INFO 4000
Informatics III

Data Science Specialization - Advanced

Week11 – Engineering Thinking with Digital Twins:
How to design, build and implement them!

Course Instructor

Jagannath Rao

raoj@uga.edu

Today

- We will expand into more details about the Digital Twin and its construction.
- We will look at some basic tools and structures required to build a Digital Twin of any system – tech and non-tech
- We shall understand some math approaches to address analytical solution of ODEs
- This is detailed input for the MP3 project you will do, which is the culmination of your INFORMATICS CERTIFICATE

Digital Twin Framework:
From Concept to Code

Building Digital Twins: Framework, Code & Concepts

- Ask:
  - What system are we modeling?
  - What are its inputs, outputs, and states?
  - What is the *goal* of the DT? (Predict? Optimize? Alert?)

- Example:
  - *Motor System*: Goal = Predict torque at 500 RPM
  - *Baking Oven*: Goal = Predict when food is done
  - *Banking system*: Goal = Predict fraud risk

Identify & Collect Physics / Rules (The “How”)

- What are the *fundamental laws* governing the system?
- Can they be written as equations?
- If not, can we approximate them with ML or PINNs?

Examples:

- DC Motor: Torque = K * I (current) — with friction term
- Oven: Heat balance: dT/dt = (Power - HeatLoss) / mass*c_p
- Bank: Fraud = (Amount > avg * 5) + (transactions/sec > 0.5) + (new device?)

Build the Core Model (Math + Code)

- Define classes for:
  - System (e.g., Motor Digital Twin)
  - Inputs, States, Outputs
  - Update rules (dynamics, equations, or ML logic)

```python
class MotorDT:
    def __init__(self, K_t=0.1, friction=0.01):
        self.K_t = K_t
        self.friction = friction
        self.speed = 0.0
        self.torque = 0.0

    def update(self, current, dt=0.01):
        # Torque = K_t * I - friction * speed
        self.torque = self.K_t * current - self.friction * self.speed
        # Speed update: d(speed)/dt = torque / J (inertia) - if needed
        self.speed += (self.torque / 1.0) * dt  # assume J=1 for simplicity
```

Implement Data Ingestion & Real-World Feedback Loop

- Where does data come from?
  - Sensors? Logs? APIs? User inputs?
- How do you ingest it?
  - From File, Stream data, from Database, MQTT, REST API

Example:

- Motor: Current sensor
- Bank: Transaction logs
- stream data to update your Digital Twin model.

```
# Pseudocode: Data Ingestion
def ingest_data(sensor_data):
    self.speed = sensor_data['speed']
    self.current = sensor_data['current']
    # Update model
    self.update(self.current)
```

Implement a PINN (If You Have Physics or Data)

- PINNs are *not required* — but *very powerful* if you have:

  → Real sensor data
  → Unknown or complex physics
  → No explicit equations

- You can *train a PINN* to approximate the physics — or learn behavioral patterns.

PINN Example:

  - Input: [current, speed, time]
  - Output: [torque, next speed]
  - Loss: MSE between predicted and actual
  - Train with sensor data
  - PINN can be *embedded in your class* — or used as a *black-box predictor*.

Add Visualization (Even If Not 3D CAD)

- Visualization is *critical* — even if you don't build CAD.
- Why? Because Digital Twins are meant to be *understood*, *debugged*, *trusted*.

Tools:

  - Plotly (for live graphs)
  - Matplotlib (for static plots)
  - Streamlit (for web dashboard)

Example:

  - **Motor:** Plot speed vs. time
  - **Bank:** Plot risk score over time

```
import matplotlib.pyplot as plt
plt.plot(self.time_series, self.speed_series)
plt.title("Motor Speed Over Time")
plt.show()
```

Deploy & Monitor — The Real-World Digital Twin Loop

DT is *not* just a model — it's a *system that runs in real-time*.

- Deploy to:
- Raspberry Pi?
- Local server?
- Cloud (AWS, Azure, GCP)

Monitor:

- Accuracy?
- Latency?
- Alerts? (e.g., "Motor overheating", "Fraud detected")

Feedback:

- Use predictions to *control* the system (e.g., turn off motor if overheating)

Iterate & Improve

Digital Twins are *not static* — they *learn*.

- Collect new data
- Retrain PINN (if applicable)
- Update rules
- Improve visualization

Example:

- If fraud model is wrong → retrain with more data

Why Is CAD / Visualization Still Important?

Even if you don't build 3D CAD, *conceptually*, Digital Twins need *visualization*.

- Because engineers and managers *need to see* what's happening.
- Because DTs are *used to make decisions* — not just studied.
- Example: In banking, you *see* the fraud risk trend — not just a number.
- In motors, you *see* when torque drops — not just a log.
- Even if you use a *simple graph* — it's *still visualization*.
- And in real-world DTs — you *will* have CAD or 3D visualization — even if it's just a "dashboard" or "simulation".

Summary of requirements for building a Digital Twin

- **Physical/Business System** (the "twin of")
- **Mathematical Model** (how it behaves — ODEs, rules, logic, etc.)
- **Real-time Sensor Data** (to update the twin)
- **Actuators/Interfaces** (to control or influence the real system)
- **Visualization & Feedback Loop** (to show state, predict, and guide decisions)
- **Update Mechanism** (how often does the model get updated?)

Python Class Approach

Class Design Philosophy:

- **State**: Variables that represent the system (temperature, position, inventory, etc.)
- **Behavior**: Methods that update the system (e.g., update(dt), apply_force(), process_order())
- **Inputs**: Sensor data, control signals, events
- **Outputs**: Predictions, diagnostics, visualizations
- **Interfaces**: Connect to real sensors/actuators (even if simulated)

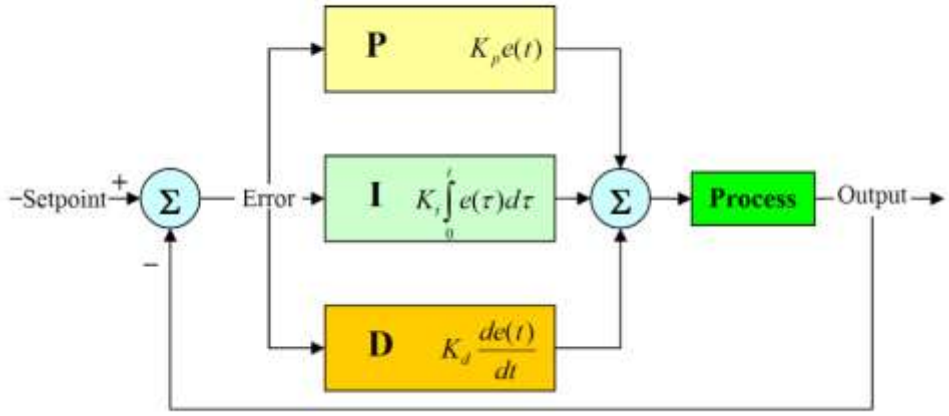
Example:

A class TemperatureDT with methods update(heat_rate, dt) and predict_next_temp()

Controlling the system is key:
What is a PID controller?

Proportional, Integral, Derivative (PID) controller

- In control systems engineering, a proportional-integral-derivative (PID) controller is essential.
- The PID controller"
  - continuously measure the difference between the desired controlled value and the present controlled value,
  - and it automatically modifies controlled system inputs to eliminate that difference.
- Setpoint versus actual value is the error and:
  - Proportional component (P) generates an output that is exactly proportionate to this value. Value is large if error is large. (proportion constant – Kp)
  - Integral component adds error over time and then multiplies it by the integral gain (Ki) and it removes any leftover error
  - Derivative component forecasts the inaccuracy in the future. By increasing the derivative gain (Kd) by the error's derivative over time, it produces a damping effect.



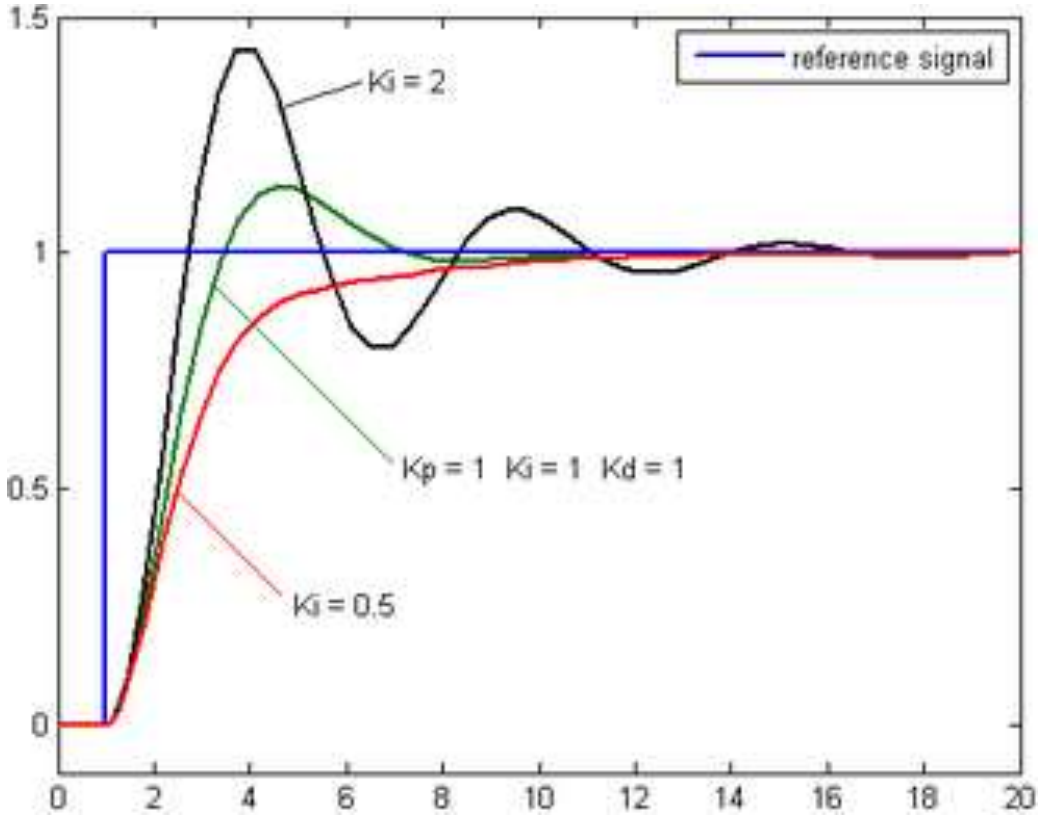
Easy way to implement programmatically

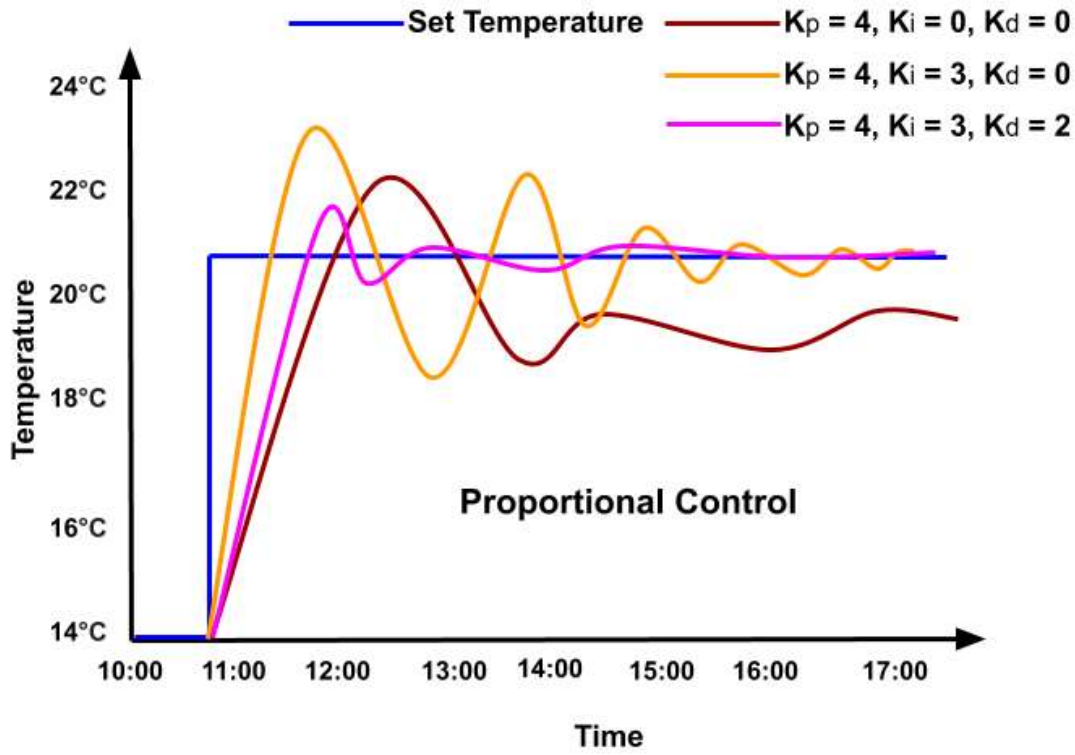
- Integral term:
  - $I_k = I_{k-1} + edt$
  - If we want I to be with a range: $I_k = clip(I_k + edt, I_{\max}\ , I_{\min}\ )$
- Derivative:
  - $\dot{\theta} = \frac{(\theta_k - \theta_{k-1})}{dt}$
  - The derivative term opposes the motion: $-K_d\dot{\theta}$ $(can\ be$ rate of change of e or signal)
- PID $= K_p e + K_i I_k - K_d\dot{\theta}$
- You can make tweaks based on the system:
  - for example, you can clip as shown above (do not have to do it always)
  - If the output is torque, we may want to limit it to the values the device is specified for
  - You will never get the correct values of coefficients K and have to tweak it till the system behaves the way it should.

```
e = r - y
integral += e*dt
deriv = (e - e_prev)/dt
u = Kp*e + Ki*integral + Kd*deriv
e_prev = e
```

```
e = r - y
integral += e*dt
deriv = (y - y_prev)/dt
u = Kp*e + Ki*integral - Kd*deriv
y_prev = y
```

Visual aid to see the ‘K’ values effects





Example of implementing a PID controller

Numerical approach to ODE:
RK4 method – simple approach

What is RK4 method and how to implement

- **RK4's Fundamental Operation:** The Runge-Kutta method is a numerical integrator. Its core calculation finds the next value of a variable, yn+1, based on its current value, yn, and its **first derivative** (dy/dt). It is inherently a first-order step process.

- RK4 **samples four slopes within the same time step** and blends them to get a much better estimate.

Think of $\dot{x} = f(x, u, t)$ and a step size dt.

1. **Slope 1 (start):**
$$k_1 = f(x_k, u_k)$$

2. **Slope 2 (midpoint, using k1):**
$$k_2 = f\left(x_k + \frac{dt}{2}k_1,\ u_k\right)$$

3. **Slope 3 (midpoint, using k2):**
$$k_3 = f(x_k + \frac{dt}{2}k_2,\ u_k)$$

4. **Slope 4 (end, using k3):**
$$k_4 = f(x_k + dt\ k_3,\ u_k)$$

5. $x_{k+1} = x_k + \frac{dt}{6}(k1 + 2k2 + 2k3 + k4)$ *Take the weighted avaerage*

Tiny scalar example

ODE: $y' = -2y$,step $dt = 0.1$, start $y_0 = 1$.

- $k_1 = f(y_0) = -2(1) = -2$
- $k_2 = f\ (y_0 + \frac{dt}{2}k_1) = -2(1 + 0.05 \cdot -2) = -2(0.9) = -1.8$
- $k_3 = f\ (y_0 + \frac{dt}{2}k_2) = -2(1 + 0.05 \cdot -1.8) = -2(0.91) = -1.82$
- $k_4 = f\ (y_0 + dt\ k_3) = -2(1 + 0.1 \cdot -1.82) = -2(0.818) = -1.636$

Blend:
$$y_1 = y_0 + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$= 1 + \frac{0.1}{6}(-2 - 3.6 - 3.64 - 1.636) \approx 0.818734.$$

Exact solution: $e^{-0.2} \approx 0.818731$

You **peek ahead** using tentative mid/end states to see how the slope changes.

@dataset decorator and its power

The very useful “@dataset” decorator in Python

- A decorator is a function that takes another function as an argument, adds some functionality to it, and then returns a new function (or the modified version of the original).
- **Data-centric:** @dataclass is best used for classes where the main purpose is to store state and data, rather than to define complex behaviors.
- It is ideal for creating simple data structures that function similarly to a typed dictionary.
- Decorator syntax: You use it by placing @dataclass on the line just before the class definition

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    name: str
    unit_price: float
    quantity_on_hand: int = 0

# You can now create instances without manually writing __init__
item = InventoryItem("phone", 1200.0)
```

```python
def decorator(func):
    def wrapper():
        print("Before calling the function.")
        func()
        print("After calling the function.")
    return wrapper

@decorator # Applying the decorator to a function
def greet():
    print("Hello, World!")
greet()
```

Use Case Example 1: The oven

Simulate and control a real-world heating system using AI (PINN) + classical control (PID) – all inside a Digital Twin.

The idea

- **Problem:** Traditional oven control relies on physical thermocouples and simple control logic. This makes it hard to predict temperature across the entire oven volume or detect future failures.
- **Goal:** Build a **Digital Twin** that can accurately predict the oven's temperature dynamics using a blend of physics and machine learning.
- **Ultimate Value:** The Digital Twin with a PINN incorporated, is fast, accurate, and physically consistent. It can ultimately **replace the slower, legacy physics simulation** in the control loop.

System Architecture

- **Core Principle:** The Twin uses a **Physics-Informed Neural Network (PINN)** to model the system.
  - **Hybrid Approach:** The PINN is trained using two simultaneous inputs:
  - **Data Loss:** Matches the PINN's output to real (or synthetic) temperature sensor data.
  - **Physics Loss:** Forces the PINN's output to obey the 1D Heat Equation.
- **Control Loop Integration:** A **PID Controller** manages the system. In its final version, the PID uses the **PINN's prediction** as its feedback signal, making the PINN the system's "live" state model.

Physics of the system and simplifications

- **Governing Equation: The system's dynamics are governed by the 1D Heat Equation (Diffusion Equation): $\partial T/\partial t=. \alpha \quad (\partial$^2 $T)/(\partial x$^2)**
- **PINN Physics Loss: The training objective includes minimizing the residual of this equation, ensuring the PINN's temperature predictions are physically valid.**
- **Simplified Physics Model (Baseline): The initial training data and the baseline plot use a highly simplified Lumped Capacitance Model.**
- **This model treats the oven as a single volume, focusing on overall energy transfer (heater input vs. heat loss).**
- **This simplification allows for fast data generation and PID tuning but is less accurate than a full PDE solution.**
- **Initial/Boundary Conditions (IC/BC):** The PINN is constrained to match the initial oven temperature (IC) and the desired set point at the boundaries (BC), forcing it toward the stable operating point.

Core Classes and Functionality

- **OvenPINN(nn.Module):**
  - **Function:** The Digital Twin itself. A standard PyTorch feed-forward network.
  - **Input:** Normalized time (t) and position (x).
  - **Output:** Normalized temperature (T).
- **pde_loss(model, ...):**
  - **Function:** Calculates the **Physics Loss** by using PyTorch's torch.autograd.grad to compute the partial derivatives (time and spatial) necessary to check if the Heat Equation is satisfied.
- **PIDController :**
  - **Function:** Classic control algorithm that calculates the required heater power (Qin) based on the error (Set Point−Current Temp).
  - **Role:** Drives the system towards the target temperature.
- **simulate_physics_step(...) :**
  - **Function:** The **Physical Twin Baseline**. Used only to generate the initial, high-quality training data and for non-controlling comparison in the final plot.

Takeaways and summary

- **Hybrid Power:** This project successfully merged data-driven modeling (neural network) with first-principles physics (Heat Equation) to create a robust model.
- **Stability Achieved:** Through careful tuning of PID parameters and **loss weights** (w_data=10.0, w_pde=0.5), the PINN achieved high fidelity, accurately tracking the target behavior without wild oscillations or long-term decay.
- **PINN as the System of Record:** In the final step, the PINN successfully replaced the traditional physics model in the control loop, demonstrating the potential for **faster, more adaptive** control in real-world applications. The PINN is the future of the Digital Twin.

Use Case Example 2: The pendulum

Simulate and control a real-world pendulum system using AI (PINN) + classical control (PID) – all inside a Digital Twin.

Type of pendulum

- **This is a *vertical* (or *upright*) pendulum — specifically, an *inverted pendulum*.**
- The goal is to **balance it upright** — that's the classic "inverted pendulum" problem
- We will use a PID + PINN to *control* it to stay balanced — not fall down.
  That's the *inverted* part: gravity pulls down, but you're trying to keep it *upright*.
- It's *upright*, unstable, and hard to control.
- We will build a **Digital Twin of the pendulum** that uses **AI (PINN)** to simulate physics and **classical control (PID)** to stabilize it — all in code, no CAD, no real hardware needed.
- Physics: $u = ml^2\ddot{\theta} + b\,\dot{\theta} + mglsin\theta$ OR $\ddot{\theta} = \frac{u - b\dot{\theta} - mglsin\theta}{\mathrm{ml}^2}$
- **Parameters:** mass m, length l, viscous friction b, gravity g.
- **Numerical solution:** 4th-order Runge–Kutta (RK4) with step dt. (will explain)

The pendulum Digital Twin

Truth/Physical side (simulated "plant"):

- Nonlinear ODE + **RK4** integrator → generates $\theta(t)$, $\dot{\theta}(t)$.
- **Controller (PID)** computes torque $u(t)$.
- **Actuator limits:** torque saturation; integral anti-windup.
  - Torque Saturation: When your controller asks for $u = 5$ N·m but the motor can only do 2.5 N·m, the command is **clipped** to 2.5).
  - Anti-Windup: If the actuator is **saturated** for a while, the integral keeps accumulating error even though the output can't increase. Don't let the integral grow without bound.

Twin/Virtual side (analytics):

- **Telemetry log:** record (t, θ, u) each step.
- **Visualization:** time series, animation, KPI (settling, overshoot).
- **Artifact store:** saves trained PINN weights + normalization for reuse.

Digital Twin Architecture (PINN + data flow)

- **Inputs available:** time t, control $u(t)$,measured $\theta(t)$ .No $\dot{\theta}$ labels needed.
- **PINN model (state form):** outputs $\theta(t)$and $\omega(t) = \dot{\theta}(t)$.
- **Physics-informed residuals:**

$$r_1 = \dot{\theta} - \omega$$
$$r_2 = ml^2\,\dot{\omega} + b\,\omega + mgl\sin\theta - u$$

- **Loss =** $\| r_1 \|^2 + \| r_2 \|^2$ **+**small data fit $\| \theta - \theta_{meas} \|^2$
  +ICs at $t = 0$(pin $\theta(0)$,gentle $\omega(0)$ **+** (weak prior on b.
- **Training flow:** run plant → log telemetry → train PINN → **save artifact** → reuse for prediction/animation without retraining.

Control (PID)

PID control to upright:

- **Error:** wrapped $e = wrap\left(\theta^{\backslash *} - \theta\right)$ so, the controller moves shortest way to π.
- **Law:** $u = K_p e + K_i \int e \ dt - K_d \ \dot{\theta}$ (derivative on measurement).
- **Practicalities:** torque saturation, **anti-windup** clamp, small **initial kick** to help swing-up.

What the PINN does

- **Digital twin inference :** reconstructs $\theta(t)$)and $\omega(t)$ (from time alone**,** enforcing physics.
- **Parameter ID :** estimates unknowns (e.g.**, damping** b) from routine operation—no special experiments. *(**Damping** b: natural braking that smooths swings; more b= more "built-in" stability but higher effort to move.)*
- **Robustness:** light data term + ICs prevent degenerate fits**;** physics keeps trajectories dynamically consistent.
- **Reuse:** saved **artifact** enables fast predictions & animations in later demos with no retraining.

AI / ML plus Physics and a PID controller is the state of the art

- We are **blending AI + classical control** in one system.
- PINN learns the **physics** → no need to hand-code equations.
- PID gives you **real-world control** → makes it usable.
- This is the **future of Digital Twins — physics-informed AI + control = intelligent simulation**.

Exercise 11

Exercise 11 details

- PINN/Digital Twin assignment without PID control – the **Falling Body with Air Resistance (Terminal Velocity)** model.

- This system models an object (like a skydiver or a dropped sphere) accelerating due to gravity while being opposed by a non-linear drag force.

- **Physics and Governing Equation -** The system's dynamics are governed by Newton's Second Law applied to the vertical axis, where the drag force is proportional to the **square of the velocity** (v2). This square term makes the system non-linear.

- **Governing Equation (First-Order ODE):**

- $m\frac{dv}{dt} = mg\ - \gamma v^2$(Where m is mass, g is gravitational acceleration, and γ is the drag coefficient).

Key Features

- Non-Linearity: The v2 term makes the equation non-linear, which is more challenging than a simple linear drop but easily handled by a PINN.

- Terminal Velocity: The solution naturally approaches a constant terminal velocity ($v_t = \left(\frac{mg}{\gamma}\right)^{0.5}$) when acceleration (dv/dt) becomes zero.

Exercise 11 details

Assignment Focus (DT Goal)

- DT Goal: The PINN acts as the digital twin, predicting the velocity v(t) and position y(t) over time.
- PINN Inputs/Outputs: Input is only time (t); Outputs are Velocity (v) and Position (y).
- Initial Conditions (v(0)=0, y(0)=0) and a few sparse data points near the start.
- The Physics Loss forces the PINN to predict the correct long-term behavior (reaching terminal velocity) even if the training data doesn't fully cover the entire terminal velocity phase.
- Simple State: The system's state is fully defined by velocity (v), and position (y) can be found by integrating velocity (dy/dt=v).
- Mass (m) 80 kg (Represents a typical object/person)
- Gravitational Accel. (g) 9.81 m/s2 (Standard)
- Drag Coefficient (γ) 0.31 kg/m (Yields a terminal velocity of ≈50.3 m/s)

End of Lesson