DIFFERENTIAL DRIVE ROBOT SIMULATION WITH P, PD, AND PID CONTROLLERS

PROJECT DETAILS

Student: IT22127464 — Dhananjaya A.G.D. **Project Title:** Robot Derivation Path Mechanism Review

INTRODUCTION

This simulation implements an interactive differential-drive robot designed to review and demonstrate the “Derivation Path Mechanism” behind robot motion and control. It visualizes how P, PD, and PID controllers affect the robot’s movement toward a click-selected target while tracking performance metrics and enabling analysis. The simulation is implemented with Pygame and NumPy, and includes plotting via Matplotlib. The code emphasizes first-principles derivations for linear and angular motion, connecting those equations to practical controllers and on-screen behavior.

KEY FEATURES

Differential-drive robot model with clear math derivations from kinematics to screen motion.

Switchable controllers: P, PD, PID (with tuned default gains).

Interactive UI (click to set destination, Start/Stop; toggle controller and performance info).

Path visualization and basic performance monitoring (path length, energy, overshoot, settling time, etc.).

On-demand performance analysis with charts (trajectory, error, velocities, energy).

Extensible architecture (state machine, modular controllers, future advanced behaviors).

MATHEMATICAL MODEL

The differential-drive robot is characterized by the velocities of its two wheels, `v_left` and `v_right`. The wheel separation (wheel base) is denoted by `L`.

The robot's overall linear and angular velocities are derived as follows:

Linear Velocity (v): `v = (v_left + v_right) / 2` The average velocity is calculated by taking the mean of the two simulated wheel velocities.

Angular Velocity (ω): `ω = (v_right - v_left) / L` This represents the rate of change of the robot's orientation.

The robot's pose `(x, y, θ)` evolves according to the following kinematic equations:

`dx/dt = v * cos(θ)`

`dy/dt = v * sin(θ)`

`dθ/dt = ω`

Here, `(x, y)` represents the robot's position, and θ is its orientation angle.

CONTROL LAWS

Controllers use the distance error (e) and heading error (e_θ) to produce a control signal u that drives the wheel velocities. The high-level `RobotController` computes a linear command (for v) and an angular command (for `ω`) from these errors, then converts `(v, ω)` into `(v_left, v_right)`.

P Controller: `u = Kp * e`

PD Controller: `u = Kp * e + Kd * d(e)/dt`

PID Controller: `u = Kp * e + Ki * ∫e dt + Kd * d(e)/dt`

Where:

e is the current distance error to the target.

`d(e)/dt` is the rate of change of the distance error.

`∫e dt` is the integral of the distance error over time.

REPOSITORY STRUCTURE

`robot_simulation.py` — Entry point. Initializes Pygame, sets up a finite state machine, and runs the main loop.

`main_state.py` — The interactive state. Handles mouse/keyboard input, renders UI (buttons, labels, path, robot), converts between world and screen coordinates, and drives the robot via `RobotController`.

`differential_drive_robot.py` — The robot model and kinematics. Maintains pose, applies wheel velocity limits/smoothing, and updates state via the derived equations.

`controllers.py` — Modular P, PD, PID controllers and the high-level `RobotController` that blends linear and angular control, applies distance-based scaling and saturation, and outputs wheel velocities.

`performance_monitor.py` — Tracks and computes performance metrics (arrival time, path length, energy, overshoot, settling time, steady-state error, control effort) and plots them.

`advanced_behaviors.py` — Future extensions: smooth paths (Bezier), spiral approaches, adaptive gains, and basic formation placeholders. Not wired into the main loop by default.

`simulation_state.py` — Simple abstract base for state machine states.

`config.py` — Central configuration (window, colors, physical limits, tuned gains, labels, version).

`Doucumentation/` — Assignment documents (`DELIVERABLES.md`, `README.md`, `REPORT.md`).

INSTALLATION

Requirements:

Python 3.10+ (tested with 3.12)

Packages: pygame, numpy, `matplotlib`

Install packages (Windows PowerShell):

```
pip install pygame numpy matplotlib
```

If you use a specific Python installation (e.g., `C:\Python\Python312\python.exe`):

```
C:\Python\Python312\python.exe -m pip install pygame numpy matplotlib
```

RUNNING THE SIMULATION

From the project root:

```
python robot_simulation.py (using python syntax)
```

```
or
Start_Simulation.bat
```

```
or
Start_Simulation.ps1
```

A Pygame window opens with the robot at the world origin (mapped to the screen center). Click anywhere (above the bottom command bar) to set a new destination.

UI AND CONTROLS

Click within the main canvas to set the destination. The robot immediately starts navigating.

Buttons (bottom bar):

Start — Reset robot to origin and reset controllers, clear path.

Stop — Halt robot and reset controller internal states.

P / PD / PID — Switch control mode.

Monitor — Toggle performance metrics display in the top-left.

Analyze — Print a performance report to the console and open analysis plots.

Path — Toggle path drawing.

Clear — Clear the accumulated path.

Keyboard:

T — Run an internal movement test sequence (prints to console).

Notes:

World-to-screen mapping uses 100 px/m with the screen center as (0,0) in world coordinates. Y is flipped for screen rendering.

Path points are throttled to avoid clutter and capped for performance.

CONFIGURATION HIGHLIGHTS (CONFIG.PY)

Timing and Window:

`SIMULATED_SECOND = 1000`, `FPS = 60`

`SCREEN_WIDTH = 800`, `SCREEN_HEIGHT = 600`, `TITLE = "RIS Assignment -- Differential Drive Robot Simulation 1.0.0"`

Colors: predefined RGB tuples for UI and drawing.

Robot and Control:

`WHEEL_BASE = 0.3` m, `MAX_LINEAR_VELOCITY = 3.0` m/s, `MAX_ANGULAR_VELOCITY = 5.0` rad/s

`MAX_ACCELERATION = 5.0` m/s^2 (used for velocity smoothing).

Tolerances: `POSITION_TOLERANCE = 0.03` m, `ANGLE_TOLERANCE = 0.1` rad.

Tuned Gains (defaults used by RobotController):

P: linear Kp=1.5, angular Kp=4.0

PD: linear Kp=2.5, Kd=0.8; angular Kp=4.5, Kd=1.2

PID: linear Kp=3.0, Ki=0.05, Kd=0.7; angular Kp=5.0, Ki=0.02, Kd=1.0

You can tweak these values to study the effect on motion, overshoot, and settling.

PERFORMANCE MONITORING AND ANALYSIS

While running, toggle “Monitor” to see basic metrics inline. Press “Analyze” to:

Print a summary (arrival time, path length, energy, overshoot, settling, steady-state error, control effort).

Show Matplotlib plots for trajectory, error, velocities, control signals, energy, and a metrics panel.

Receive optimization suggestions (e.g., adjust Kp/Kd/Ki) based on observed behavior.

Tip: Switch between P/PD/PID and compare plots to understand how derivative and integral actions shape the response.

ADVANCED BEHAVIORS (PREVIEW)

The `advanced_behaviors.py` module includes:

Curved path generation via cubic Bezier interpolation.

Spiral approach waypoints for precise docking.

An adaptive controller scaffold that adjusts gains based on observed performance.

Simple formation logic (leader-follower offsets).

These are not integrated into `main_state.py` by default but are suitable for experiments and future extensions. A typical integration pattern is to produce waypoints with `RobotBehavior` and command the robot to track them sequentially using the existing `RobotController`.

TUNING GUIDANCE

Start with P control and increase Kp until response is fast but not unstable.

Add Kd (PD) to reduce overshoot and oscillations.

Add Ki (PID) to remove steady-state error; use integral limits to prevent windup.

Use “Analyze” to corroborate tuning with metrics and plots.

KNOWN LIMITATIONS

No obstacle modeling; path planning is straight-line to target (advanced path generators are provided but not wired in).

Single robot in the main application (formation code is a placeholder for future work).

Physics is purely kinematic; no slippage or dynamics.

CREDITS

Student: IT22127464 — Dhananjaya A.G.D. **Course context:** RIS Assignment — Differential Drive Robot Simulation **Docs:** see `Doucumentation/` for assignment deliverables and report.

TROUBLESHOOTING

If the window does not open or crashes, ensure Pygame is installed and that your GPU/driver supports basic 2D rendering.

If plots do not show, verify Matplotlib is installed and your Python environment is the one used to run the app.

On high-DPI displays, adjust `meters_to_pixels` in `main_state.py` for preferred scaling.