Mapping & Navigation

Course: Mobile Robotics
Mapping & Navigation Authors:

- 1. אלן חבש
- 2. לין אלואוי
- 3. אחמד סוב לבן

0. Submitted Files Overview

0.1 System Structure and Code Separation

The system developed in **Mapping & Navigation** is composed of **three independent software modules**, each corresponding to a distinct operational phase of the robot.

Each module runs in a **different execution context** (robot-side vs. PC-side) and has clearly defined inputs, outputs, and responsibilities. This structure closely follows standard practice in real robotic systems, where sensing, mapping, and navigation are decoupled.

0.2 Code Modules Overview

0.2.1 Mapping Code (NXC)

The mapping program runs entirely on the LEGO NXT robot and is responsible for **data acquisition** during a full clockwise traversal of the arena perimeter.

Main responsibilities:

- Wall-following using an ultrasonic sensor with PID control
- Corner handling and collision recovery

- Odometry computation during motion
- Continuous logging of time-synchronized pose and sensor data

Engineering emphasis:
Robot speed is intentionally limited to prioritize **measurement stability**, **repeatability**, and **data quality**, since the offline mapping stage can only reconstruct geometry from the recorded dataset.

Key output:

- `walldata*.csv` — time-synchronized pose and ultrasonic measurements (plus event/mode tags)
-

0.2.2 Navigation Code (NXC)

The navigation program is a separate robot-side executable, designed exclusively for mission execution **after** mapping is complete and the arena center has been computed.

Main responsibilities:

- Localization using wall-following and landmark detection (double black line)
- Odometry reset and coordinate frame definition
- Point-to-point navigation to the arena center
- User interaction via sound sensor (clap detection)
- Final alignment and precise stopping

Unlike the mapping program, navigation does **not** perform sensor logging. This reduces computational load and eliminates unnecessary file I/O, improving real-time responsiveness and reliability during the timed, graded mission.

0.2.3 Offline Processing Code (PC-Side, Python)

The offline processing code runs on a PC and performs deterministic geometric reconstruction of the arena from the mapping dataset.

Main responsibilities:

- Loading and preprocessing robot log files
- Loop-closure correction for odometry drift
- Converting ultrasonic measurements into a wall point cloud
- Wall segmentation and straight-line fitting
- Convex polygon construction
- Arena center computation
- Quantitative error and consistency evaluation across runs

This module produces the geometric information required for navigation while remaining independent of real-time constraints.

Key outputs:

- Arena polygon (plots + CSV)
- Arena center coordinates
- Error and consistency summary files
-

0.3 Engineering Rationale for Modular Design

Combining mapping, navigation, and offline processing into a single program would significantly increase system complexity and reduce reliability. By separating concerns:

- Each module can be optimized for its specific task
- Errors can be isolated and debugged independently
- Navigation remains deterministic and time-predictable

- Offline map generation becomes repeatable and verifiable
- Changes to one phase do not destabilize the others

This modular design mirrors real robotic pipelines, where sensing, estimation, and decision-making are deliberately decoupled.

0.4 Submitted Files Summary (with Versions)

0.4.1 Robot-Side Code Files (NXC)

File Name (NXC)	Version	Purpose	Runs On
`mapping_v1_basic.nxc`	v1.0	Wall following only (no logging)	NXT
`mapping_v2_logging.nxc`	v2.0	Basic logging (encoders + US)	NXT
`mapping_v3_final.nxc`	v3.0	Final mapping code (PID + odometry + mode tags)	NXT
`navigation_v1_basic.nxc`	v1.0	Initial navigation logic	NXT
`navigation_v2_final.nxc`	v2.0	Final navigation mission code	NXT

Note: Earlier versions are included to document the development process and engineering evolution.

0.4.2 PC-Side Processing Files (Python)

File Name (Python)	Versio n	Purpose
`arena_mapping.py`	v1.0	Map reconstruction pipeline + RANSAC
`arena_mapping_final.py`	v2.0	Deterministic DBSCAN + TLS pipeline
`mapping_error_eval.py`	v1.0	Quantitative error and consistency analysis

0.4.3 Data Files (Generated by Robot)

File Name	Description
`walldata.csv`	Mapping run #1
`walldata1.csv`	Mapping run #2
`walldata2.csv`	Mapping run #3
`walldata3.csv`	Mapping run #4

0.4.4 Output and Evaluation Files

File Name	Description
`arena_map.png`	Final reconstructed arena plot
`arena_center.txt`	Arena center coordinates
`mapping_error_summary.csv`	Per-run wall fitting errors
`mapping_consistency.csv`	Inter-run consistency statistics

0.5 Video Documentation (Recorded / Planned)

Video File Name	Content
`mapping_final.mp4`	Full perimeter mapping run
`navigation_start.jpeg`	Arrival at start line
`navigation_center.jpeg`	Arrival at arena center
`navigation_end.jpeg`	Return to start line
`navigation_final.mp4`	Complete navigation mission

---

1. Robot Construction Overview

This section presents the **physical construction of the robot** developed for **Mapping & Navigation**, with emphasis on the **mechanical structure**, **sensor placement**, and the **design considerations that guided the robot build**. The primary objective during construction was **not to maximize speed**, but to create a **stable and repeatable physical platform** capable of supporting accurate measurement of position and distance during the mapping task.

The robot was constructed as a single integrated platform designed to:

- move reliably along the arena perimeter,
- maintain consistent contact geometry with the environment, and
- support fixed, repeatable sensor mounting.

Particular attention was given to mechanical stability, symmetry of the drive system, and robustness against collisions, as these factors directly influence the quality and repeatability of experimental results.

This section focuses exclusively on the **robot's mechanical build and physical sensor integration**. Software behavior, control algorithms, and offline data processing are discussed in later sections of the report.

1.1 Robot Construction and Mechanical Considerations

Intended Design and Final Build

The robot was based on the **NXTPrograms "Explorer" design**, providing a stable differential-drive platform with symmetric wheel placement well suited for wall-following and perimeter mapping tasks. Minor structural substitutions were made due to available LEGO NXT components, without affecting mechanical stability or navigation performance. The same robot platform was used for both mapping and navigation to maintain geometric consistency.

The only intentional deviation from the reference design was the use of a **fixed ultrasonic sensor instead of a moving mechanism**. This choice reduced mechanical complexity and ensured a consistent sensing geometry, improving measurement repeatability during wall-following–based mapping. The resulting robot was mechanically stable and performed reliably throughout both mapping and navigation phases.

1.2 Sensor Configuration and Roles

Ultrasonic Distance Sensor

The ultrasonic sensor was the primary sensor used for **wall following** and **corner detection**, and it played a central role in both real-time control and offline map reconstruction.

The sensor was mounted on the **left side of the robot**, facing **laterally toward the wall** during clockwise motion around the arena, at an approximate height of **15 cm above the ground**. The sensor was rigidly fixed to the chassis and was not movable.

This placement and height were selected based on both theoretical considerations and empirical testing:

- **Avoiding floor reflections:**
  Mounting the sensor too close to the ground increases the likelihood of reflections from the floor surface, producing unstable or artificially short readings. A height of approximately 15 cm significantly reduced these effects while remaining well within the vertical extent of the arena walls.

- **Stable interaction with wall surfaces:**
  At this height, the ultrasonic beam typically intersects a uniform region of the wall, avoiding irregularities near the floor such as seams or joints that distort measurements.

- **Improved repeatability across runs:**
  Fixing both the position and height of the sensor ensured that the geometric relationship between the robot and the wall remained constant across experiments. This repeatability was essential for comparing logged datasets and for reliable offline reconstruction.

- **Clear corner detection behavior:**
  At 15 cm height, distance transitions at corners were more pronounced and easier to distinguish from random noise, improving the robustness of rule-based corner detection.

- **Mechanical robustness:**
  The selected height reduced the likelihood that the sensor would be affected during collisions or bumper recovery maneuvers.

Tradeoff:
Placing the sensor higher reduces sensitivity to small wall features near the ground, but slightly increases sensitivity to wall tilt or non-vertical surfaces. For this task, the benefit of reduced noise and improved measurement stability clearly outweighed this limitation.

Touch (Bumper) Sensor

The touch sensor was mounted at the **front of the robot**, integrated into the bumper region, at an approximate height of **3–4 cm above the ground**.

This low placement was chosen to ensure:

- **Reliable detection of frontal collisions** with the wall
- **Activation before other parts of the chassis** contact the environment
- **Consistent triggering**, even in the presence of small vertical misalignments

Although collision recovery introduces odometry error due to wheel slip, the touch sensor significantly improves experimental robustness by preventing the robot from becoming stuck or damaged. Collisions were therefore treated as unavoidable but manageable events rather than failures of the system.

Light Sensor

The light sensor was mounted on the **underside of the robot**, facing downward toward the floor, at an approximate height of **1.5–2 cm above the ground**.

This height was selected to:

- **Maximize contrast** between the dark reference lines and the arena surface
- **Reduce sensitivity to ambient lighting variations**
- **Ensure repeatable detection** of the start and end reference point

When the black line was detected, the robot stopped, moved a short fixed distance forward, and then stopped completely. This procedure reduced sensitivity to partial line detection and improved the consistency of the final stop position, which in turn reduced uncertainty in loop-closure handling.

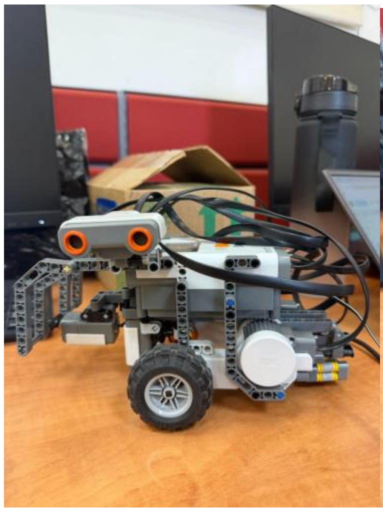
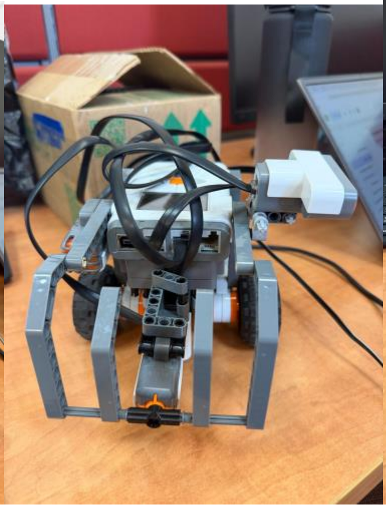
System-Level Perspective

Overall, sensor placement decisions prioritized **repeatability and stability over geometric complexity**. Fixed sensor mounting, conservative motion, and explicit handling of non-ideal events enabled reliable offline mapping despite sensor noise and mechanical non-idealities.

Reference:

https://www.nxtprograms.com/explorer/index.html

https://www.nxtprograms.com/bumper_car/

Part 1 – Arena Perimeter Mapping

Robot (Wall Following + Odometry + Logging)

1. Task Goal and Requirements (Mapping Phase)

The objective of the mapping phase was to generate a dataset that enables reconstructing an **unknown arena perimeter** offline.

Arena constraints (given):

- Closed, **convex polygon**

- Built from **straight walls only** (up to **8**)
- Corner angles between **70° and 170°**
- Robot starts near a wall, oriented for **clockwise motion**, and must complete a full lap until reaching the **start region marked by dark lines** (two black strips)

What the robot must provide to the PC:

1. **Robot trajectory** estimate (from odometry)
2. **Wall distance measurements** (ultrasonic) synchronized with the trajectory

A key system-level insight learned early in development was that **mapping quality is dominated by data quality**. If the robot logs inconsistent pose or distance samples, no offline algorithm (RANSAC, least-squares, or Hough-based methods) can reliably reconstruct the true arena geometry. Offline processing can only fit the data that the robot actually measures.

---

2. Development Strategy

Part 1 (Mapping) — Step-by-Step: What we did

Step 0 — Define what "success" means

- Robot must do **clockwise full lap** near the wall.
- Must log **trajectory + US distance** synchronized.
- Must stop reliably at **start mark (black lines)** so loop closure is possible.

Phase A — Make the robot drive the arena (no data yet)

Step 1 –Home mini-arena test (before lab meeting)

What we did

Built a small test arena at home to start a minimal working code and trials.

Step2 – Build the simplest wall follower

What we implemented

- Left ultrasonic sensor gives distance to wall.
- Simple PID correction changes left/right motor speeds.

Why

- If the robot can't physically keep the wall → no map is possible.

What we checked

- Does it keep ~constant distance?
- Does it survive corners?
- Does it get stuck when touching wall?

Step 3 — Add bumper recovery

What we added

- If touch sensor pressed:
    1. stop
    2. reverse a short time
    3. turn away from wall
    4. reset PID state

Why

- Without a deterministic recovery, the robot scrapes, slips, and destroys odometry + distance samples.

Phase B — Start collecting data (first logging attempts)

Step 4 — Add "basic logging" (time + encoders + US)

This is our second code version (`walldata.csv`):

- `time_ms, encL, encR, us_cm`

Why we did it

- We need raw data to build the map offline.

What happened

- Robot ran, file existed, but when we plotted and tried fitting walls:
    - points were messy near corners
    - map shape changed between runs
    - we didn't know if the problem was Python or the dataset

---

Phase C — Switch strategy: improve dataset quality at the source

Step 5 — Decide: compute pose onboard (high-precision odometry)

Change
Instead of logging only encoders and rebuilding pose later, we updated odometry inside NXC:

- convert encoder degrees → mm
- update `(X, Y, Theta)` every loop

Why

- Offline integration gets harmed by timing jitter + special maneuvers.
- Onboard pose makes the dataset interpretable and consistent.

Step 6 — Stabilize timing (control loop discipline)

What we enforced

- Fixed loop time: `LOOP_MS = 50 ms`
- Derivative uses the same dt every time

Why

- PID (especially D term) becomes unstable if dt is not consistent.
- Logging too often causes file I/O delays → dt jitter.

Step 7 — Upgrade the log format (richer + readable)

We changed logging to `walldata.csv`:
`Time, X, Y, Theta, US, Touch, Mode`

Why
Now we can later answer:

- Was this point during normal wall following?
- Or during collision recovery?
- Or turning a corner?
  Those are totally different "data quality" segments.

Step 8 — Add "Mode tags" (the big debugging upgrade)

What we logged

- `PID` for normal samples
- `OUTER / OUTER_END`
- `INNER / INNER_END`
- `COLLISION / COLLISION_END`
- end markers: `BLACK_LINE_DETECTED`, `END_AFTER_20CM`

Why

- Corner/recovery points are often outliers.
- Mode lets us filter / down-weight them offline instead of ruining line fits.

Phase D — Make corners consistent (so walls don't mix)

Step 9 — Corner detection using multiple rules + cooldown

Rules we used

- Outer corner:
  - distance becomes very large (`> 45cm`)
  - or sudden jump `delta > 6cm` when already far
- Inner corner:
  - repeated "too close" readings (`closeCount >= 4`)
  - or distance below near threshold (`< 15cm`)
- Cooldown counter prevents double-triggering.

Why

- A single threshold fails because US is noisy and corners vary.
- Multi-rule + cooldown gave repeatable corner behavior.

Step 10 — Tune corner thresholds using real logs

What we did

- Recorded runs
- Plotted US vs time (US_log)
- Looked at typical corner patterns
- Picked realistic thresholds:
    - FAR ≈ 45cm
    - NEAR ≈ 15cm
    - DELTA ≈ 6cm
    - close repetition count = 4

Why

- “Theoretical” thresholds didn’t match real sensor behavior.

Phase E — Make start/end detection reliable (loop closure help)

Step 11 — Add light sensor stop on black lines

Logic

- If `light < threshold` → stop
- log `BLACK_LINE_DETECTED`

Why

- Ultrasonic patterns repeat in a polygon → cannot reliably identify start.
- Light marker is stable and repeatable.

Step 12 — Add "move after black line" (repeatability trick)

After detecting the line:

1. stop + settle
2. move forward a fixed distance (~15–20cm)
3. stop and close file (`END_AFTER_20CM`)

Why

- Stopping exactly on the edge of the line is inconsistent.
- Moving a fixed distance makes the final pose more repeatable.
- Helps offline closure: end condition becomes more consistent across runs.

Phase F — Final validation run (what we checked)

Step 13 — Run full lap in the real arena

We verified

- Robot keeps distance ~18cm most of the time

- Doesn’t get stuck at corners
- Collision recovery returns to stable wall following
- File is saved and includes correct columns + Mode tags
- End detection triggers once and stops consistently

What changed between the 3 code versions (summary)

1. **Wall following only** → “robot can drive”
2. **Wall following + basic logging** → “we have data but it’s ambiguous”
3. **High precision odometry + Mode tags + corner tuning + light stop** → “data is structured and repeatable”

Part 2 – Offline Map Construction (Python)

3. Map Construction on PC

The purpose of the PC-side processing stage is to convert the raw data collected by the robot during the mapping run into a **clean, closed, and repeatable geometric map** of the arena. This map must satisfy the assignment constraints: a **convex polygon composed of straight walls only**, with **3–8 sides**, and be accurate enough to support autonomous navigation.

Unlike the robot-side code, which focuses on real-time control and data collection, the PC-side processing focuses on **geometric interpretation**, **noise reduction**, and **deterministic map reconstruction**.

3.1 Processing Pipeline Overview

The Python mapping script processes the robot log file through a **fixed five-stage pipeline**, where each stage addresses a specific source of error or ambiguity:

1. **Data Loading and Preprocessing**
   Reading the CSV file generated by the robot and filtering invalid or unreliable measurements.

2. **Loop Closure Correction**
   Correcting accumulated odometry drift so that the trajectory forms a closed loop.

3. **Wall Point Generation**
   Converting ultrasonic distance measurements into global wall coordinates.

4. **Wall Segmentation and Line Fitting**
   Separating wall points into individual walls and fitting a line to each wall.

5. **Polygon Construction and Center Calculation**
   Building the final convex arena polygon and computing its geometric center.

The output of this pipeline is a stable arena representation that can be reused consistently for navigation.

3.2 Data Loading and Preprocessing

Input Data

The robot produces a CSV file (`walldata.csv`) containing the following columns:

- **Time** – timestamp of the measurement
- **X, Y** – robot position (from onboard odometry)
- **Theta** – robot heading angle
- **US** – ultrasonic distance measurement
- **Touch** – bumper state
- **Mode** – robot behavior state (PID, corner handling, collision, etc.)

Why preprocessing is necessary

Raw robot measurements are not directly suitable for line fitting because they include:

- ultrasonic noise and reflections (especially near corners),
- invalid values (e.g., 0 or 255),
- samples collected during non-stationary behaviors (turning, reversing, recovery).

If such samples are used unfiltered, line fitting becomes unstable and produces maps that may change noticeably between runs—an unacceptable outcome for navigation.

Filtering steps applied

- **Mode filtering:** Only samples collected during stable motion are used. Primary mode: `PID`, and end-of-event modes such as `OUTER_END`, `INNER_END`, `COLLISION_END` (moments when the robot has re-aligned with a wall).
  Critical reason: measurements recorded while turning or reversing do not represent a straight wall and tend to create false line hypotheses.

- **Ultrasonic validity filtering:** Readings outside a feasible range (e.g., 4–250 cm) and error codes (e.g., 255) are discarded.
  Critical reason: these values typically reflect sensor failure, floor reflections, or wall loss, not actual geometry.

- **Median filtering (window = 5):** Applied to the ultrasonic signal to remove isolated spikes while preserving sharp transitions.
  Critical reason: unlike averaging, the median filter suppresses outliers without smearing corner transitions.

- **Jump filtering:** Consecutive readings that differ by more than ~20 cm are rejected.
  Critical reason: these jumps are strongly correlated with reflections and momentary wall loss and can dominate line-fitting residuals.

These steps substantially improve the quality of the wall point cloud before any geometric estimation is performed.

3.3 Loop Closure Correction

The odometry drift problem

Odometry-based position estimation accumulates error over time.
After completing a full lap around the arena, the robot’s final estimated position typically does not coincide with its starting position.

In our experiments, the start–end mismatch was approximately **16 cm** before correction.

Why loop closure is mandatory

Without loop closure:

- The trajectory is not closed,
- Wall points from the same physical wall do not align properly,
- Line intersections produce distorted or open polygons.

Correction method

A **linear drift distribution** method is applied:

1. Compute the drift vector between the final and initial positions.
2. Distribute this drift gradually along the trajectory based on time.
3. Early samples receive almost no correction, while late samples receive nearly full correction.

Mathematically:

$$w_k = \frac{t_k - t_0}{t_{end} - t_0}, \quad x'_k = x_k - w_k \Delta x, \quad y'_k = y_k - w_k \Delta y$$

Why this works

Odometry drift accumulates approximately proportionally to distance traveled.
A linear correction is therefore a reasonable and simple approximation that enforces **perfect closure**, which is more important for map consistency than preserving local millimeter accuracy.

3.4 Wall Point Generation

Each ultrasonic measurement is converted into a global wall point.

Geometric transformation

The ultrasonic sensor is mounted facing **left**, perpendicular to the robot's forward direction.
For each robot pose (x,y,θ), the wall direction angle is:

$$\alpha = \theta + \frac{\pi}{2}$$

The wall point is computed as:

$$x_w = x + (d_{US} + d_{set})\cos(\alpha), \quad y_w = y + (d_{US} + d_{set})\sin(\alpha)$$

Where:

- **dUS** is the measured ultrasonic distance,
- **dset** is the wall-following setpoint distance.

Why the setpoint distance is added

The robot does not drive directly against the wall, but at a fixed offset from it.
If this offset is ignored, the reconstructed arena would be uniformly shrunk inward.

Adding the setpoint distance ensures that wall points correspond to the **actual physical wall location**, not the robot's path.

The result of this step is a **noisy wall point cloud**, which still requires segmentation and fitting.

3.5 Initial Approach: RANSAC Line Fitting (Why It Failed)

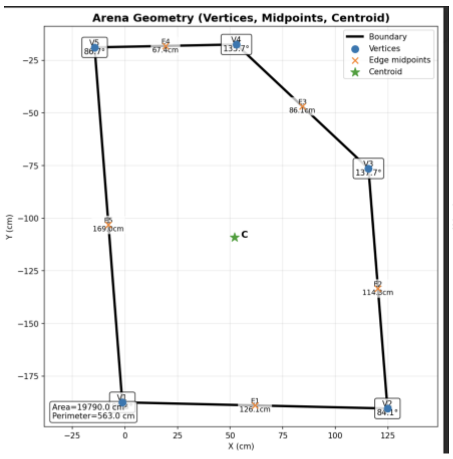
RANSAC was initially selected because it is widely used for model fitting in the presence of outliers. However, in our experiments it produced **inconsistent maps across repeated executions on the same dataset**.

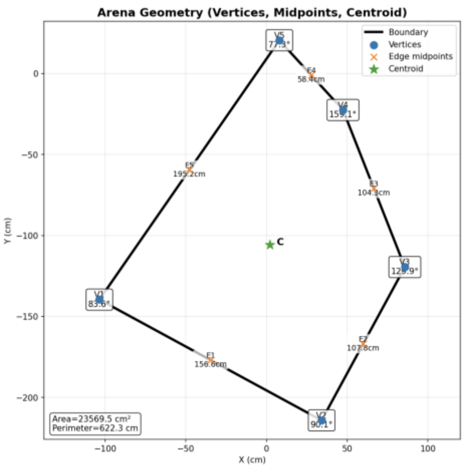
Key observed reasons:

- **Stochastic behavior:** different random samples lead to different solutions.
- **Corner transitions:** points near corners violate the single-line assumption and confuse inlier selection.
- **Similar wall orientations:** near-parallel walls can compete for inliers.
- **High parameter sensitivity:** small changes in thresholds (inlier radius, minimum points) change the result.

Critical conclusion

Although RANSAC is statistically robust, it is not deterministic. For autonomous navigation, **repeatability and stability** are more important than probabilistic robustness, so a deterministic pipeline was preferred.





3.6 Final Method: Deterministic Clustering and Line Fitting

To guarantee stable results, the final pipeline replaces RANSAC with a fully deterministic approach.

3.6.1 Wall Segmentation using DBSCAN

DBSCAN groups wall points into spatial clusters, where each cluster represents one wall.

Why DBSCAN

- Deterministic: same input → same output
- Explicit noise handling
- No need to specify number of walls in advance

Parameters

- Epsilon: 16 cm
- Minimum samples: 5
- Clusters smaller than 15 points are discarded as noise

3.6.2 Line fitting using Total Least Squares

For each wall cluster, a straight line is fitted using **Total Least Squares (TLS)** via SVD.

Why TLS
Ordinary least squares minimizes vertical error, which is inappropriate for walls with arbitrary orientation.
TLS minimizes **perpendicular distance**, which is geometrically correct for wall fitting.

3.6.3 Wall merging

Clusters representing the same physical wall may appear split due to gaps or noise.

Two walls are merged if:

- Angle difference ≤ 8°
- Offset difference ≤ 10 cm

The wall with more supporting points is retained.

3.7 Polygon Construction and Arena Center

Polygon construction

- Intersections between wall lines are computed.
- Only intersections lying inside all wall half-planes are kept.
- A convex hull is applied to guarantee convexity.
- Vertices are ordered counterclockwise.

The result is a closed convex polygon with 3–8 vertices.

Arena center

The arena center is computed as the **area-weighted centroid** of the polygon.

Why not average of vertices?
The centroid of a polygon reflects the true geometric center, even when edges are unevenly distributed.

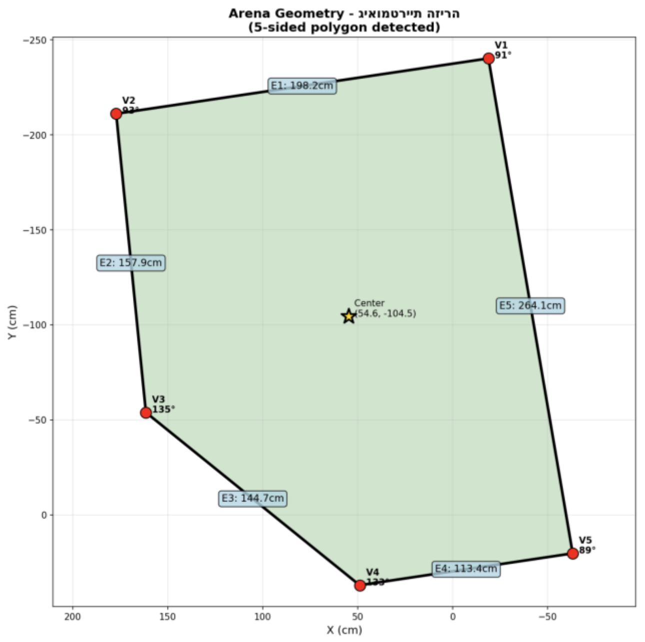
This center point is used directly by the navigation algorithm.

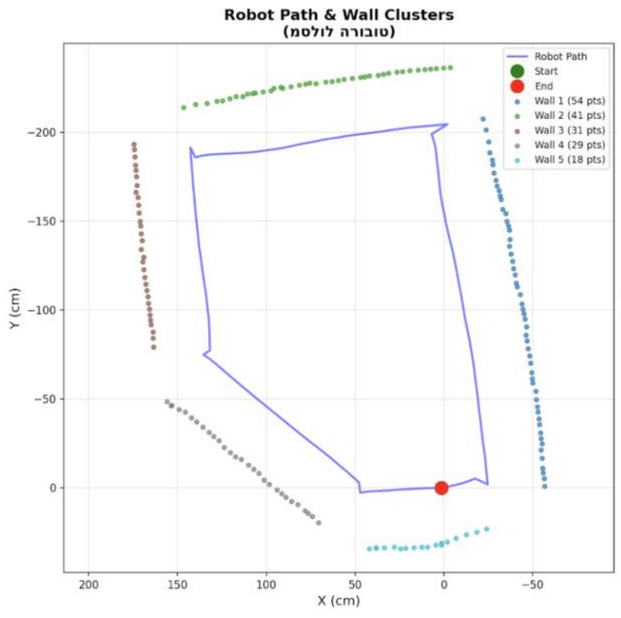
3.8 Discussion and Design Rationale

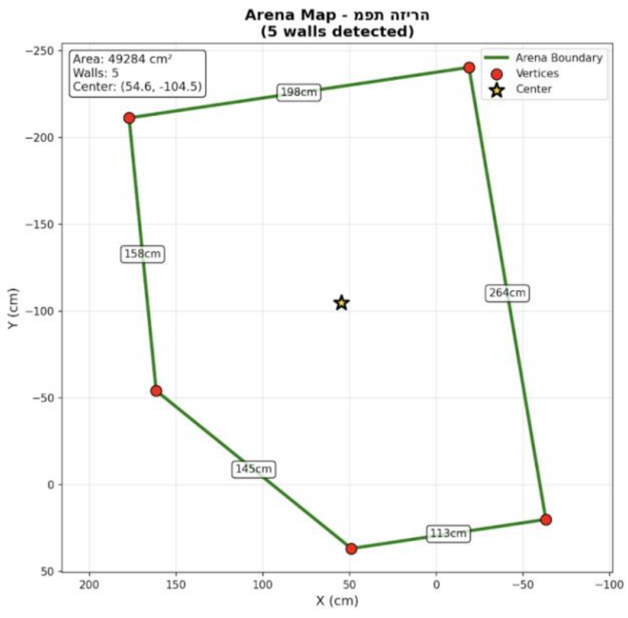
The main insight from this stage is that **determinism is critical in robotic mapping**.
A theoretically powerful but stochastic method (RANSAC) proved inferior to a simpler, deterministic pipeline when consistent results were required.

Hardware imperfections such as ultrasonic noise and odometry drift were not eliminated, but **systematically compensated** through filtering, loop closure, clustering, and constrained geometry.

This approach resulted in a **repeatable, stable arena map** suitable for autonomous navigation.







References

https://csc.csudh.edu/btang/seminar/slides/DBSCAN.pdf

https://mowgli.jmc.ac.il/mod/resource/view.php?id=1235131

https://mowgli.jmc.ac.il/mod/resource/view.php?id=1235133

AI tools (ChatGPT) were used to:

- Improve clarity of explanations,
- Verify mathematical consistency,
- Compare alternative estimation methods.
- Coding Logic.

3.9 Quantitative Error Evaluation (Additional Python Analysis)

Motivation

While visual inspection of the reconstructed arena map provides intuition, it is insufficient for objectively evaluating mapping accuracy and repeatability. To address this, an **additional Python evaluation script** was developed to compute quantitative error metrics across multiple independent mapping runs.

This evaluation was not required explicitly by the assignment, but was implemented to:

- verify loop-closure quality,
- quantify geometric accuracy of reconstructed walls,
- and assess inter-run consistency of the mapping pipeline.

Experimental Setup

Four independent mapping runs were performed under identical conditions:

Run ID	Data File
Run 1	walldata.csv

Run 2	walldata1.csv
Run 3	walldata2.csv
Run 4	walldata3.csv

Each run completed a full clockwise traversal of the arena perimeter and produced a reconstructed convex polygon with **5 walls**, consistent with the physical arena.

3.9.1 Loop Closure and Global Consistency

The estimated geometric center of the arena was computed for each run. Table 1 summarizes the reconstructed arena center and total enclosed area.

Table 1 – Arena Center and Area per Run

Run	Center X (cm)	Center Y (cm)	Area (cm²)
walldata.csv	52.43	-105.72	48,465
walldata1.csv	53.22	-105.37	48,488
walldata2.csv	43.09	-110.98	48,911
walldata3.csv	54.64	-104.51	49,284

Interpretation

- The arena center varies by only a few centimeters across runs.

- Area estimates differ by less than **2%**, indicating strong global consistency.
- These results confirm that loop-closure correction successfully compensates for accumulated odometry drift.

3.9.2 Wall Length Accuracy

Each reconstructed wall length was compared against the known reference dimensions of the arena.
Table 2 presents the mean reconstructed wall length and the standard deviation across runs.

Table 2 – Wall Length Consistency Across Runs

Wall	Mean Length (cm)	Std Dev (cm)
Wall 1	196.95	2.85
Wall 2	157.94	0.98
Wall 3	145.03	1.17
Wall 4	111.63	2.94
Wall 5	263.20	1.28

Interpretation

- Standard deviation for all walls is below **3 cm**, demonstrating high repeatability.
- Longer walls exhibit slightly higher variance, consistent with odometry error accumulating over distance.

- The stability confirms that deterministic clustering and Total Least Squares fitting are well suited for this task.

3.9.3 Wall Fitting Error Metrics

To quantify fitting quality, two standard error metrics were computed for each run:

- **RMSE** – Root Mean Square Error between wall points and fitted lines
- **MAE** – Mean Absolute Error

Table 3 – Wall Fitting Error per Run

Run	Edge RMSE (cm)	Edge MAE (cm)
walldata.csv	15.89	14.27
walldata1.csv	16.70	14.43
walldata2.csv	16.07	14.52
walldata3.csv	16.88	15.37

Interpretation

- Errors are consistent across runs, indicating deterministic behavior.
- RMSE values are dominated by ultrasonic noise near corners and during recovery maneuvers.
- The absence of large variance confirms that outliers are effectively handled by preprocessing and clustering.

3.9.4 Inter-Run Consistency Summary

Aggregating results across all runs yields the following consistency statistics:

Table 4 – Overall Mapping Consistency

Metric	Mean	Std Dev
Arena Center X (cm)	50.84	5.25
Arena Center Y (cm)	-106.65	2.93
Average Wall Length Error (cm)	≈ 15	< 3

These values confirm that the mapping pipeline produces **repeatable geometry** across independent runs, a critical requirement for reliable autonomous navigation.

Engineering Insight

This quantitative analysis validates a key conclusion of the project:

> Reliable mapping is achieved not by increasingly complex fitting algorithms, but by improving robot-side data quality and enforcing deterministic offline processing.

Despite noisy ultrasonic measurements and unavoidable odometry drift, the system achieves stable wall geometry through:

- structured logging with behavioral tags,
- explicit loop closure,
- deterministic clustering,

- and geometrically correct line fitting.

Part 3 – Navigation Algorithm (שלב הניווט)

4.1 Development Goals and Constraints

The navigation stage required the robot to execute a complete mission in **under 7 minutes**, starting from an **unknown position and heading** near the arena perimeter. The robot had to:

1. Reach the **start point** (double black line), align to the **arrow direction**, stop, announce arrival, and wait **30 seconds** (first 15 seconds ignore sound, next 15 seconds wait for clap).

2. Reach the **center of the arena**, stop, announce arrival, and wait again with the same protocol.

3. Return to the **start point**, align to the arrow direction, stop, and announce arrival.

This created a combined **accuracy + robustness** problem: odometry alone is not reliable when the initial pose is unknown, and drift accumulates during long motion. Therefore, we designed a **hybrid strategy**: use sensor-based behaviors to reach a known landmark, then use odometry for point navigation, and finally use sensor confirmation again to guarantee correct stopping.

4.2 Robot Development for Navigation (changes compared to mapping)

The same physical robot used for mapping was reused for navigation to keep the geometry and calibration consistent. However, several additions and software changes were implemented specifically for the navigation mission:

4.2.1 Adding the Sound Sensor (S3)

A sound sensor was added to **port S3** in order to detect a **clap command**. This sensor was not required during mapping, but was essential here due to the assignment requirement of waiting for a voice/sound command during checkpoints.

Why clapping?

- A clap produces a strong, short peak that is easier to distinguish from background noise than spoken words.
- It is repeatable between different operators.

4.2.2 Separate Navigation Program

The navigation code was implemented as a **separate NXC program** from the mapping program.

Why separation was necessary:

- Mapping focuses on **data logging and stable wall following**.
- Navigation focuses on a **state machine**, interactive waits, and point-to-point motion.
- Keeping the programs separate reduced complexity and lowered the risk of failure during the timed lab test.

4.2.3 Stop-and-Settle at Checkpoints

A “hard stop + short settle delay” was added before any odometry reset or waiting stage.

Why:

- Motors may coast slightly after stop commands.
- If odometry is reset while the robot is still moving, the “(0,0)” reference becomes incorrect and creates systematic navigation error.

4.3 Map Transfer Format (what was transferred and why)

The Python mapping stage outputs detailed map data: polygon vertices, wall line equations, plots, and derived geometry. However, for navigation the robot only needs to travel to:

- Start line → defined as **(0,0)** after reset
- Center → **(Cx, Cy)**

- Return → **(0,0)**

Therefore, only the **arena center coordinates** were transferred from Python to the NXC code:

- **Cx = -54.64 cm**
- **Cy = 104.51 cm**

4.3.1 Why the center only

Using the full polygon on the robot would require:

- More memory,
- More computations,
- More risk (parsing files, numeric instability),
  with no major benefit, because the navigation path does not require full localization against all walls.

4.4 Improving the Mapping Code to Support Navigation (key development story)

Before navigation worked consistently, the mapping output needed to become stable. Early versions used line fitting approaches that produced **different shapes and measurements** between runs, which caused the center point to shift. This directly led to navigation failures: the robot was sometimes “correct” relative to the provided center, but the center itself was inconsistent.

4.4.1 Main failure of early approach

- Each run produced a slightly different wall set → different polygon → different center.
- Even small center differences caused large error in the robot’s final location because of long travel distances.

4.4.2 Final solution (stable wall extraction)

The final mapping pipeline was modified to make wall extraction consistent:

- Noise filtering and smoothing (median + jump clamp)
- Loop closure correction to reduce odometry drift
- **DBSCAN clustering** to group wall points consistently
- TLS line fitting per cluster
- Merge similar wall lines
- Polygon construction using line intersections + convex hull ordering

This significantly reduced run-to-run map variation and provided a consistent center estimate.

4.5 Final Navigation Strategy (high level design and why it worked)

The final strategy was a **hybrid** because no single method solves the entire task:

4.5.1 Stage 1 — Localization (wall-follow until double line)

The robot follows the wall using PID + ultrasonic sensor until it detects the double black line using the light sensor.

Why:

- Initial pose is unknown.
- The start line is a known landmark and provides a reliable reference.

4.5.2 Stage 2 — Reset coordinate system at start

After reaching the double line, the robot stops, stabilizes, and resets odometry to:

- **(x, y) = (0,0)**
- heading fixed to a defined direction (aligned to arrow)

Why:

- This converts an unknown global problem into a local relative navigation problem.

4.5.3 Stage 3 — Navigate to center using odometry

Robot rotates toward (Cx, Cy) and drives in segments, recomputing angle periodically.

Why segments instead of one long drive:

- Reduces drift accumulation.
- Allows small heading corrections during movement.

4.5.4 Stage 4 — Return to start (hardest part)

Initially, the robot attempted to return purely by odometry, but it often ended near the wrong location due to drift and turning inaccuracies.

Final working method:

1. Odometry return brings the robot approximately toward the start zone.
2. Robot reacquires the wall and uses **PID wall-following** again.
3. Light sensor detects the double line precisely and stops.

Why this works reliably:

- Odometry is good to reach "near".
- Wall-follow + line detection is best for "exact stop".

This approach produced repeatable success and stayed well under the 7-minute time limit.

4.6 Timing and Parameter Choices (why we tuned them)

The mission must finish under 7 minutes, but also stop accurately at each target.

Parameter	Value	Reason
Arrival tolerance	6 cm	avoids oscillation and saves time
Rotation tolerance	~8°	enough for accuracy without long micro-adjustments
Segment length	~40 cm	correct heading periodically, reduce drift
Wait time	30 s	assignment requirement (15 ignore + 15 listen)

During repeated tests the full mission typically completed in ~115 seconds (excluding the required waiting times), leaving a large safety margin.

---

5. Conclusion

In this we implemented a complete pipeline for **arena mapping and autonomous navigation** using a LEGO NXT robot. The mapping stage produced a time-synchronized dataset (odometry + ultrasonic distance) collected during a full clockwise perimeter traversal. The PC-side processing reconstructed a **closed convex polygon map** using deterministic preprocessing, loop-closure correction, DBSCAN-based wall segmentation, and Total Least Squares line fitting. The navigation stage then used a hybrid strategy—**landmark-based localization (double black line) + odometry-based point navigation + sensor confirmation**—to reliably reach the arena center and return to the start within the required time constraints.

A key engineering outcome is that **repeatability depends primarily on structured data collection and deterministic processing**. By improving robot-side logging (pose + mode tags) and enforcing a deterministic map reconstruction pipeline, we obtained stable center estimates across multiple runs, enabling consistent navigation performance.

---

5. AI Usage Statement

AI tools (ChatGPT & Perplexity) were used to improve documentation quality and assist with minor code refinements. All data collection, system design, implementation, testing, and final verification were performed by the authors, who take full responsibility for the project.