ICP-HSS-Integrated Coconut Processing and Harvest Simulation System

Allen Thomas Alex
Abhirami M
Abhinav Varghese
Albin Chacko
Department of Robotics & Automation
Saintgits College of Engineering (Autonomous), Kottayam.

Abstract

Meet the Coconut Harvester Dashboard—a playful blend of math, code, and orchard hijinks. Picture a six-wheeled rover rolling through a procedurally generated grove, sizing up coconuts, firing up a power cutter, and cooling down before the sap gets sticky. Behind the scenes, partial differential equations keep the blade temperate, Laplace transforms wrangle the arm's "don't overshoot" behavior, and optimization decides which tree deserves first crack. All of this feeds a developer-styled Matplotlib control room packed with animated panels, neon risk gauges, and an AI "Mission Brain" that narrates what the rover is thinking. The experience is intentionally approachable: students can trace a line from textbook PDE stencils to the glowing telemetry readouts, or from Laplace-domain algebra to the arm's graceful squat. A few pytest suites stand guard, guaranteeing that the simulation behaves when you run it headlessly at 3AM before lab. Whether you're an engineer in training, a professor hunting for a lab demo, or simply a robotics geek who likes coconuts, the dashboard delivers a cheerful tour of data flow, numerical modeling, and planning under constraints. It's part teaching tool, part interactive toy, and entirely designed to make computational engineering feel like a tropical field trip.

1 Introduction

What happens when you give a robotics lab an orchard, a stack of math textbooks, and a sense of mischief? You get the Coconut Harvester Dashboard: a simulation that proves serious engineering concepts can wear a fun shirt. The mission is simple—get coconuts from trees without burning the cutter, draining the battery, or bashing into obstacles—but every step is powered by concepts students learn in class. The result is a single playground where theoretical ideas turn into animated panels and satisfying "mission complete" readouts.

The dashboard kicks off by building a stylized grove: irrigated furrows, leafy canopy discs, sprinkle-dot coconuts, and a set of circular "do not bump" obstacles. Drop in the six-wheeled rover—polygon chassis, chunky tires, articulated arm—and fire up the mission planner. That planner is a greedy strategist: it calculates how much time and energy every "drive, scan, deploy, cut, cool" sequence would cost, checks for obstacle collisions, and drafts a queue that stays within mission limits. Trees that can't be reached get marked "blocked," while the lucky targets are lined up for action.

The software is composed of several interdependent modules:

- **dynamics.py** – models arm movement dynamics.
- **thermal.py** – simulates the thermal behavior of the harvesting tool.
- **kinematics.py** – calculates arm positioning and motion.

- **optimizer.py** – plans an optimal mission path.
- **main.py** – coordinates subsystems and renders the visualization dashboard.

Testing modules under the `tests/` directory ensure the safety and correctness of all critical behaviors.

Methodology (Module Mapping)

1. Mission Planning (Module 05 – Optimization)

Code: `optimizer.py`

Module Link: `05-optimization.qmd` introduces linear programming concepts and resource allocation. The planner applies these concepts by estimating time and energy costs for operations such as *drive, scan, deploy, cut, cool*, checking obstacle feasibility (segment-circle test), and constructing a greedy schedule under mission constraints. Trees blocked by obstacles are tagged, and feasible ones are queued, mirroring the optimization mindset presented in Module 05.

2. Dynamics & Kinematics (Modules 01 and 03)

Code: `dynamics.py`, `kinematics.py`

Module Link: `01-intro-python.qmd` includes forward-kinematics exercises; the dashboard utilizes those ideas in `kinematics.py` for the arm's 2-link forward kinematics and world-frame overlay. `03-laplace-basics.qmd` covers Laplace transforms and step responses; `dynamics.py` solves:

$$I\theta'' + c\theta' + k\theta = ku(t)$$

symbolically, lambdifies it, and extracts the 2% settling time. That settling time tunes the arm-deployment delay during the *DEPLOYING_ARM* phase.

3. Thermal Modeling (Module 02 – PDE Numerical)

Code: `thermal.py`

Module Link: `02-pde-numerical.qmd` explains FTCS and stability criteria. The tool's temperature is governed by the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + Q$$

`thermal.py` implements FTCS with stability condition:

$$r = \frac{\alpha \Delta t}{\Delta x^2} \leq 0.5$$

It heats during *CUT_TREE* ($Q > 0$) and cools during *COOL_TOOL* ($Q = 0$), computing a duty cycle that adheres to the PDE stability lessons from Module 02.

4. Visualization & Telemetry (Module 01 + Capstone Context)

Code: `main.py`

Module Link: `01-intro-python.qmd` introduces scientific plotting. `main.py` orchestrates the mission, updates system state each frame, and renders the developer-styled Matplotlib dashboard—world view,

tree tracker, mission utilization, arm response, cutter temperature, systems telemetry, obstacle clearance, AI reasoning log, and mission timeline—demonstrating how Module 01's plotting tools enable a comprehensive monitoring console.

5. AI Commentary and Planner Console (Module 04 – Laplace Applications + Narrative Layer)

Code: `main.py` (AI panel, planner log)

Module Link: `04-laplace-apps.qmd` explores system responses to switching events. The AI "Mission Brain" and console mirror these state changes as human-friendly, timestamped updates—tying the module's piecewise and impulse-response intuition to the narrative telemetry layer.

6. Testing & Continuous Learning (Across Modules)

Code: `tests/` (thermal, optimizer, dashboard)

Module Link: Each module's concepts are reinforced through corresponding test files:

- **Module 02:** `test_thermal.py` checks cooling monotonicity and heating energy increase.
- **Module 05:** `test_optimizer.py` validates time/energy ceilings and obstacle blocking.
- **Module 01 + Dashboard Integration:** `test_dashboard.py` ensures headless initialization and telemetry logging.

2 Illustrations and Graphs

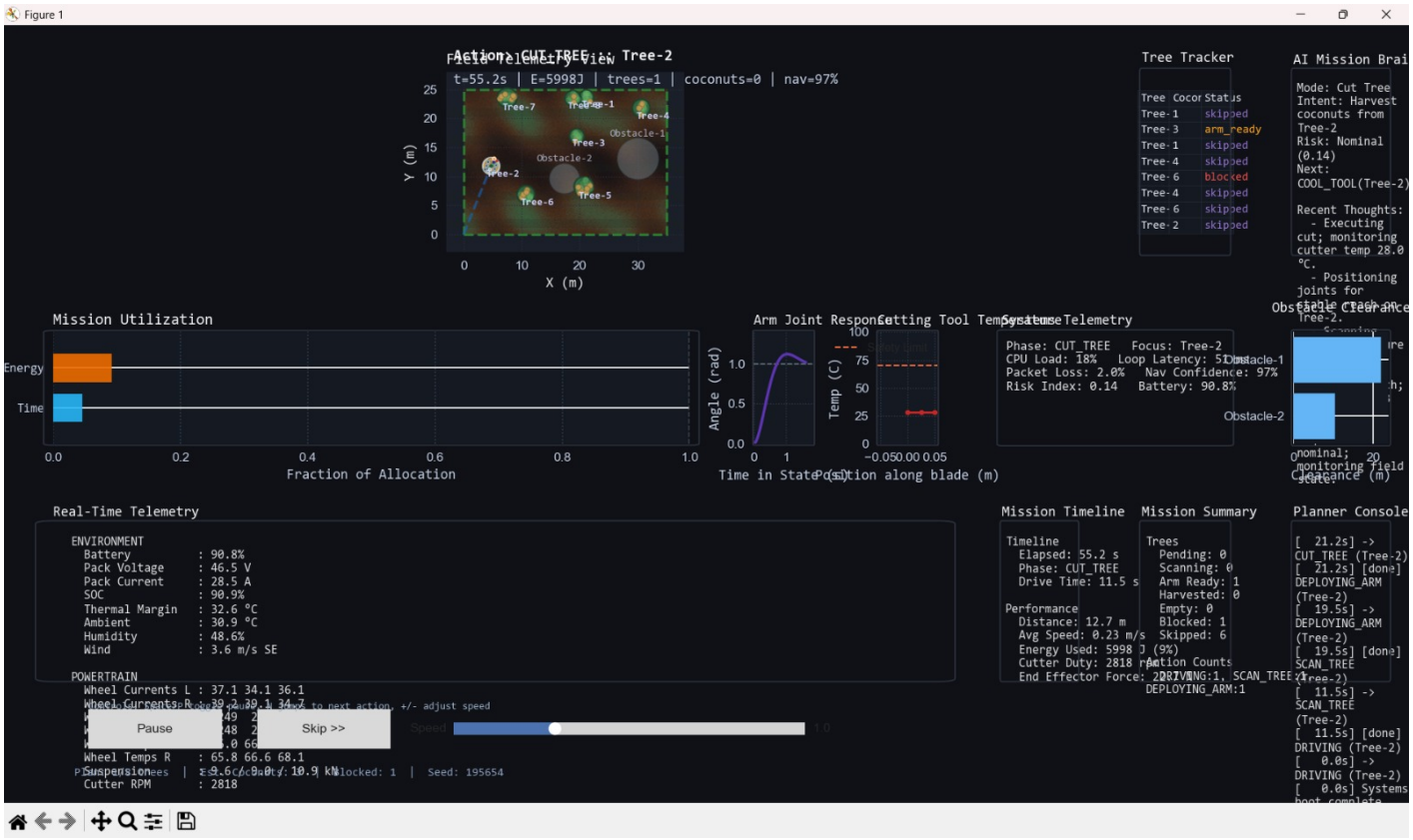

Figure 1: Mission control telemetry interface showing live system data and performance indicators.

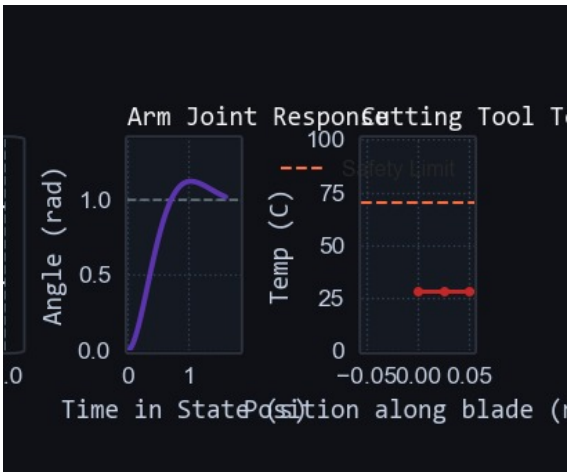

Figure 2: Arm Joint Response and Tool Temperature visualization.

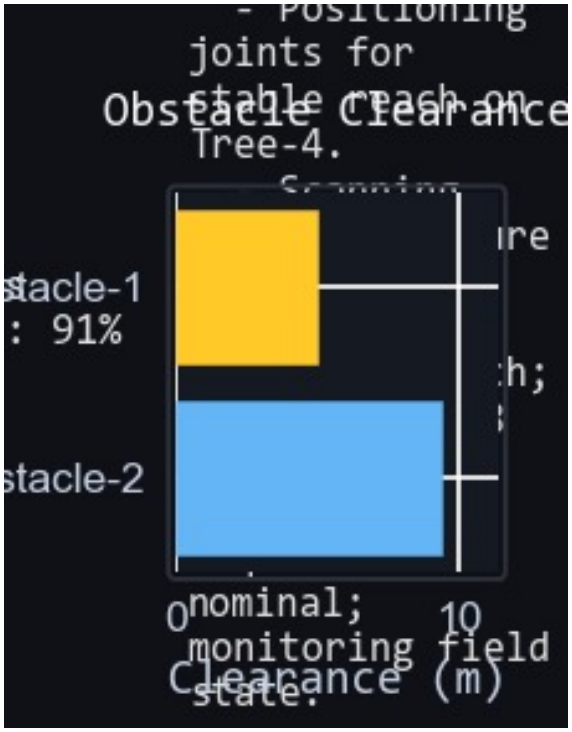

Figure 3: Obstacle clearance tracking visualization.

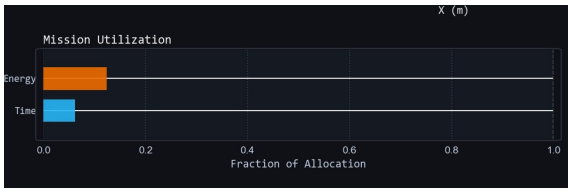

Figure 4: Obstacle clearance tracking visualization.

3 Results

Functional Visualization: The dashboard renders a procedurally textured orchard, rover path, articulated arm overlay, and status-colored trees. AI reasoning and planner logs narrate mission progress.
Telemetry Coverage: Real-time panels expose environment conditions, drivetrain currents/torques, joint angles/temperatures, communication metrics, and task objectives. CSV exports mirror these fields for offline studies.
Planner Behavior: Missions execute under defined time/energy ceilings. Obstacles flagged by segment-circle intersections are excluded, with affected trees labeled "blocked."
Thermal Safety: The FTCS solver produces heating/cooling profiles that cap maximum tool temperature; telemetry records thermal margins and end-effector forces.
Test Outcomes: Automated tests (6 total) pass, confirming stable thermal behavior, planner compliance, and dashboard headless readiness, with only benign Matplotlib animation warnings in non-render mode.

4 Conclusion

The Autonomous Coconut Harvester Dashboard demonstrates how PDE solvers, Laplace-domain analyses, and optimization heuristics can live side by side in a robotics workflow. By grounding theoretical modules in a richly instrumented rover mission, the project offers students a concrete bridge between

classroom mathematics and system-level engineering. Future improvements could replace the greedy planner with MILP/TSP formulations, enrich sensing and terrain models, integrate closed-loop control for the arm, and add replay tooling for mission telemetry. Nevertheless, the current codebase—complete with documentation and tests—already serves as a robust educational platform for computational methods in electronics and robotics.

References

[1] Pytest Framework, last accessed on 15 October, 2025,`https://docs.pytest.org`

[2] SymPy Documentation, last accessed on 15th October,2025,`https://docs.sympy.org`

[3] NymPy Documentation, last accessed on 15th October, 2025, `https://numpy.org/doc`

[4] Matplotlib Documentation, last accessed on 17th October,2025,`https://matplotlib.org/stable/index.html`

[5] Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G. (2010). Robotics: Modelling, Planning and Control. Springer.

[6] Chapra, S. C., Canale, R. P. (2015). Numerical Methods for Engineers (7th ed.). McGraw-Hill Education.