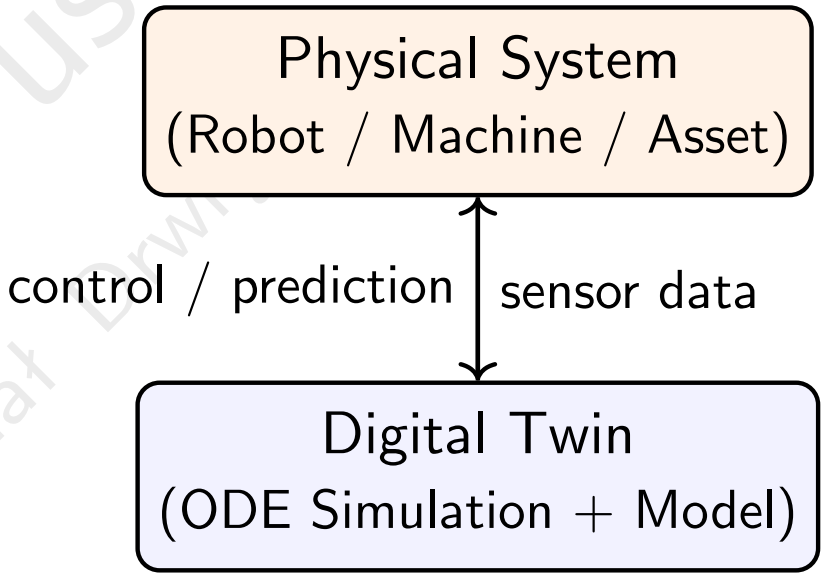


Benefit: Extreme safety. You can simulate failures (engine loss, wind gusts,

Digital Twins: The Living Model

A Digital Twin is a virtual replica of a physical asset that evolves over time.

- **Dynamic Sync:** Unlike a standard simulation, a Digital Twin is constantly updated with real-time data from IoT sensors.
- **Predictive Maintenance:** Using an ODE solver to "fast-forward" the system and predict failures.



Simulation vs Digital Twin

Simulation

- Starts from an assumed initial state
- Runs independently from reality
- No real-time data integration
- Used for:
  - design validation
  - testing algorithms
  - training (RL, control)
- Answers: *"What could happen?"*

Key Property

Offline, open-loop system

Digital Twin

- Initialized and updated with real sensor data
- Continuously synchronized with physical system
- Uses real-time feedback loop
- Used for:
  - monitoring
  - prediction
  - anomaly detection
- Answers: *"What is happening and what will happen?"*

Key Property

Closed-loop, real-time system

Where Simulation Fails (and how we fix it)

Simulation issues:

- Idealized contacts (real friction is stochastic)
- No actuator dynamics (motors have inertia, delays)
- Perfect state estimation (real sensors have noise/bias)
- Simplified collision geometry (meshes vs primitives)

Mitigations:

- Domain randomization (already discussed)
- System identification (fit sim parameters to real data)
- Hybrid sim-to-real (fine-tune on real robot for few episodes)
- Learned residual dynamics (augment sim with neural net)

Summary

The end

Thank you for your attention!