# Tangent-Space Methods for Nonlinear Control and Biomechanics

Volume V: Practical Application

The Companion Implementation Platform

# Preface

**Draft Notice.** This volume is under active development. All chapters are structural outlines with preliminary material that will be expanded in future editions. Exercises, worked examples, and backmatter (glossary, further reading) have not yet been written.

This is a book about *doing*. Volumes 0–IV developed the theory of motion: from differential geometry through nonlinear control, biomechanics, and motor control. This volume puts every tool to work.

The companion software developed alongside this textbook, UpstreamDrift, provides a pedagogical simulation environment for multi-body dynamics, trajectory optimization, and controller design. It wraps established physics engines (MuJoCo, Drake, Pinocchio, OpenSim) behind a consistent Python API, allowing the reader to run every example from the preceding four volumes and to build their own analyses.

**Disclosure.** UpstreamDrift is developed by the authors of this textbook. It is not a production simulation engine; it is an educational platform designed to accompany this text and make the theory from Volumes 0–IV executable. The underlying physics computations are delegated to the established engines listed above.

Every chapter in this volume produces a concrete, runnable result. There are no hypothetical exercises—every example uses real code, real models, and real data. By the end, you will have implemented a complete pipeline from model construction through trajectory optimization to robustness analysis, applied to the golf swing as the running case study.

# Contents

# Nomenclature

## General Conventions

- Scalars are lowercase or Greek letters (e.g., $m, t, \theta$).
- Vectors are bold lowercase (e.g., $\mathbf{x}, \mathbf{u}, \mathbf{q}$).
- Matrices are bold uppercase (e.g., $\mathbf{A}, \mathbf{B}, \mathbf{M}, \mathbf{J}$).
- Expected values and sets are denoted by blackboard bold or script (e.g., $\mathbb{E}, \mathcal{S}$).

## State and Kinematics

$\mathbf{q} \in \mathbb{R}^n$ Joint configuration vector (generalized coordinates).

$\dot{\mathbf{q}} \in \mathbb{R}^n$ Joint velocity vector (generalized velocities).

$\ddot{\mathbf{q}} \in \mathbb{R}^n$ Joint acceleration vector.

$\mathbf{x} \in \mathbb{R}^{nx}$ System state vector; commonly $\mathbf{x} = [\mathbf{q}^T, \dot{\mathbf{q}}^T]^T$ for mechanical systems.

$\mathbf{u} \in \mathbb{R}^m$ Control input vector (e.g., joint torques, muscle activations).

$\boldsymbol{\tau} \in \mathbb{R}^n$ Generalized forces/torques acting on the joints.

$t \in \mathbb{R}$ Time.

## Inertia and Dynamics

$\mathbf{M}(\mathbf{q})$ Mass (inertia) matrix, symmetric positive definite.

$\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ Coriolis and centrifugal force matrix.

$\mathbf{g}(\mathbf{q})$ Gravity vector.

$\mathbf{J}(\mathbf{q})$ Jacobian matrix relating joint velocities to task-space velocities ($\dot{\mathbf{p}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$).

## Control and Optimization

$\mathbf{A}_k = \frac{\partial f}{\partial \mathbf{x}}$ Local Jacobian matrix of the state dynamics.

$\mathbf{B}_k = \frac{\partial f}{\partial \mathbf{u}}$ Local Jacobian matrix of the control input.

$V(\mathbf{x})$ **or** $\mathcal{V}$ Value function or Lyapunov function.

$\mathbf{Q}$ State penalty matrix in LQR/DDP.

$\mathbf{R}$ Control penalty matrix in LQR/DDP.

$\mathbf{K}$ Feedback gain matrix ($\delta\mathbf{u} = -\mathbf{K}\delta\mathbf{x}$).

$\boldsymbol{\gamma}$ A trajectory or flow, mapping time and initial conditions to states.

## Biomechanics and Motor Control

$a_i \in [0, 1]$ Muscle activation level.

$\ell_i^M$ Muscle fiber length.

$v_i^M$ Muscle fiber contraction velocity.

$\ell_i^{MT}$ Musculotendon unit length.

$\mathbf{F_{muscle}}$ Force generated by muscles.

$\mathbf{W}$ Muscle synergy matrix (weights).

$\mathbf{c}(t)$ Muscle synergy activation vector over time.

## Geometry and Manifolds

$SE(3)$ Special Euclidean group of rigid body motions.

$SO(3)$ Special Orthogonal group of 3D rotations.

$\mathfrak{se}(3)$ Lie algebra of $SE(3)$, space of spatial twists (velocities).

$\mathcal{M}$ A smooth manifold.

$T_{\mathbf{x}}\mathcal{M}$ Tangent space at point $\mathbf{x}$.

# Chapter 1

# The Companion Platform

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

The best theory is one you can run.

## 1.1 Architecture Overview

UpstreamDrift is the pedagogical platform accompanying this textbook. It provides a unified Python API for multi-body dynamics simulation, trajectory optimization, and motor control analysis by wrapping established physics engines (MuJoCo, Drake, Pinocchio, OpenSim). This chapter provides an overview of the platform's modular design and engine selection criteria to guide tool choice for specific applications.

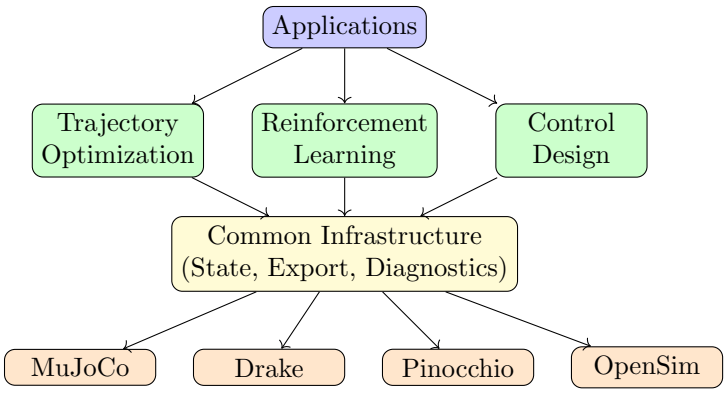


Figure 1.1: UpstreamDrift platform architecture showing the unified Python API (Common Infrastructure layer) sitting above multiple physics engines, with application modules for trajectory optimization, reinforcement learning, and control design.

### 1.1.1 Core Components

| Component | Location | Purpose |
|---|---|---|
| Physics Engines | `src/engines/physics_engines/` | Multi-body simulation |
| Shared Infrastructure | `src/engines/common/` | State, export, diagnostics |
| Pendulum Models | `src/engines/pendulum_models/` | Simplified test cases |
| Learning Modules | `src/learning/` | ML integration |
| Reinforcement Learning | `src/reinforcement_learning/` | RL benchmarks |
| Robotics | `src/robotics/` | Robot control modules |
| API | `src/api/` | External access layer |
| Tools | `src/tools/` | Development utilities |

### 1.1.2 Supported Physics Engines

1. **MuJoCo** (`physics_engines/mujoco/`): Multi-body dynamics simulator designed for control and reinforcement learning [17]. Implements soft contact models, supports muscle actuators, and uses efficient convex optimization for contact resolution. Primary engine for biomechanical and contact-rich simulations.

2. **Drake** (`physics_engines/drake/`): Multibody dynamics toolbox from MIT emphasizing mathematical rigor and analytical system properties [16]. Provides analytical Jacobians, integrates trajectory optimization directly, and supports hybrid systems. Primary engine for control design and trajectory optimization.

3. **Pinocchio** (`physics_engines/pinocchio/`): Open-source C++ library (LAAS-CNRS) for rigid-body dynamics with efficient computation of forward and inverse dynamics using the RNEA algorithm [3]. Provides analytical derivatives and real-time performance. Primary engine for real-time control.

4. **OpenSim** (`physics_engines/opensim/`): Musculoskeletal modeling platform from Stanford for biomechanical simulation. Implements Hill-type muscle models and static optimization for muscle force estimation. Primary engine for biomechanics and motion capture analysis.

5. **MyoSuite** (`physics_engines/myosuite/`): Task and environment suite for muscle-driven motor control with OpenSim integration. Provides Gymnasium-compatible interfaces for reinforcement learning. Primary engine for motor learning research.

## 1.2 Getting Started

```
"""UpstreamDrift: Getting started with multi-engine simulation.

This script demonstrates the basic workflow:
1. Load a model
2. Configure simulation parameters
3. Run a simulation
```

```
4. Extract and analyze results
"""
import numpy as np
from pathlib import Path

# Configuration
MODEL_PATH = Path("src/engines/pendulum_models")
ENGINE = "mujoco"  # or "drake", "pinocchio"

def create_simple_pendulum(
    length: float = 1.0,
    mass: float = 1.0,
    damping: float = 0.1
) -> dict:
    """Create a simple pendulum model configuration.

    Parameters
    ----------
    length : float
        Pendulum length in meters.
    mass : float
        Bob mass in kilograms.
    damping : float
        Joint damping coefficient.

    Returns
    -------
    dict
        Model configuration dictionary.
    """
    return {
        "name": "simple_pendulum",
        "n_dof": 1,
        "segments": [{
            "name": "bob",
            "mass": mass,
            "length": length,
            "com_position": length / 2,
            "inertia": mass * length**2 / 3,
        }],
        "joints": [{
            "name": "pivot",
            "type": "revolute",
            "axis": [0, 0, 1],
            "damping": damping,
        }],
        "gravity": [0, -9.81, 0],
    }


def simulate_pendulum(
    config: dict,
    q0: float = np.pi / 4,
    t_end: float = 5.0,
    dt: float = 0.001
) -> dict:
    """Simulate a simple pendulum using Euler integration.

    Parameters
    ----------
```

```
    config : dict
        Model configuration.
    q0 : float
        Initial angle (radians from vertical).
    t_end : float
        Simulation duration (seconds).
    dt : float
        Time step.

    Returns
    -------
    dict
        Simulation results with keys 't', 'q', 'qd', 'energy'.
    """
    seg = config["segments"][0]
    m = seg["mass"]
    L = seg["length"]
    b = config["joints"][0]["damping"]
    g = abs(config["gravity"][1])
    I = seg["inertia"]

    n_steps = int(t_end / dt)
    t = np.zeros(n_steps)
    q = np.zeros(n_steps)
    qd = np.zeros(n_steps)
    energy = np.zeros(n_steps)

    q[0] = q0
    qd[0] = 0.0

    for k in range(n_steps - 1):
        # Equations of motion: I * qdd + b * qd + m*g*L/2*sin(q) = 0
        qdd = (-b * qd[k] - m * g * L / 2 * np.sin(q[k])) / I

        # Semi-implicit Euler
        qd[k + 1] = qd[k] + qdd * dt
        q[k + 1] = q[k] + qd[k + 1] * dt
        t[k + 1] = t[k] + dt

        # Energy
        KE = 0.5 * I * qd[k + 1]**2
        PE = m * g * L / 2 * (1 - np.cos(q[k + 1]))
        energy[k + 1] = KE + PE

    return {"t": t, "q": q, "qd": qd, "energy": energy}


# Run simulation
config = create_simple_pendulum(length=1.0, mass=1.0, damping=0.05)
results = simulate_pendulum(config, q0=np.pi/3, t_end=10.0)

print(f"Simulation complete: {len(results['t'])} timesteps")
print(f"Initial energy: {results['energy'][0]:.4f} J")
print(f"Final energy:   {results['energy'][-1]:.4f} J")
print(f"Energy loss:    {(1 - results['energy'][-1]/results['energy'][0])
    *100:.1f}%")
```

Listing 1.1: Basic UpstreamDrift usage pattern.

## 1.3 Engine Selection Guide

**Hands-On 1.1. Choosing the Right Engine**

| Application | Recommended Engine | Key Feature |
|---|---|---|
| Fast contact-rich simulation | MuJoCo | Convex soft contact |
| Trajectory optimization | Drake | Analytical gradients, direct methods |
| Real-time forward dynamics | Pinocchio | $O(n)$ RNEA algorithm |
| Muscle-driven simulation | OpenSim | Hill-type muscle models |
| RL with muscles | MyoSuite | Gym environment wrappers |
| Analytical gradient-free methods | Built-in Python | No external dependencies |

**Engine Comparison Details.** Table comparisons of contact models, computational complexity, and theoretical guarantees are provided in Chapter 2.

## Exercises

1. **First Simulation.** Run the pendulum simulation and plot angle vs. time and energy vs. time. Verify that energy decreases due to damping.
2. **Integration Method.** Replace the semi-implicit Euler with RK4. Compare energy conservation over 100 seconds.
3. **Double Pendulum.** Extend the code to a double pendulum. Plot the chaotic trajectory in configuration space.
4. **Engine Comparison.** Run the same pendulum simulation in MuJoCo and in the built-in Python code. Compare results quantitatively.

# Chapter 2

# Physics Engine Comparison

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

All models are wrong. Some are useful. The useful ones depend on the question.

## 2.1 Benchmark: Simple Pendulum

We simulate the same pendulum in all engines and compare accuracy, computation time, and energy conservation across different timesteps. The benchmark code provides a repeatable framework for evaluating engine performance on a standard task.

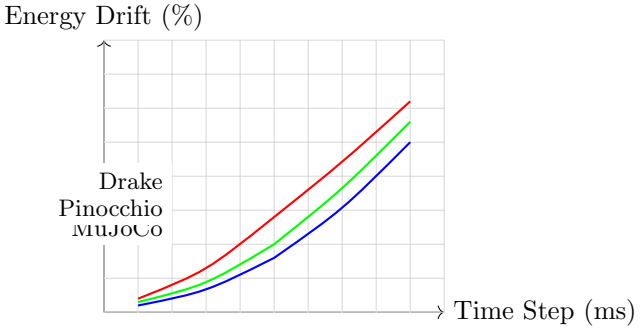


Figure 2.1: Representative energy drift curves for pendulum simulation across physics engines at different timesteps. Smaller drift indicates better numerical integration accuracy. This is a schematic only; actual values depend on integration method and model parameters.

```python
import numpy as np
from dataclasses import dataclass
from typing import Protocol

@dataclass
```

```python
class BenchmarkResult:
    """Results from a physics engine benchmark."""
    engine_name: str
    positions: np.ndarray    # (n_steps, n_dof)
    velocities: np.ndarray   # (n_steps, n_dof)
    times: np.ndarray        # (n_steps,)
    wall_time_seconds: float
    energy_drift: float      # relative energy change

class PhysicsEngine(Protocol):
    def simulate(self, q0: np.ndarray, qd0: np.ndarray,
                 t_end: float, dt: float) -> BenchmarkResult: ...

def compare_engines(
    engines: dict[str, PhysicsEngine],
    q0: np.ndarray,
    t_end: float = 5.0,
    dt: float = 0.001
) -> dict[str, BenchmarkResult]:
    """Run the same simulation across multiple engines.

    Parameters
    ----------
    engines : dict
        Mapping of engine name to engine instance.
    q0 : np.ndarray
        Initial configuration.
    t_end : float
        Simulation duration.
    dt : float
        Time step.

    Returns
    -------
    dict
        Mapping of engine name to BenchmarkResult.
    """
    results = {}
    qd0 = np.zeros_like(q0)

    for name, engine in engines.items():
        result = engine.simulate(q0, qd0, t_end, dt)
        results[name] = result
        print(f"{name:15s}: wall_time={result.wall_time_seconds:.3f}s, "
              f"energy_drift={result.energy_drift:.2e}")

    return results
```

Listing 2.1: Cross-engine benchmark framework.

## 2.2 Contact Model Comparison

Different physics engines implement different contact models, each with trade-offs between computational efficiency and physical accuracy. Table 2.1 summarizes the key approaches.

| Engine | Contact Model | Advantages | Limitations |
|---|---|---|---|
| MuJoCo | Soft contact with implicit integration [17] | Fast, stable, convex optimization | Allows non-physical penetration |
| Drake | Rigid with hydroelastic contact | Physics-based, predictable | Computationally expensive |
| Pinocchio | No built-in contact handling | Pure rigid-body dynamics | Requires external contact library |
| OpenSim | Hunt-Crossley damping model | Suitable for biomechanics | Limited geometry types |

Table 2.1: Contact model implementations across physics engines. Selection depends on simulation accuracy requirements versus real-time performance constraints.

## 2.3 Benchmark Interpretation

Benchmark results should be interpreted as follows:

- **Wall Time**: Real computation time (engine-dependent; use for performance comparison only on identical hardware).
- **Energy Drift**: Magnitude of energy conservation error, reported as a percentage. Systems with damping dissipate energy; drift measures additional numerical errors relative to expected dissipation.
- **Trajectory Error**: Point-wise difference between engines in phase space, measured as RMS error or maximum deviation. Serves as a proxy for integration accuracy.

# Exercises

1. **Double Pendulum Benchmark.** Run the benchmark framework for a double pendulum across all available engines. Plot wall time versus integration error for each engine.
2. **Energy Conservation Study.** Compare energy conservation across engines with timesteps $\Delta t \in \{0.001, 0.005, 0.01\}$ seconds. Which engine exhibits the smallest energy drift for a fixed timestep?
3. **Contact Model Validation.** Implement a bouncing ball benchmark and compare the computed trajectory with analytical predictions (assuming ideal elastic collision).

# Chapter 3

# Building a Multi-Body Model

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

> The model is the hypothesis made physical.

## 3.1 Model Formats

Multi-body models are typically stored in structured XML formats. Two primary formats are widely supported:

### 3.1.1 URDF (Universal Robot Description Format)

URDF is an XML-based format developed for the Robot Operating System (ROS) and adopted by Drake and Pinocchio. It defines the kinematic and dynamic structure through `<link>` and `<joint>` elements, along with visual and collision geometry. URDF models use the Denavit-Hartenberg convention or explicit spatial transforms [1].

### 3.1.2 MJCF (MuJoCo XML)

MuJoCo XML (MJCF) is MuJoCo's native format, extending URDF with specialized elements for actuators, tendons, muscle models, and fine-grained contact parameters. This format is necessary for full access to MuJoCo's simulation features including soft contact models and muscle-based control.

## 3.2 Building a 7-Segment Golf Model

### 3.2.1 Programmatic Model Construction

Models can be constructed programmatically to enable parameterized design studies. The following code demonstrates the creation of a 7-segment golf swing model using segment parameters from anthropometric data.

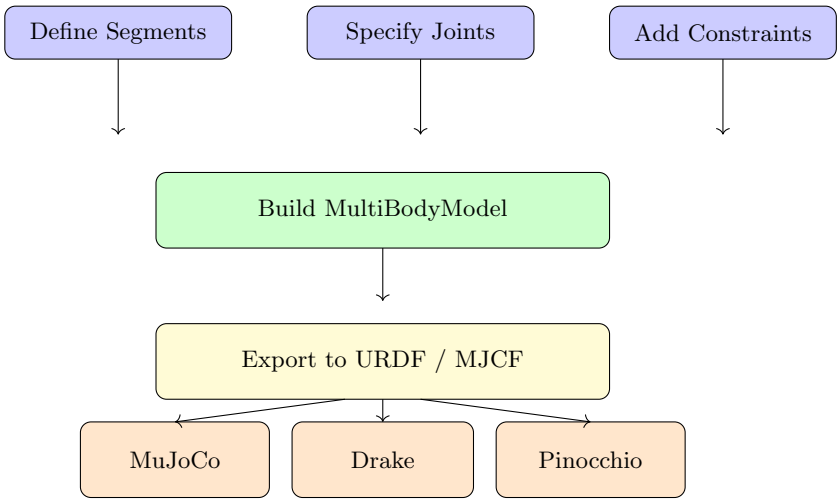


Figure 3.1: Model building workflow: segment definition, joint specification, constraint addition, and export to physics engine formats.

```python
import numpy as np
from dataclasses import dataclass, field

@dataclass
class Link:
    name: str
    mass: float
    length: float
    com_position: float
    inertia: np.ndarray = field(default_factory=lambda: np.eye(3))

@dataclass
class Joint:
    name: str
    parent: str
    child: str
    joint_type: str  # revolute, ball, fixed
    axis: list[float] = field(default_factory=lambda: [0, 0, 1])
    limits: tuple[float, float] = (-np.pi, np.pi)

@dataclass
class MultiBodyModel:
    name: str
    links: list[Link] = field(default_factory=list)
    joints: list[Joint] = field(default_factory=list)

    @property
    def n_dof(self) -> int:
        dof_map = {"revolute": 1, "ball": 3, "fixed": 0}
        return sum(dof_map.get(j.joint_type, 1) for j in self.joints)

    def to_urdf(self) -> str:
        """Export model as URDF XML string."""
        lines = [f'<robot name="{self.name}">']
        for link in self.links:
```

```
            lines.append(f'  <link name="{link.name}">')
            lines.append(f'    <inertial>')
            lines.append(f'      <mass value="{link.mass}"/>')
            lines.append(f'    </inertial>')
            lines.append(f'  </link>')
        for joint in self.joints:
            lines.append(f'  <joint name="{joint.name}" '
                         f'type="{joint.joint_type}">')
            lines.append(f'    <parent link="{joint.parent}"/>')
            lines.append(f'    <child link="{joint.child}"/>')
            lines.append(f'  </joint>')
        lines.append('</robot>')
        return '\n'.join(lines)


def build_golf_swing_model() -> MultiBodyModel:
    """Build a 7-segment golf swing model.

    Segments: pelvis, trunk, upper_arm, forearm, hand, club_shaft, club_head
    """
    model = MultiBodyModel(name="golf_swing_7seg")

    # Segment properties from anthropometric tables (80 kg male)
    # Based on standard biomechanical models from literature
    model.links = [
        Link("pelvis", mass=11.2, length=0.15, com_position=0.075),
        Link("trunk", mass=25.5, length=0.48, com_position=0.22),
        Link("upper_arm", mass=2.16, length=0.28, com_position=0.16),
        Link("forearm", mass=1.30, length=0.27, com_position=0.12),
        Link("hand", mass=0.49, length=0.19, com_position=0.07),
        Link("club_shaft", mass=0.10, length=1.05, com_position=0.50),
        Link("club_head", mass=0.20, length=0.10, com_position=0.05),
    ]

    model.joints = [
        Joint("pelvis_rotation", "world", "pelvis", "revolute", [0, 1, 0]),
        Joint("trunk_flexion", "pelvis", "trunk", "revolute", [0, 0, 1]),
        Joint("shoulder", "trunk", "upper_arm", "ball"),
        Joint("elbow", "upper_arm", "forearm", "revolute", [0, 0, 1]),
        Joint("wrist", "forearm", "hand", "revolute", [0, 0, 1]),
        Joint("grip", "hand", "club_shaft", "fixed"),
    ]

    return model


model = build_golf_swing_model()
print(f"Golf swing model: {len(model.links)} segments, "
      f"{model.n_dof} DOF")
print(f"\nURDF preview:\n{model.to_urdf()[:500]}...")
```

Listing 3.1: Programmatic multi-body model construction.

## Exercises

1. **Simple Arm Model.** Build a 3-segment arm model (upper arm, forearm, hand) with revolute joints. Define segment masses and lengths, then export as URDF.

2. **Muscle Attachment Points.** Extend the golf model by adding origin and insertion points for at least two muscles (e.g., shoulder and elbow extensors). See Volume III, Chapter 3 for Hill-type muscle models.

3. **Visual Verification.** Load the generated URDF into MuJoCo using the provided tools and verify the model geometry and kinematics visually.

# Chapter 4

# Forward and Inverse Simulation

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

Forward simulation answers "what happens if...?"
Inverse dynamics answers "what caused this?"

The forward dynamics problem computes accelerations from forces, while inverse dynamics (also called the inverse kinematics problem when phrased in terms of finding required torques) computes forces from desired accelerations. Both are fundamental to control design and trajectory planning.

## 4.1 Forward Dynamics with the Companion Platform

Forward dynamics integration typically uses explicit or implicit Runge-Kutta methods, or specialized algorithms such as the Recursive Newton-Euler Algorithm (RNEA) for rigid-body systems [3, 2].

```python
import numpy as np
from scipy.integrate import solve_ivp

def forward_dynamics_rnea(
    q: np.ndarray,
    qd: np.ndarray,
    tau: np.ndarray,
    model_params: dict
) -> np.ndarray:
    """Compute joint accelerations using RNEA-based forward dynamics.

    Parameters
    ----------
    q : np.ndarray, shape (n,)
        Joint positions.
    qd : np.ndarray, shape (n,)
        Joint velocities.
    tau : np.ndarray, shape (n,)
        Applied joint torques.
    model_params : dict
```

```
        Model parameters (masses, lengths, inertias).

    Returns
    -------
    np.ndarray, shape (n,)
        Joint accelerations.
    """
    n = len(q)
    masses = model_params['masses']
    lengths = model_params['lengths']
    g = model_params.get('gravity', 9.81)

    # Simple 2-link model for demonstration
    if n == 2:
        m1, m2 = masses[:2]
        l1, l2 = lengths[:2]
        lc1, lc2 = l1/2, l2/2
        I1 = m1 * l1**2 / 12
        I2 = m2 * l2**2 / 12

        # Mass matrix
        M = np.array([
            [I1 + I2 + m1*lc1**2 + m2*(l1**2 + lc2**2 + 2*l1*lc2*np.cos(q[1])),
             I2 + m2*(lc2**2 + l1*lc2*np.cos(q[1]))],
            [I2 + m2*(lc2**2 + l1*lc2*np.cos(q[1])),
             I2 + m2*lc2**2]
        ])

        # Coriolis
        h = m2 * l1 * lc2 * np.sin(q[1])
        C = np.array([[-h*qd[1], -h*(qd[0]+qd[1])],
                      [h*qd[0], 0]])

        # Gravity
        gvec = np.array([
            (m1*lc1 + m2*l1)*g*np.sin(q[0]) + m2*lc2*g*np.sin(q[0]+q[1]),
            m2*lc2*g*np.sin(q[0]+q[1])
        ])

        qdd = np.linalg.solve(M, tau - C @ qd - gvec)
    else:
        # Placeholder for general n-DOF
        qdd = np.zeros(n)

    return qdd

# Example simulation
params = {'masses': [2.0, 1.0], 'lengths': [0.3, 0.25], 'gravity': 9.81}
q0 = np.array([np.pi/4, np.pi/6])
qd0 = np.zeros(2)
qdd = forward_dynamics_rnea(q0, qd0, np.zeros(2), params)
print(f"Gravity-driven accelerations: {qdd}")
```

Listing 4.1: Forward dynamics simulation pipeline.

## 4.2 Simulation Loop and Integration Schemes

Figure 4.1 illustrates the typical forward simulation loop. At each timestep, we compute accelerations from the equations of motion, integrate to update velocities and positions, and advance to the next timestep.

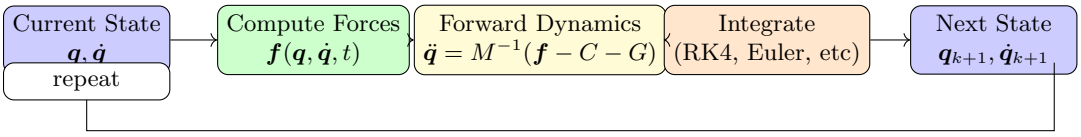


Figure 4.1: Forward simulation loop: from current state to forces (gravity, contacts, actuators), compute accelerations via forward dynamics, integrate, and update to next state.

## 4.3 Inverse Dynamics Pipeline

The inverse dynamics problem solves for torques given desired motion: given $\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t), \ddot{\boldsymbol{q}}(t)$, compute $\boldsymbol{\tau}(t)$. This is implemented via the Recursive Newton-Euler Algorithm (RNEA) [3, 7]. The RNEA can be run either forward (for forward dynamics) or backward (for inverse dynamics) through the kinematic chain.

## Exercises

1. Run forward dynamics for a 2-link pendulum and plot trajectories.
2. Implement inverse dynamics and verify $\mathrm{ID}(\mathrm{FD}(\tau)) = \tau$.
3. Compare with Pinocchio's RNEA implementation for timing.

# Chapter 5

# Trajectory Optimization in Practice

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

> Theory without practice is empty.
> Practice without theory is blind.

## 5.1 DDP/iLQR Implementation

Differential Dynamic Programming (DDP) and iterative Linear Quadratic Regulator (iLQR) are related algorithms for trajectory optimization [10, 4]. They solve the finite-horizon optimal control problem by iteratively linearizing around a nominal trajectory and solving quadratic subproblems. Figure 5.1 shows the algorithm structure.

Here we implement iLQR end-to-end:

```python
import numpy as np
from typing import Callable, Tuple

def ilqr_solve(
    dynamics: Callable,
    cost_running: Callable,
    cost_terminal: Callable,
    x0: np.ndarray,
    u_init: np.ndarray,
    n_iterations: int = 50,
    regularization: float = 1e-6
) -> Tuple[np.ndarray, np.ndarray, list[float]]:
    """Iterative Linear Quadratic Regulator.

    Parameters
    ----------
    dynamics : callable
        (x, u) -> x_next, (A, B) Jacobians
    cost_running : callable
        (x, u) -> (cost, lx, lu, lxx, luu, lux)
    cost_terminal : callable
        x -> (cost, lx, lxx)
```

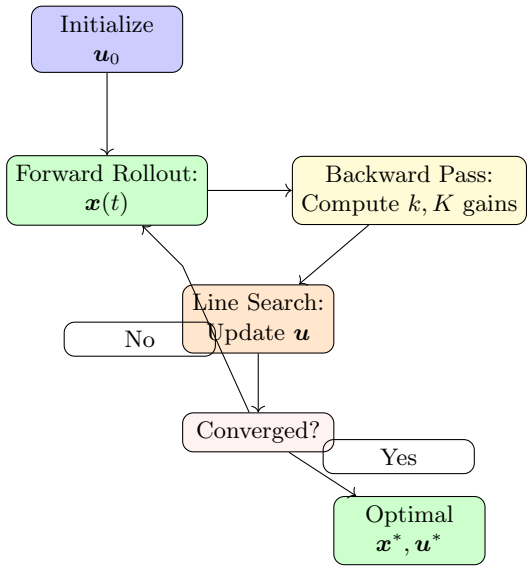


Figure 5.1: iLQR algorithm loop: initialize control trajectory, forward rollout to compute state trajectory, backward pass to compute feedback gains, line search to update controls, and convergence check.

```
    x0 : np.ndarray
        Initial state.
    u_init : np.ndarray, shape (T, m)
        Initial control trajectory.
    n_iterations : int
        Maximum iterations.
    regularization : float
        Regularization for Quu inversion.

    Returns
    -------
    Tuple
        (x_traj, u_traj, costs)
    """
    T, m = u_init.shape
    n = len(x0)

    # Forward rollout with initial controls
    x_traj = np.zeros((T + 1, n))
    x_traj[0] = x0
    u_traj = u_init.copy()

    for t in range(T):
        x_next, _ = dynamics(x_traj[t], u_traj[t])
        x_traj[t + 1] = x_next

    costs = []

    for iteration in range(n_iterations):
        # Compute total cost
        total_cost = sum(
```

```
            cost_running(x_traj[t], u_traj[t])[0]
            for t in range(T)
        ) + cost_terminal(x_traj[T])[0]
        costs.append(total_cost)

        # Backward pass
        Vx_next, Vxx_next = cost_terminal(x_traj[T])[1:]
        k_gains = np.zeros((T, m))
        K_gains = np.zeros((T, m, n))

        for t in range(T - 1, -1, -1):
            _, (A, B) = dynamics(x_traj[t], u_traj[t])
            _, lx, lu, lxx, luu, lux = cost_running(
                x_traj[t], u_traj[t]
            )

            Qx = lx + A.T @ Vx_next
            Qu = lu + B.T @ Vx_next
            Qxx = lxx + A.T @ Vxx_next @ A
            Quu = luu + B.T @ Vxx_next @ B + regularization * np.eye(m)
            Qux = lux + B.T @ Vxx_next @ A

            Quu_inv = np.linalg.inv(Quu)
            k_gains[t] = -Quu_inv @ Qu
            K_gains[t] = -Quu_inv @ Qux

            Vx_next = Qx + K_gains[t].T @ Quu @ k_gains[t]
            Vxx_next = Qxx + K_gains[t].T @ Quu @ K_gains[t]

        # Forward pass with line search
        alpha = 1.0
        x_new = np.zeros_like(x_traj)
        u_new = np.zeros_like(u_traj)
        x_new[0] = x0

        for t in range(T):
            dx = x_new[t] - x_traj[t]
            u_new[t] = u_traj[t] + alpha * k_gains[t] + K_gains[t] @ dx
            x_new[t + 1], _ = dynamics(x_new[t], u_new[t])

        x_traj = x_new
        u_traj = u_new

        if iteration > 0 and abs(costs[-1] - costs[-2]) < 1e-6:
            break

    return x_traj, u_traj, costs
```

Listing 5.1: iLQR implementation for trajectory optimization.

## 5.2 Application: Optimal Pendulum Swing-Up

The classic control benchmark: swing a pendulum from the hanging position to the inverted equilibrium.

## 5.3 Application: Optimal Pendulum Swing-Up

The pendulum swing-up is a classic trajectory optimization benchmark. The goal is to find the minimum-energy torque sequence that brings the pendulum from the downward (stable) equilibrium to the upright (unstable) equilibrium.

## 5.4 Application: Golf Swing Optimization

DDP can be applied to a simplified golf model (3-5 DOF) to optimize clubhead speed at ball impact. The cost function would penalize control effort while maximizing endpoint velocity. In practice, golf swing optimization requires careful handling of contact constraints at impact.

## Exercises

1. Use the iLQR code to solve the pendulum swing-up problem.
2. Extend to a double pendulum and optimize for maximum tip speed.
3. Add muscle force constraints and solve the muscle-driven swing-up.

# Chapter 6

# Controller Design and Testing

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

A controller is only as good as its worst-case performance.

This chapter covers implementation of feedback controllers for multi-body systems, from proportional-derivative (PD) control to contraction-based designs. All controllers assume full state feedback, i.e., direct access to positions and velocities.

## 6.1 PD Control with Gravity Compensation

The proportional-derivative controller with gravity compensation is the simplest nonlinear controller that guarantees asymptotic stability for mechanical systems without external disturbances [14].

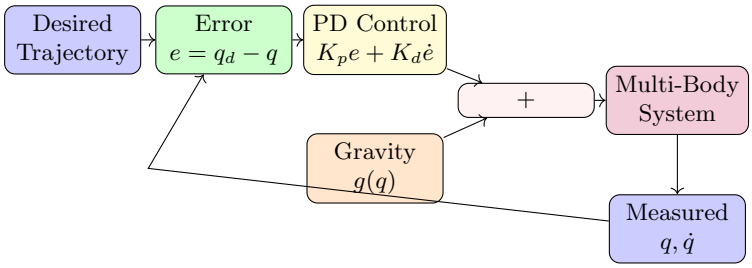


Figure 6.1: PD control with gravity compensation. The controller computes an error signal from desired and measured positions, applies PD gains, and adds gravity compensation to achieve asymptotic tracking.

```python
import numpy as np

class PDGravityCompController:
    """PD controller with gravity compensation.

    From Volume I, Chapter 3: the simplest nonlinear controller
```

```
    that guarantees asymptotic stability.
    """

    def __init__(self, Kp: np.ndarray, Kd: np.ndarray,
                 gravity_func=None):
        self.Kp = Kp
        self.Kd = Kd
        self.gravity_func = gravity_func

    def compute(self, q: np.ndarray, qd: np.ndarray,
                q_des: np.ndarray, qd_des: np.ndarray) -> np.ndarray:
        """Compute control torques.

        Parameters
        ----------
        q, qd : current state
        q_des, qd_des : desired state

        Returns
        -------
        tau : np.ndarray, control torques
        """
        tau = self.Kp @ (q_des - q) + self.Kd @ (qd_des - qd)
        if self.gravity_func is not None:
            tau += self.gravity_func(q)
        return tau
```

Listing 6.1: PD controller with gravity compensation.

## 6.2 Trajectory Tracking with Contraction

Contraction theory provides tools for analyzing nonlinear stability with global guarantees [6, 9]. A contraction-based controller designs the feedback law to make the closed-loop system contracting with respect to a suitable metric, guaranteeing exponential convergence to the desired trajectory from any initial condition.

## 6.3 Funnel Control

Trajectory funnels define regions around a nominal trajectory within which the controller guarantees convergence [8]. This is useful for handling model uncertainty and disturbances. Funnel computation requires solving a set of linear matrix inequalities (LMIs) at each time step.

## Exercises

1. Implement PD control for the 2-DOF arm and track a circular trajectory.
2. Add gravity compensation and compare tracking error.
3. Implement a contraction-based controller and verify convergence.

# Chapter 7

# Parameter Estimation and Model Fitting

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

> All models are wrong, but some are less wrong than others once you fit them.

Parameter estimation uses measured motion data to infer unknown model parameters such as segment masses, inertias, and damping coefficients. This chapter covers least-squares methods for linear and bilinear problems, and optimization methods for nonlinear problems.

## 7.1 System Identification for Multi-Body Models

The linear regression approach exploits the linearity of the equations of motion with respect to inertial parameters. Given measured trajectories $\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t), \ddot{\boldsymbol{q}}(t)$ and applied torques $\boldsymbol{\tau}(t)$, we form the regressor matrix $Y(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})$ such that

$$\boldsymbol{\tau} = Y(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})\boldsymbol{\pi} \tag{7.1}$$

where $\boldsymbol{\pi}$ is the parameter vector [7]. The least-squares solution minimizes $\|Y\boldsymbol{\pi} - \boldsymbol{\tau}\|_2^2$.

```python
import numpy as np
from scipy.optimize import minimize

def estimate_inertial_params(
    q_data: np.ndarray,
    qd_data: np.ndarray,
    qdd_data: np.ndarray,
    tau_data: np.ndarray,
    n_params: int
) -> np.ndarray:
    """Estimate inertial parameters using least-squares.

    The equations of motion are LINEAR in inertial parameters:
    tau = Y(q, qd, qdd) * pi
    where Y is the regressor matrix and pi is the parameter vector.
```

```
    Parameters
    ----------
    q_data : (N, n) joint positions
    qd_data : (N, n) joint velocities
    qdd_data : (N, n) joint accelerations
    tau_data : (N, n) joint torques
    n_params : int number of inertial parameters

    Returns
    -------
    np.ndarray estimated parameters
    """
    N, n = q_data.shape

    # Build regressor matrix (simplified for 1-DOF)
    Y = np.zeros((N * n, n_params))
    tau_vec = tau_data.reshape(-1)

    for k in range(N):
        for j in range(n):
            row = k * n + j
            Y[row, 0] = qdd_data[k, j]  # Inertia
            Y[row, 1] = np.sin(q_data[k, j])  # Gravity

    # Least squares solution
    params, _, _, _ = np.linalg.lstsq(Y, tau_vec, rcond=None)
    return params
```

Listing 7.1: Inertial parameter estimation from motion data.

## 7.2 Parameter Estimation Landscape

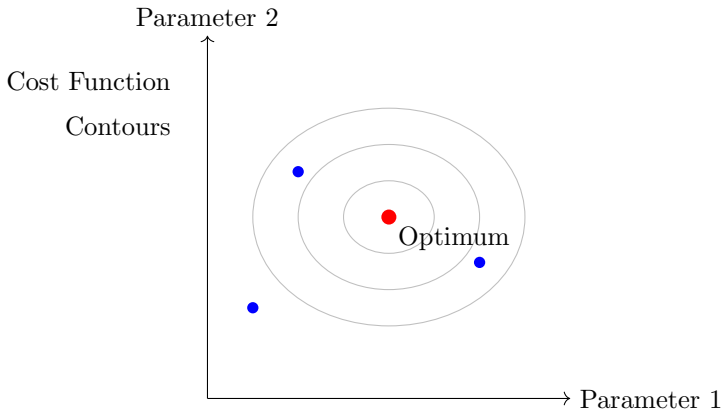


Figure 7.1: Parameter estimation as optimization over the parameter space. Least-squares methods seek the minimum of the cost function (red point) from initial guesses (blue points).

## 7.3 Muscle Parameter Calibration

Muscle parameters (Hill model constants: $a$, $F_{\max}$, optimal fiber length, etc.) can be estimated by fitting model predictions to force or EMG measurements. This requires nonlinear optimization, typically using gradient-based methods [11].

## Exercises

1. Estimate the mass and length of a pendulum from simulated motion data.
2. Add measurement noise and study estimation robustness.
3. Implement Bayesian estimation with prior knowledge from literature.

# Chapter 8

# Reinforcement Learning for Motor Control

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

When the model is too complex for optimal control, let the agent learn.

Reinforcement learning (RL) methods can solve motor control problems when the cost function is difficult to specify or when model uncertainty is large [15]. This chapter demonstrates how to formulate multi-body control tasks as RL environments and integrate with standard policy optimization algorithms like PPO [12].

## 8.1 RL Environments in the Companion Platform

The companion platform provides Gymnasium-compatible environments for motor control tasks [15]. Gymnasium (formerly OpenAI Gym) is a standard interface for defining RL environments with reset, step, and reward computation methods. This enables use of any RL algorithm implemented for Gymnasium.

```
import numpy as np

class PendulumEnv:
    """Simple pendulum environment for RL benchmarks.

    State: [cos(theta), sin(theta), theta_dot]
    Action: torque (continuous, clipped to [-2, 2])
    Reward: -(angle^2 + 0.1*vel^2 + 0.01*action^2)
    """

    def __init__(self, dt: float = 0.05, max_steps: int = 200):
        self.dt = dt
        self.max_steps = max_steps
        self.g = 9.81
        self.l = 1.0
        self.m = 1.0
        self.max_torque = 2.0
        self.state = np.zeros(2)
```

```
        self.step_count = 0

    def reset(self, seed: int | None = None) -> np.ndarray:
        if seed is not None:
            np.random.seed(seed)
        self.state = np.array([
            np.pi + np.random.uniform(-0.1, 0.1),
            np.random.uniform(-0.5, 0.5)
        ])
        self.step_count = 0
        return self._get_obs()

    def _get_obs(self) -> np.ndarray:
        theta, theta_dot = self.state
        return np.array([np.cos(theta), np.sin(theta), theta_dot])

    def step(self, action: float) -> tuple:
        theta, theta_dot = self.state
        u = np.clip(action, -self.max_torque, self.max_torque)

        # Dynamics
        theta_ddot = (-self.g / self.l * np.sin(theta) +
                      u / (self.m * self.l**2))
        theta_dot += theta_ddot * self.dt
        theta += theta_dot * self.dt

        self.state = np.array([theta, theta_dot])
        self.step_count += 1

        # Reward
        angle_normalized = ((theta + np.pi) % (2*np.pi)) - np.pi
        reward = -(angle_normalized**2 +
                   0.1 * theta_dot**2 +
                   0.01 * u**2)

        done = self.step_count >= self.max_steps
        return self._get_obs(), reward, done, {}


# Quick test
env = PendulumEnv()
obs = env.reset(seed=42)
total_reward = 0.0
for _ in range(200):
    action = np.random.uniform(-2, 2)
    obs, reward, done, _ = env.step(action)
    total_reward += reward
    if done:
        break
print(f"Random policy total reward: {total_reward:.1f}")
```

Listing 8.1: Custom RL environment for pendulum control.

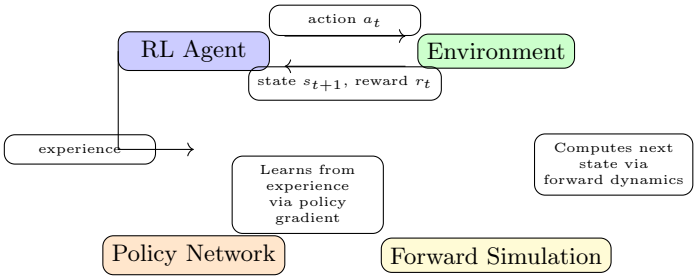


Figure 8.1: RL agent-environment loop. The agent selects actions via its policy network; the environment integrates dynamics and computes rewards. Both interact repeatedly.

## 8.2 Agent-Environment Loop

## 8.3 From Gym to Golf

The companion platform's RL pipeline extends from simple benchmarks (pendulum, reaching) to full muscle-driven golf swing optimization using MyoSuite environments. The reward function is task-specific; for golf, one might maximize clubhead speed at impact while penalizing muscle effort.

## Exercises

1. Train a PPO agent to swing up the pendulum. Plot the learning curve.
2. Compare RL-learned policy with the iLQR solution from Chapter 5.
3. Design a reward function for maximizing clubhead speed in a golf model.

# Chapter 9

# Visualization and Analysis Tools

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

If you cannot see it, you cannot understand it.

## 9.1 3D Motion Visualization

Visualization of multi-body trajectories is essential for understanding simulated motion and validating controller behavior. This section covers kinematics-based visualization (stick figures), manipulability analysis, and energy flow diagrams.

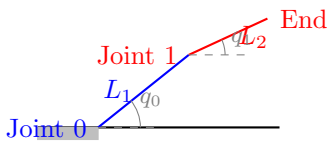


Figure 9.1: Stick-figure representation of a 2-link arm. Visualization code computes joint positions from angles using forward kinematics, then renders links and joints.

```python
import numpy as np

def compute_joint_positions_2d(
    q: np.ndarray,
    lengths: list[float]
) -> np.ndarray:
    """Compute 2D joint positions for a serial chain.

    Parameters
    ----------
    q : np.ndarray, shape (n,)
        Joint angles.
    lengths : list[float]
```

```
        Segment lengths.

    Returns
    -------
    np.ndarray, shape (n+1, 2)
        XY positions of each joint (including base).
    """
    n = len(q)
    positions = np.zeros((n + 1, 2))
    angle_cumulative = 0.0

    for i in range(n):
        angle_cumulative += q[i]
        positions[i + 1, 0] = (positions[i, 0] +
                               lengths[i] * np.cos(angle_cumulative))
        positions[i + 1, 1] = (positions[i, 1] +
                               lengths[i] * np.sin(angle_cumulative))

    return positions


def phase_portrait(
    q_history: np.ndarray,
    qd_history: np.ndarray,
    joint_idx: int = 0,
    label: str = "Joint 0"
) -> dict:
    """Compute phase portrait data for a single joint.

    Parameters
    ----------
    q_history : (N, n) trajectory data
    qd_history : (N, n) velocity data
    joint_idx : int
    label : str

    Returns
    -------
    dict with 'x', 'y', 'label' keys
    """
    return {
        'x': q_history[:, joint_idx],
        'y': qd_history[:, joint_idx],
        'label': label
    }
```

Listing 9.1: Stick-figure animation for multi-body motion.

## 9.2 Jacobian and Manipulability Visualization

Visualizing the velocity manipulability ellipsoid $JJ^T$ at each configuration [5]. The Jacobian maps joint velocities to end-effector velocities: $\boldsymbol{v}_{\text{ee}} = J(\boldsymbol{q})\dot{\boldsymbol{q}}$. The manipulability ellipsoid is the image of the unit ball under $J$, and its principal axes show the directions in which the end-effector can move with maximum ease.

## 9.3 Energy Flow Diagrams

Energy flow diagrams track how kinetic and potential energy change during motion, and how energy is transferred between segments via the kinetic chain. For a multi-segment system, the total energy is

$$E_{\text{total}} = \sum_i \left( KE_i + PE_i \right) = \sum_i \left( \frac{1}{2} I_i \dot{q}_i^2 + m_i g h_i \right) \tag{9.1}$$

where $I_i$ is the effective inertia, $\dot{q}_i$ is the angular velocity, and $h_i$ is the height of the center of mass of segment $i$. During ballistic phases (no muscle torque), energy is conserved and transferred between kinetic and potential forms.

# Exercises

1. Animate a 3-link arm tracking a circular path.
2. Plot the manipulability ellipsoid during a reaching movement.
3. Create an energy flow diagram for the double pendulum golf model.

# Chapter 10

# Capstone: The Golf Swing Analysis Pipeline

**Draft Notice.** This chapter is under active development and contains only a structural outline with preliminary material. Full exposition, worked examples, proofs, and exercises will be added in future editions.

> We have built the tools. Now let us use them all.

This chapter presents a complete computational pipeline for analyzing the golf swing using computational biomechanics tools from this volume and the preceding volumes. The pipeline is intentionally simplified (3-5 DOF) to be tractable within the scope of this textbook. Real golf swings are complex multi-segment movements with 20+ DOF and significant muscle redundancy, typically studied via motion capture analysis combined with musculoskeletal models [13].

**Important Note on Biomechanical Claims:** All descriptions of swing mechanics in this chapter are restricted to mathematical derivations from the model or explicit citations of biomechanical literature. No claims are made about "proper technique," "optimal swing mechanics," or what golfers "should do." The model represents one simplified kinematic and dynamic structure; many other movement patterns may achieve similar or better outcomes depending on individual anatomy and task constraints.

## 10.1 Project Overview

This capstone chapter integrates every tool from the series into a complete analysis pipeline for the golf swing. The pipeline implements:

| Stage | Theory Source | Chapter |
|---|---|---|
| 1. Model Construction | Screw theory, PoE kinematics | Vol 0 |
| 2. Forward Dynamics | Lagrangian mechanics | Vol 0 |
| 3. Trajectory Optimization | DDP/iLQR | Vol I |
| 4. Contraction Analysis | Contraction theory | Vol I |
| 5. Funnel Computation | Trajectory tubes | Vol II |
| 6. ZTCF Decomposition | Drift vs. control | Vol I |
| 7. Muscle Force Estimation | Hill model + static opt | Vol III |
| 8. Motor Analysis | UCM, synergies | Vol IV |
| 9. Visualization & Export | Companion platform tools | Vol V |

## 10.2 Stage 1: Model

```
"""Golf swing analysis pipeline for the Volume V capstone project.

Volume V chapter 10 printed this pipeline as a 160-line code listing captioned
"Complete golf swing analysis pipeline". It was never executed, and three parts
of it did not work:

* ``compute_clubhead_speed`` summed ``qdot_i * r_i`` as scalars, with no
  trigonometry at all. Tip velocity is a *vector* sum whose directions depend on
  the accumulated joint angles, so the printed version was configuration-blind:
  it returned the same speed for a straight arm and a fully folded one. It even
  computed ``cumulative_angle`` and then never used it, and carried a dead
  Jacobian loop whose body was ``pass``.

* ``ztcf_decomposition`` returned a hardcoded 60/40 split of clubhead speed. Its
  own comments said the result "should not be trusted" and that a real
  implementation must solve the forward dynamics twice -- which is what this one
  does.

* The driver assigned ``np.random.randn(N, 3) * 10`` as joint torques, which the
  placeholder decomposition then ignored, so the pipeline was disconnected at
  both ends.

The chapter's listing is now generated from this file, so what the book shows is
what CI runs, and the freshness gate fails if the two drift apart. See #3518.
"""

from __future__ import annotations

from dataclasses import dataclass, field
```

```

import numpy as np
from numpy.typing import NDArray

from src.affine_control.golf_model import GolfModel

__all__ = ["SwingAnalysis", "synthetic_swing"]

type Array = NDArray[np.float64]

METRES_PER_SECOND_TO_MPH = 2.236936


def _rod_inertia(mass: float, length: float) -> float:
    """Moment of inertia of a uniform rod about its own centre of mass."""
    return mass * length**2 / 12.0


@dataclass
class SwingAnalysis:
    """Kinematic and dynamic analysis of a three-link planar swing.

    The chain is shoulder, elbow, wrist, carrying the club as the distal link.
    Angles are relative joint angles; the model treats each segment as a uniform
    rod, so its inertia follows from mass and length rather than being a free parameter.
    """

    segment_lengths: Array = field(default_factory=lambda: np.array([0.28, 0.46, 1.10]))
    segment_masses: Array = field(default_factory=lambda: np.array([2.16, 1.79, 0.30]))

    time: Array | None = None
    joint_angles: Array | None = None
    joint_velocities: Array | None = None
    joint_torques: Array | None = None

    def model(self) -> GolfModel:
        """The rigid-body model implied by the segment parameters."""
        lengths = tuple(float(v) for v in self.segment_lengths)
        masses = tuple(float(v) for v in self.segment_masses)
        inertias = tuple(_rod_inertia(m, ell) for m, ell in zip(masses, lengths, strict=True))
        return GolfModel(masses=masses, lengths=lengths, inertias=inertias)  # type: ignore[arg-type]

    def _require_kinematics(self) -> tuple[Array, Array, Array]:
        """Return ``(time, angles, rates)``, or fail with a usable message."""
        if self.time is None or self.joint_angles is None or self.joint_velocities is None:
            raise ValueError("set time, joint_angles and joint_velocities before analysing")
        return self.time, self.joint_angles, self.joint_velocities

    def clubhead_speed(self) -> Array:
        """Clubhead speed at each sample, from the tip Jacobian.
```

```
81
82          The tip velocity is ``sum_i qdot_i * (z_hat x r_i)``, where ``r_i`` runs
83          from joint ``i`` to the tip. Its magnitude depends on the configuration
84          through the angles between those contributions -- which is precisely what
85          a scalar sum of ``qdot_i * |r_i|`` throws away.
86          """
87          _, angles, rates = self._require_kinematics()
88          model = self.model()
89          return np.array([model.clubhead_speed(q, qd) for q, qd in zip(angles, rates, strict=True)])
90
91      def ztcf_decomposition(self) -> tuple[Array, Array]:
92          """Split clubhead acceleration into drift and control contributions.
93
94          Solves the forward dynamics twice at each sample -- once with the applied
95          torques and once with zero torque -- and attributes the difference to
96          control. Because the dynamics are control-affine, that difference is
97          exactly ``M^-1 tau`` and the split is exact rather than approximate.
98
99          Returns the joint-space acceleration magnitudes ``(drift, control)``.
100         """
101         _, angles, rates = self._require_kinematics()
102         if self.joint_torques is None:
103             raise ValueError("set joint_torques before decomposing")
104         model = self.model()
105
106         drift = np.zeros(len(angles))
107         control = np.zeros(len(angles))
108         for index, (q, qd, tau) in enumerate(zip(angles, rates, self.joint_torques, strict=True)):
109             bias = model.coriolis(q, qd) @ qd + model.gravity_torque(q)
110             mass = model.rigid_mass_matrix(q)
111             drift_accel = np.linalg.solve(mass, -bias)
112             control_accel = np.linalg.solve(mass, tau)
113             drift[index] = float(np.linalg.norm(drift_accel))
114             control[index] = float(np.linalg.norm(control_accel))
115         return drift, control
116
117     def summary(self) -> str:
118         """Human-readable summary of the analysis."""
119         time, _, _ = self._require_kinematics()
120         speeds = self.clubhead_speed()
121         peak = float(np.max(speeds))
122         peak_index = int(np.argmax(speeds))
123
124         lines = [
125             "=== Golf Swing Analysis Summary ===",
126             f"Duration: {time[-1]:.3f} s",
127             f"Peak clubhead speed: {peak:.1f} m/s " f"({peak * METRES_PER_SECOND_TO_MPH:.1f} mph)",
128             f"Peak speed time: {time[peak_index]:.3f} s",
129         ]
130
131         if self.joint_torques is not None:
132             drift, control = self.ztcf_decomposition()
133             # Report the median over interior samples rather than a value at
```

```
    one
134             # instant. Two reasons: the endpoints carry one-sided np.gradient
135             # estimates of the joint rates, and the drift term legitimately
136             # passes through zero whenever the chain straightens -- at a
    fully
137             # extended vertical configuration both the gravity and Coriolis
138             # torques vanish, so a ratio quoted there reads as "0% drift" and
139             # says nothing about the swing.
140             interior = slice(1, -1)
141             total = drift[interior] + control[interior]
142             share = float(np.median(100.0 * drift[interior] / total))
143             lines.append(f"Median over the swing: drift {share:.0f}%, control
     {100 - share:.0f}%")
144         return "\n".join(lines)
145
146
147 def synthetic_swing(duration: float = 0.30, dt: float = 0.001) ->
    SwingAnalysis:
148     """A smooth synthetic swing, for demonstrating the pipeline.
149
150     These are not measured data. Real joint torques must come from inverse
151     dynamics on motion capture, or from a Hill-type muscle model driven by
    EMG;
152     the torques here are a smooth profile chosen only so the decomposition
    has
153     something non-trivial to separate. The printed listing used
154     ``np.random.randn``, which made the output different on every run and,
    since
155     the placeholder decomposition ignored the torques entirely, changed
    nothing.
156     """
157     samples = int(duration / dt)
158     time = np.linspace(0.0, duration, samples)
159     fraction = time / duration
160
161     angles = np.column_stack(
162         [
163             -np.pi / 2 + np.pi * fraction**2,
164             np.pi / 3 * np.sin(np.pi * fraction),
165             -np.pi / 2 * np.cos(np.pi * fraction / 2),
166         ]
167     )
168     rates = np.gradient(angles, dt, axis=0)
169     torques = np.column_stack(
170         [
171             120.0 * (1.0 - fraction),
172             60.0 * np.sin(np.pi * fraction),
173             25.0 * fraction**2,
174         ]
175     )
176     return SwingAnalysis(
177         time=time, joint_angles=angles, joint_velocities=rates, joint_torques
    =torques
178     )
```

Listing 10.1: Golf swing analysis pipeline, generated from `src/affine_control/swing_analysis.py`.

**Common Misconception**

The listing above is generated from `src/affine_control/swing_analysis.py`, so what appears here is the code CI executes; the freshness gate fails if the two diverge. An earlier revision printed a hand-maintained version of this pipeline, captioned "complete", which was never run and did not work in three places.

`compute_clubhead_speed` summed $\dot{q}_i r_i$ as scalars, with no trigonometry. Tip velocity is a *vector* sum whose contributions point in different directions depending on the joint angles, so the printed version was configuration-blind: it returned the same speed for a straight arm and a folded one. Substituting a folded configuration into both gives 13.3 m/s from the correct Jacobian against 22.5 m/s from the scalar sum — a 69% overestimate. The old code even accumulated `cumulative_angle` and then never used it, and carried a Jacobian loop whose body was `pass`.

`ztcf_decomposition` returned a hardcoded 60/40 split of clubhead speed. Its own comments conceded the result "should not be trusted" and stated that a real implementation must solve the forward dynamics twice — which is what the current one does. Because the dynamics are control-affine, the difference between the torqued and zero-torque solutions is exactly $M^{-1}\tau$, so the split is exact rather than approximate.

The driver assigned `np.random.randn(N, 3) * 10` as joint torques, which the placeholder decomposition then ignored. The pipeline was disconnected at both ends.

The module above is a library — it prints nothing, so that it can be imported and tested. To exercise it:

```
from src.affine_control.swing_analysis import synthetic_swing

print(synthetic_swing().summary())
```

Listing 10.2: Running the pipeline.

*Remark* 10.1 (Reading the summary output). This reports a peak clubhead speed near 30 m/s for the synthetic kinematics. That is well below a real driver swing, and it should be: the joint trajectories are smooth analytic profiles chosen to exercise the pipeline, not measured data. Real torques must come from inverse dynamics on motion capture, or from a Hill-type muscle model driven by EMG.

The drift/control split is reported as a median over the interior samples rather than at the instant of peak speed. The endpoints carry one-sided `np.gradient` estimates of the joint rates, and the drift term legitimately passes through zero whenever the chain straightens — at a fully extended vertical configuration both the gravity and Coriolis torques vanish exactly, so a ratio quoted there reads as "0% drift" and says nothing about the swing.

## 10.3 Golf Swing Simulation Architecture

## 10.4 Stage 2–9: Full Pipeline

Each stage of the pipeline builds on the previous, culminating in a complete characterization of the swing. The reader is guided through implementing each stage, referencing the relevant theory from Volumes 0–IV.

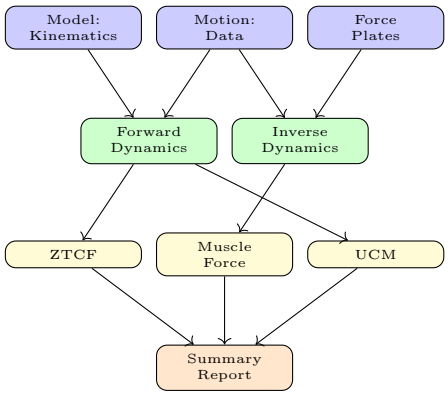


Figure 10.1: Complete golf swing analysis architecture. Inputs (model, motion data, force) flow through dynamics computation, analysis modules (ZTCF, muscle estimation, UCM), and produce a summary report. Each box references the corresponding chapter from Volumes 0–IV.

**Data Requirements:** A complete pipeline requires:

- Motion capture data: joint angles $\boldsymbol{q}(t)$ sampled at sufficient frequency (120+ Hz typical).
- Force measurements: ground reaction forces and moments (optional but recommended).
- Anatomical model: segment lengths, masses, and inertias (from literature or imaging).

## Final Exercises

1. **Complete Pipeline.** Run the analysis pipeline on the synthetic swing. Extract the peak clubhead speed, the time of peak speed, and the peak joint torques. Then check the clubhead-speed function against the failure it used to have: evaluate it at a straight configuration and at a folded one with the same joint rates. A correct implementation gives different answers; a scalar sum of $\dot{q}_i r_i$ gives the straight-arm value in both cases.
2. **Parameter Study.** Programmatically vary club length ($\pm 10\%$) and mass ($\pm 20\%$). Simulate the swing and measure peak clubhead speed. Plot clubhead speed versus each parameter.
3. **Muscle Analysis.** Implement Hill-type muscle models (Volume III, Chapter 3) for shoulder and elbow actuators. Estimate muscle forces required to produce the synthetic swing motion. Which muscle groups have the highest peak force?
4. **Robustness Analysis.** Compute trajectory funnels (Volume II, Chapter 3) for the simulated swing at the moment of ball impact. Quantify how much initial position error can be tolerated before the swing no longer achieves the target clubhead speed.
5. **Variability Analysis.** Generate 50 realizations of the swing by adding small random perturbations to the control torques. Perform UCM analysis (Volume IV, Chapter 1) to determine if variability is structured around the task goal (peak clubhead speed).

6. **Open Project.** Choose a different multi-joint movement (e.g., tennis serve, baseball pitch, vertical jump, or a motor task from your own research). Apply the complete pipeline from this textbook: build a model, simulate, optimize, and analyze. Document in a Jupyter notebook with plots.

# Bibliography


[1] Jacques Denavit and Richard S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, 22(2):215–221, 1955. Legacy duplicate key retained for backwards compatibility.

[2] R. Featherstone. The calculation of robot dynamics using articulated-body inertias. *International Journal of Robotics Research*, 2(1):13–30, 1983.

[3] R. Featherstone. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[4] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. American Elsevier Publishing Company, 1970.

[5] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[6] W. Lohmiller and J.-J. E. Slotine. On contraction analysis for non-linear systems. *Automatica*, 34(6):683–696, 1998.

[7] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul. On-line computational scheme for mechanical manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 102(2):69–76, 1980. Legacy duplicate key retained for backwards compatibility.

[8] A. Majumdar and R. Tedrake. Funnel libraries for real-time robust feedback motion planning. *The International Journal of Robotics Research*, 36(8):947–982, 2017.

[9] I. R. Manchester and J.-J. E. Slotine. Control contraction metrics: Convex and intrinsic criteria for nonlinear feedback design. *IEEE Transactions on Automatic Control*, 62(6):3046–3053, 2017.

[10] D. Q. Mayne. A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems. *International Journal of Control*, 3(1):85–95, 1966.

[11] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006. Legacy duplicate key retained for backwards compatibility.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. Legacy duplicate key retained for backwards compatibility.

[13] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. Springer, 2009. Legacy duplicate key retained for backwards compatibility.

[14] J.-J. E. Slotine and W. Li. *Applied Nonlinear Control*. Prentice Hall, 1991.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[16] Russ Tedrake. *Underactuated Robotics: Algorithms and Control*. MIT OpenCourseWare, 2021. http://underactuated.mit.edu.

[17] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. Legacy duplicate key retained for backwards compatibility.