USER MANUAL

GISELLE

Pedestrian-Aware Emergency Braking for Low-Friction Settlement Roads

Team GISELLE · CPWC2026-0040

Index

Table of Contents · GISELLE User Manual

1. Introduction 3

Overview · Objectives of the Project

2. System Requirements 3

Hardware · Software

3. Installation and Execution Guide 4

4. System Architecture 5

Module Roles · Key Variables List

5. Code Details 7

Inputs from Simulator · Module-by-module

6. Validation Results 9

1. Introduction

Overview

This document is the user manual for "GISELLE — Pedestrian-Aware Emergency Braking for Low-Friction Settlement Roads." The system assumes a near-future scenario in which lunar or Martian outposts have grown into permanent settlements with roads, surface traffic, and pedestrians. On regolith-covered roads the effective tire-road friction coefficient is far below Earth asphalt ($\mu \approx 0.17$ in our test environment), so the braking distance of the same vehicle at the same speed grows by more than five times (measured: 8.2 m on Earth vs 42.7 m on the lunar surface at ~44 km/h). An ADAS stack tuned for Earth quietly assumes Earth friction; on settlement roads that assumption produces collisions.

GISELLE is a simulator-based Advanced Driver Assistance System that (1) automatically detects pedestrians with a transfer-trained YOLO11 network, (2) measures their distance and speed with a stereo-vision algorithm using two virtual cameras, (3) cross-verifies every detection against simulator telemetry so that false positives can never trigger the brake, and (4) executes a friction-aware emergency braking plan through PWM actuation of the parking brake, stopping the vehicle a class-dependent safety margin in front of the pedestrian (8 m for an adult, 10 m for a child).

Objectives of the Project

Conventional AEB systems are calibrated for Earth road surfaces, and their braking-onset distances are physically insufficient when friction drops by a factor of five. Testing a pedestrian AEB for low-gravity, low-friction environments is impossible on Earth — and must never be attempted with real pedestrians. We overcome both problems through simulation. UC-win/Road lets us set the surface friction per road, place child and adult pedestrians crossing at controlled speeds (8 km/h and 12 km/h), replay the identical emergency scenario repeatedly, and — critically — compare every perception output against simulator ground truth. Our objectives are: reliable pedestrian detection and classification (adult vs child); real-time distance and speed measurement without HD maps or beacons; braking control that stops within a repeatable margin on $\mu = 0.17$; and passenger comfort verified against NASA-STD human acceleration and jerk limits.

2. System Requirements

Hardware Requirements

(The following specifications are the official recommendations for the smooth operation of the UC-win/Road simulator.)

- CPU: Intel Core i7 / AMD Ryzen 7 (10th Gen or newer)
- RAM: 32 GB
- GPU: NVIDIA GeForce RTX 3060 or higher (VRAM 8 GB+)
- Storage: 50 GB+ free space (SSD recommended)

Software Requirements

- Operating System: Windows 10 / 11 (64-bit)
- Simulator: UC-win/Road Ver.19.0.1 Ultimate (64-bit)
- Language: Python 3.11 (64-bit, matching the simulator's bit version)

External Python libraries (installed via pip):

- `ultralytics` — YOLO11 inference (pedestrian detection)
- `opencv-python` — image processing, template matching, HUD rendering
- `numpy` — numerical computation (speed regression, disparity refinement)
- `pywin32` — COM automation and window capture on Windows

3. Installation and Execution Guide

3.1 Prerequisites

Install UC-win/Road & Python: install UC-win/Road Ver.19.0.1 and Python 3.11 (64-bit) and enable the Python API plugin (refer to the official UC-win/Road documentation).

Register the COM server (one time): in the [Python API] ribbon, click the cube icon labeled "Not registered" so that it changes to "Registered". Without this step external Python cannot attach to the simulator.

Install the required libraries:

```
pip install ultralytics opencv-python numpy pywin32
```

3.2 Simulator Setup

a. Log output (telemetry). Open the Log Export settings and enable: items "Driver car" and "Surrounding moving objects" (radius 500 m); host rows 127.0.0.1 : 4000 and 127.0.0.1 : 4001, each with its enable checkbox ticked (brake.py listens on 4000, detect.py on 4001). Start logging together with the scenario (a Start Logs scenario event is recommended).

b. Stereo view windows. Create two views named cam_L and cam_R, coordinate system "relative to the driver car": cam_L = (lateral −0.25 m, height +1.6 m, forward +0.5 m), cam_R = (lateral +0.25 m, height +1.6 m, forward +0.5 m) — baseline B = 0.50 m. Both views: identical direction (yaw 0, pitch 0, roll 0), vertical FOV 45°, mirror alternation off, mouse control off.

c. Pedestrian IDs. Pedestrian object IDs appear in the telemetry stream and are stable per project (in our project: child = 5764 at 8 km/h, adult = 5763 at 12 km/h). If your project differs, pass `--child <ID> --adult <ID>` to both programs. Candidate IDs are easy to identify from the stream: pedestrians hold a constant walking-speed signature (2.2 / 3.3 m/s).

3.3 Run the System

Open the project in UC-win/Road, start driving (driver's-seat view with the dashboard visible), start the scenario and logging, then run the two programs in separate terminals:

```
# Terminal A — perception (YOLO + stereo + verification + HUD)
python detect.py --model best.pt

# Terminal B — braking controller
python brake.py
```

For a decision-log-only rehearsal without actuation, run `python brake.py --actuator dry`. To run perception without the camera windows (telemetry-only fallback): `python detect.py --no-stereo`. Stop either program with Ctrl+C; brake.py always releases the parking brake on exit.

3.4 Verifying Operation

- detect.py console prints one line per pedestrian: distance, speed, forward/passed, and VERIFIED status; the HUD window draws adult boxes in green, child in orange, unverified in yellow.
- brake.py console prints the state machine (CRUISE → BRAKE), the live trigger distance, and STOPPED with the final margin.
- The perception output contract file `stereo_person.json` is overwritten atomically several times per second and can be inspected at any time.

4. System Architecture

Module Roles and Key Functions

The system is organized as two independent processes over a shared access layer. Perception (detect.py) and control (brake.py) never block each other: perception can crash, restart, or lag without taking the brake loop down.

common.py (Shared Access Layer): everything that touches UC-win/Road. EgoLink attaches to the running simulator via COM (attach-only with a spawn guard — it can never launch a second simulator instance) and reads the driver car's position, heading and speed. Telemetry receives the Log Export UDP stream, tracks each pedestrian ID, and computes speeds from the simulation timestamps inside the packets (wall-clock arrival times are distorted by packet bursts). It also owns the north-south coordinate mirror conversion ($z' = C - z$) between the simulator's internal frame and the log frame.

stereo.py (Stereo Vision Algorithm): captures the cam_L / cam_R view windows (PrintWindow with a screen-region fallback — the 3D view is OpenGL-rendered), computes the disparity of a person bounding box by template-matching the torso band with sub-pixel parabola refinement, converts it to depth by $Z = f \cdot B$ / disparity and lateral position $X = (u - cx) \cdot Z / f$, and estimates crossing / approach speed by linear regression of (X, Z) over a sliding time window.

detect.py (Perception Main): runs YOLO11 (transfer-trained: classes adult / child) on the left camera image, obtains stereo range and speed for each box, and cross-verifies every detection against telemetry by azimuth (±12°) and range (±25 %). Verified detections inherit the authoritative class label; unverified boxes (tires, poles) are displayed but excluded from decisions. The nearest verified forward pedestrian is published to stereo_person.json (atomic write; 999999 = no target).

brake.py (Braking Controller): consumes pedestrian geometry, computes a friction-aware trigger distance every cycle, and executes braking by PWM actuation of the parking brake with PI duty correction. It stops the vehicle a class-dependent margin before the pedestrian, detects when the target has been passed, and releases automatically. A dry-run mode logs decisions without actuating.

Key Variables List

Category	Description	Meaning / Update rule	Variable (in code)
Perception	Distance to target	Distance to the nearest verified forward pedestrian; 999999 = no target (controller keeps cruising)	`distance_m`
Perception	Ego speed feed	Ego speed in m/s supplied to the controller with each perception frame	`closing_speed_mps`
Perception	Pedestrian class	adult / child — selects the stop margin (8 m / 10 m)	`class`

Category	Description	Meaning / Update rule	Variable (in code)
Perception	Verification flag	True when YOLO box and telemetry agree in azimuth and range	`verified`
Perception	Azimuth tolerance	Max angular disagreement for verification (12°)	`AZ_TOL_DEG`
Stereo	Focal length	f = (H/2) / tan(vFOV/2), from the view's vertical FOV	`f_px`
Stereo	Baseline	cam_L – cam_R spacing (0.50 m)	`baseline`
Stereo	Disparity	Left-right pixel shift of the target patch (sub-pixel)	`disparity`
Control	Trigger distance	Braking-onset distance recomputed every cycle from speed, latency and friction-limited deceleration	`trigger`
Control	Target deceleration	Deceleration required to stop at the margin, clamped to [0.4, 5.0] m/s^2	`a_des`
Control	Brake duty	PWM parking-brake duty, feed-forward + PI correction	`duty`
Control	Speed plan	Desired speed profile integrated from a_des	`v_des`
Control	Stop margins	Class-dependent stop distance: adult 8 m, child 10 m	`D_STOP_BY_CLASS`
Control	Pass detection	Minimum-distance memory; releases the brake once the target recedes	`min_d / passed`
Telemetry	Mirror constant	North-south frame conversion z′ = C − z (measured per map)	`MIRROR_C`
Telemetry	Teleport filter	Speed samples above 30 m/s are route-loop artifacts and are discarded	`TELEPORT`
Telemetry	Closing rate	EMA-filtered closing speed per target	`close_rate`

5. Code Details

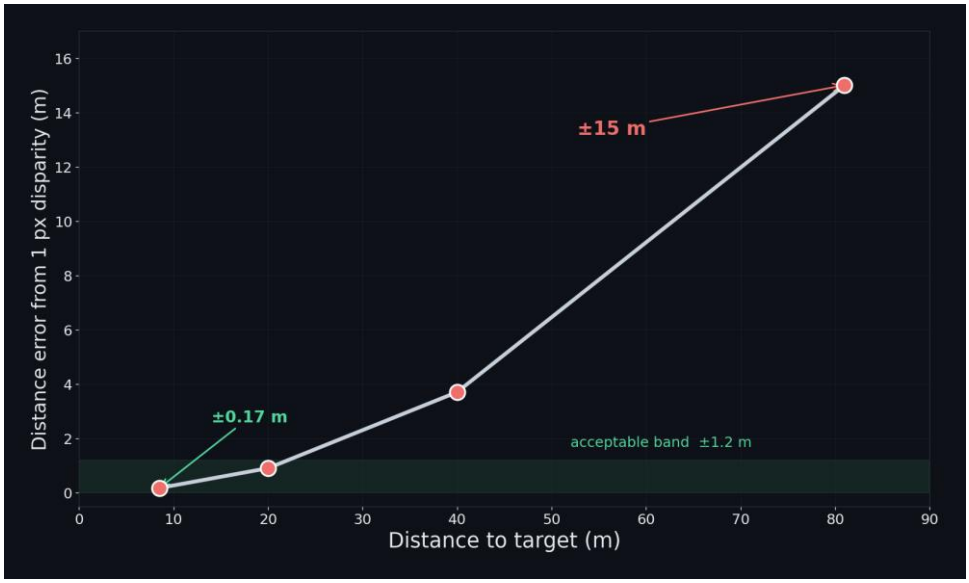
Inputs from Simulator

- **Stereo camera views:** two RGB image streams (cam_L / cam_R) captured from view windows fixed to the vehicle body — the virtual equivalent of a front stereo camera rig.
- **Vehicle information (COM):** driver car position, heading vector and speed, read through the UC-win/Road COM API `(SimulationCore.TrafficSimulation.Driver.CurrentCar)`.
- **Telemetry stream (UDP):** positions of surrounding moving objects from the Log Export plugin, used for pedestrian ground truth, speed signatures, and detection verification.

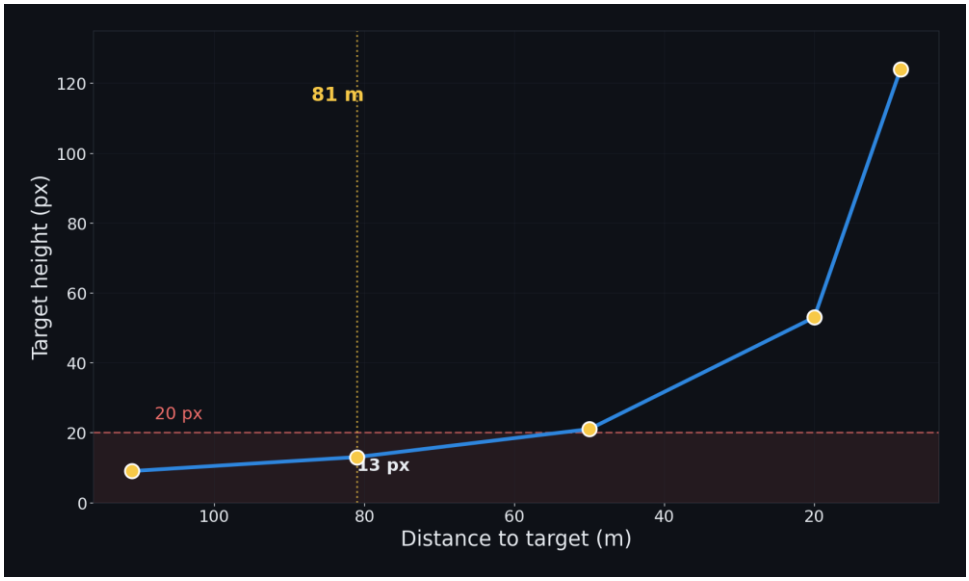
5.1 Stereo distance & speed (stereo.py)

The torso band of each detected person (15–60 % of box height — legs move too much) is template-matched along the same rows of the right image; the correlation peak is refined to sub-pixel precision by parabola interpolation:

```
Z = f_px * baseline / disparity          # depth
X = (u - cx) * Z / f_px                  # lateral position (right +)
dist = sqrt(X² + Z²)
vx =  dX/dt   (crossing speed)   vz = -dZ/dt   (approach speed)
```



Stereo ranging error caused by 1 px of disparity: ±0.17 m at 9 m, ±15 m at 81 m. Stereo precision belongs to the near field.



A pedestrian at 81 m is 13 px tall — below the 20 px detection floor. This limit motivates the near/far sensor role split.

These two measurements define GISELLE's sensor architecture: **stereo vision owns the near field; telemetry cross-verification owns the far field.** Detections are matched to telemetry pedestrians by azimuth and range; only verified targets are eligible to trigger braking.

5.2 Verification (detect.py)

```
az_stereo = atan2(X, Z)                          # camera frame
az_tel    = atan2(rx*dz - rz*dx, rx*dx+rz*dz) # heading frame
verified  = |az_stereo - az_tel| <= 12°  and  |d_stereo - d_tel| <= 0.25·d_tel + 3
m
```

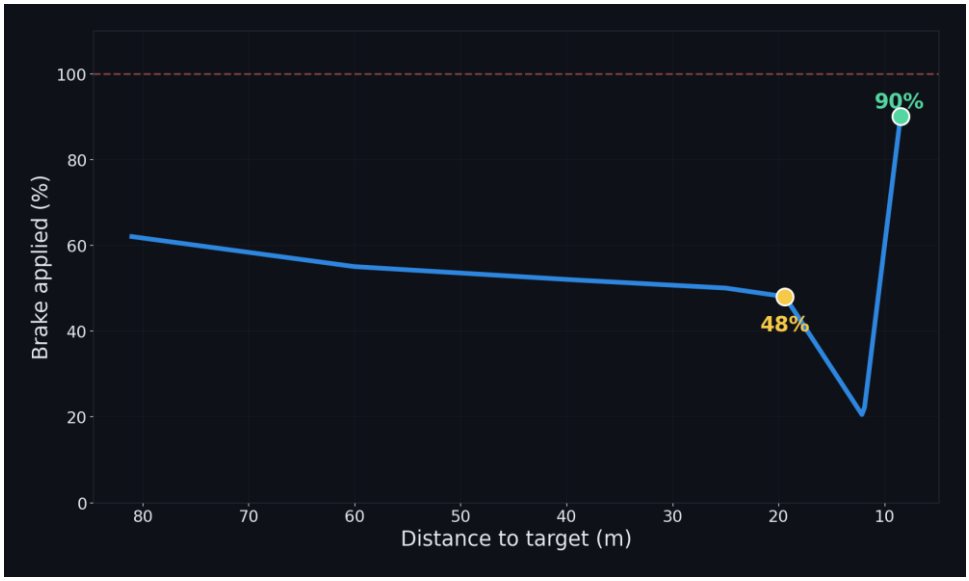
A verified detection inherits the telemetry class label (adult / child), which makes the on-screen classification robust even when the detector confuses visually similar classes at long range.

5.3 Braking control (brake.py)

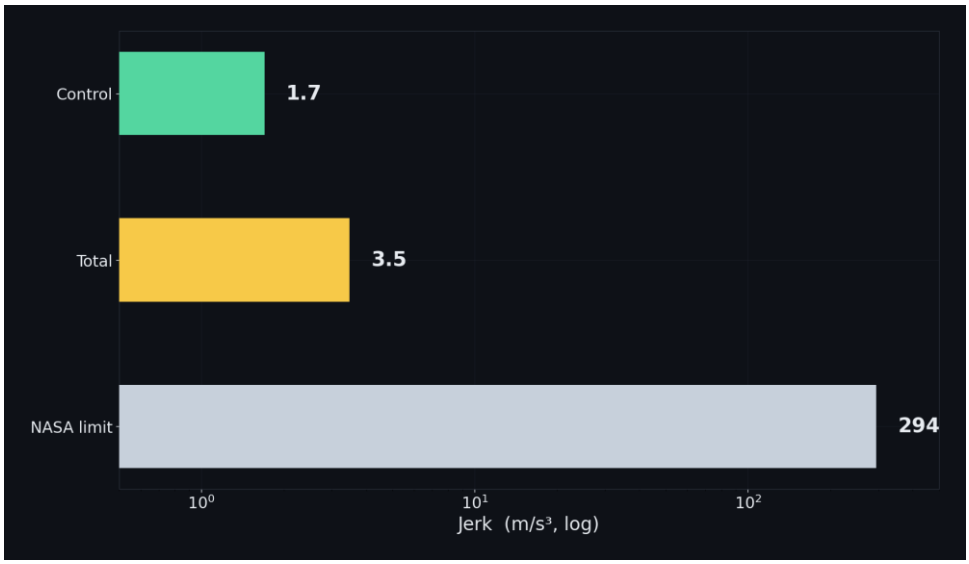
The trigger distance is recomputed every cycle so that braking begins exactly when the remaining distance equals what physics requires on the current surface:

```
trigger = margin + v·t_latency + safety · ( v²/2 + v_ped·v ) / a_target
a_des   = ( v²/2 + v_ped·v ) / (d_remaining - margin)     # clamped 0.4..5.0
duty    = (a_des + a_push)/(a_pb + a_push) + Kp·err + Ki·∫err
```

Actuation uses the parking brake as a PWM actuator (period 30 ms): full engagement decelerates at a measured 6.15 m/s², and the duty cycle blends it against the scenario's push force. The PI term corrects the speed-plan error, which is what makes the stop distance repeatable across entry speeds.



Distance-scheduled braking: ~48 % of brake authority holds the deceleration plan; a 90 % finisher plants the stop.



Jerk decomposition: the control loop contributes 1.7 m/s³ (total 3.5) against the NASA-STD limit of 294 m/s³.

5.4 Shared access layer (common.py)

Attach-only COM. The layer first tries GetActiveObject; where the simulator does not expose a running-object entry it falls back to Dispatch, guarded by a window-existence check and a before/after process comparison that kills any accidentally spawned instance. External tooling therefore can never multiply simulator processes.

Sim-timestamp speeds. All object speeds are computed from the simulation timestamps inside the log packets, not from packet arrival times — inference workloads cause packet bursts that would otherwise collapse measured speeds.

Coordinate mirror. The simulator's internal frame and the log frame are mirrored along the north-south axis; the layer converts with z′ = C − z using a per-map measured constant.

6. Validation Results

- **Stop repeatability:** 11 emergency runs at 20 / 35 / 50 km/h on μ = 0.17 — every stop within ±1.2 m of the 8 m stop line; overall campaign 18 versions, 60+ runs, 0 collisions.
- **Passenger safety margins:** peak deceleration 1.99 m/s² (10.1 % of the NASA-STD limit) and control-loop jerk 1.7 m/s³, total 3.5 m/s³ (1.2 % of the 294 m/s³ limit) — an emergency stop that passengers experience as a controlled glide.
- **Post-stop creep elimination:** low-speed vehicle dynamics ignore brake input while the traffic path-follower keeps advancing the vehicle (root cause isolated by eliminating brake force, slope, drivetrain torque and control override). A position-anchor hold reduced post-stop drift from 6.92 m to 0.02 m across eight controller versions.
- **Perception verification:** azimuth + range cross-check against telemetry rejects YOLO false positives (tires, poles) before they reach the controller; verified classification selects the child's larger 10 m margin automatically.