Digital Twins: from simulation to a real boat

Dubins-like model, UDP communication and Luenberger observer

Antoine MORVAN

ENSTA

October 21, 2026

Outline

1. Digital twins
2. From the Dubins car to our boat
3. Practice 1: a model in simulation
4. Connecting to the real boat
5. The Luenberger observer

What is a digital twin?

A **digital twin** is a virtual copy of a real object or system, which stays **connected** to the real one.

1. **A physical system**: a robot, a car, a wind turbine, a factory...
2. **A virtual model**: a program describing how the system behaves
3. **A data link**: sensors send measurements to the model, and the model may send decisions back

Key idea: a simulation is a model. A digital twin is a model *kept in sync with reality*.

Model, shadow, twin

Level	Information flow	Example
Digital model	none	simulated robot arm on a laptop
Digital shadow	real $\rightarrow$ virtual	the simulated arm copies the real one
Digital twin	real $\leftrightarrow$ virtual	the simulation also predicts and sends commands

In this course we will build:

- first a **model** (Python simulation),
- then a **twin** connected to a real boat.

What is it used for?

We experiment on the **copy** instead of the real thing: real experiments are expensive, slow or dangerous.

- **Test safely**: try a risky manoeuvre on the virtual robot first
- **Predict**: anticipate wear and fix a part before it breaks
- **Monitor**: follow a machine from far away
- **Optimize**: try a thousand plans in minutes, keep the best
- **Understand**: estimate what sensors cannot measure directly

Examples: jet engines in flight, wind farms, factories, hospitals, cities...

The question a twin answers in robotics

"If I send this command, what will my robot do?"

To answer it, we need a **model** of how the robot moves.

Real robots are complicated, so we start with the simplest model that still captures something essential about motion: the **Dubins car**.

The Dubins car: the rules

A small vehicle that:

- only drives **forward** (no reverse),
- moves at **constant speed**,
- can **steer** left or right, but only up to a limit.

Consequence: it cannot turn on the spot. To make a U-turn it needs room to draw a curve, like a real car in a parking lot.

The Dubins car: the equations

State $X = (x, y, \theta)^\top$: position x, y and heading θ. The only control is the steering u:

$$\begin{cases} \dot{x} = v \cos\theta \\ \dot{y} = v \sin\theta \\ \dot{\theta} = u, \qquad -u_{\max} \leq u \leq u_{\max} \end{cases}$$

- v: constant speed, u: steering command
- $\dot{x}$ means "how fast x is changing"

Why start with it?

- **Simple**: a few lines of code
- **Realistic constraint**: wheeled robots and boats cannot move sideways
- **Classic question**: what is the shortest path between two positions *and* orientations?
  - Dubins (1957): it is always made of **straight lines and circular arcs**
- It is a **first step** towards the model of our boat.

The boat model

The state is the vector $X = (x,\ y,\ \theta)^\top$: position x, y and heading θ.
We control the two thrusters u_1 (left) and u_2 (right):

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases} \qquad \begin{cases} v = k_2\,\dfrac{u_1 + u_2}{2} \\ \omega = k_1\,(u_2 - u_1) \end{cases}$$

- v: forward speed (average of the thrusts)
- ω: rotation speed (difference of the thrusts)
- k_1, k_2: gains to be identified on the real boat

Plan of the practical session

1. **Simulation**: build the model of our boat with `roblib`
2. **Network**: learn how to talk to the boat with UDP
3. **Real boat**: compare model and measurements
4. **Observer**: combine both with a Luenberger observer for a better estimate

From equations to code: Euler integration

We cannot compute the continuous motion exactly, so we advance by **small time steps** dt:

$$x_{k+1} = x_k + dt\, v \cos\theta_k, \quad y_{k+1} = y_k + dt\, v \sin\theta_k, \quad \theta_{k+1} = \theta_k + dt\, \omega$$

- "New state = old state + small step $\times$ speed of change"
- Smaller dt: more accurate, but more computations

Simulating our boat

```
from roblib import *          # numpy, matplotlib helpers

k1, k2 = 1.0, 1.0             # gains (to be identified)

def f(x, u):                  # x = (x, y, theta), u = (u1, u2)
    v = k2 * (u[0] + u[1]) / 2
    w = k1 * (u[1] - u[0])
    return array([v*cos(x[2]), v*sin(x[2]), w])

dt = 0.05
x = array([0., 0., 0.])

ax = init_figure(-10, 10, -10, 10)
for k in range(400):
    u = array([1.0, 0.8])     # left, right thrust
    x = x + dt * f(x, u)      # Euler step
    clear(ax)
    draw_tank(x)              # draw the vehicle
    pause(0.01)
```

Check the exact drawing function names in your `roblib` version.

Exercises

1. Run the simulation with $u_1 = u_2$: what trajectory do you get?
2. Try $u_1 < u_2$, then $u_1 > u_2$: which way does the boat turn?
3. Try $u_1 = -u_2$: what happens?
4. Make the boat follow a square by changing (u_1, u_2) over time.

We now have a model. Is it a digital twin yet? No: it is not connected to a real boat.

Talking to the boat over the network

- The boat and the computer are connected by **Wi-Fi** and exchange **messages**.
- A **protocol** is a set of rules for sending these messages.
- **UDP** (User Datagram Protocol) is the simplest one:
  - you write an **address** (IP + port) and **send**
  - no handshake, no confirmation, no guarantee of arrival

	UDP	**TCP**
Analogy	Postcard	Registered letter
Delivery guaranteed?	No	Yes
Delay	Very low	Higher
Used for	Games, robot control	Web, emails, files

Why UDP for a robot? An old command is useless: we prefer fresh data to complete data.

UDP in practice (Python)

We need the boat's **IP address** (who?) and a **port** (which door?).

```
import socket

BOAT_IP, PORT = "192.168.1.50", 5005   # adapt

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # UDP
sock.settimeout(0.1)                    # never wait forever

sock.sendto(b"0.5,0.3", (BOAT_IP, PORT))   # command (u1, u2)

try:
    data, addr = sock.recvfrom(1024)       # measurement
    print(data.decode())
except socket.timeout:
    print("Nothing received")              # packets can be lost!
```

1. Messages are just **bytes**.
2. A message can be **lost or late**: the code must survive it.
3. This is the **real** $\leftrightarrow$ **virtual** link of our twin.

Model vs reality: the problem

We run the model with the same commands as the real boat and compare.

- The **model alone** drifts away: wind, current, thrusters never exactly equal, wrong $k_1, k_2 \ldots$
- The **sensor alone** is noisy and sometimes missing (lost packets).

Neither is satisfactory. Can we combine them?

Yes: this is the role of an **observer**.

The Luenberger observer: the idea

Predict, then correct:

1. **Predict**: use the model to guess where the boat should be
2. **Compare**: measurement minus prediction = error
3. **Correct**: shift the estimate by a fraction of this error

Analogy: walking in the dark with a map. You count your steps (model) and, from time to time, you spot a landmark (measurement) and adjust your position.

The Luenberger observer: the equations

$$\dot{\hat{X}} = \underbrace{f(\hat{X}, u)}_{\text{model}} + \underbrace{L\,(y - \hat{y})}_{\text{correction}}, \qquad \hat{y} = C\hat{X}$$

- $\hat{X}$: **estimate** of the state
- y: **measurement**, $\hat{y}$: what the model expects to measure
- L: the **gain**, the "trust" dial
  - large L: trust the sensor (fast, but noisy)
  - small L: trust the model (smooth, but drifts)

f is non-linear here: the observer is used in its non-linear form, which works well in practice with a well-chosen gain.

Observer in Python

Measurements: position (x, y) and heading θ, so $C = I$ and $L = \mathrm{diag}(l_1, l_1, l_2)$.

```
L = diag([2., 2., 2.])            # gains: to be tuned
xhat = array([0., 0., 0.])        # estimate

while True:
    u = array([u1, u2])           # same command sent to the boat
    sock.sendto(encode(u), (BOAT_IP, PORT))
    try:
        data, _ = sock.recvfrom(1024)
        y = decode(data)          # measured (x, y, theta)
        err = y - xhat
        err[2] = sawtooth(err[2])    # angle error in [-pi, pi]
    except socket.timeout:
        err = zeros(3)            # no measurement: model only
    xhat = xhat + dt * (f(xhat, u) + L @ err)
    # draw xhat, y, and compare
```

Experiments with the real boat

1. Run the model alone ($L = 0$): how fast does it drift?
2. Use the measurements alone: how noisy are they?
3. Increase L progressively: find a good compromise.
4. Unplug the sensor for a few seconds: what does the estimate do?
5. Identify k_1, k_2 with a known command: does the estimate improve?

Now it is a digital twin: model and real boat are connected, and the twin gives a better estimate than either one alone.

Summary

- A **digital twin** is a model connected to a real system.
- The **Dubins car** is a simple model; our boat uses two thrusters (u_1, u_2).
- **UDP** provides the real $\leftrightarrow$ virtual data link.
- The **Luenberger observer** fuses model and measurements: **predict, then correct**.

To go further: Kalman filter (optimal gain), more realistic boat dynamics, using the twin to plan trajectories.