IMU-Based Pedestrian Positioning Laboratory (AAE4203)

Laboratory Date: 3rd November 2025, 8:30–11:30
Lab Report Due Date: Before 24th November 2025

The Hong Kong Polytechnic University
Department of Aeronautical and Aviation Engineering
2025/26 Semester 1

THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

Contents

1 Lab Introduction

Pedestrian Dead-Reckoning (PDR) is a fundamental technique for indoor and pedestrian positioning, especially in environments where satellite-based navigation (GNSS) is unavailable or unreliable. PDR estimates a person's position by detecting individual steps and calculating displacement based on step length and heading direction. This step-based navigation method is robust against cumulative sensor drift and provides valuable experience in integrating sensor data, algorithm development, and practical positioning applications.

The PDR laboratory focuses on data collection using portable devices and algorithm implementation with AI-assisted programming. Students are expected to collect motion data from a smartphone IMU, process the data using a PDR algorithm, and evaluate the resulting trajectories. The expected learning outcomes of this lab are as follows:

- **Objective 1:** Acquire practical skills in collecting IMU data from a smartphone or other portable devices, including understanding sampling rates, maintaining consistent device orientation, and properly logging data.
- **Objective 2:** Implement a PDR algorithm in Python, covering step detection, step length estimation, heading calculation, and signal processing techniques.
- **Objective 3:** Apply AI-assisted programming tools to support algorithm development, including generating code for data processing, visualization, and analysis.
- **Objective 4:** Evaluate and optimize PDR algorithm performance by visualizing trajectories, validating step detection, adapting dynamic thresholds, and understanding the influence of key parameters on positioning accuracy.
- **Objective 5:** Compile a concise lab report documenting: (1) PDR positioning results and trajectory visualizations, (2) analysis of algorithm performance and potential sources of error, and (3) reflections and insights gained from the laboratory exercises.

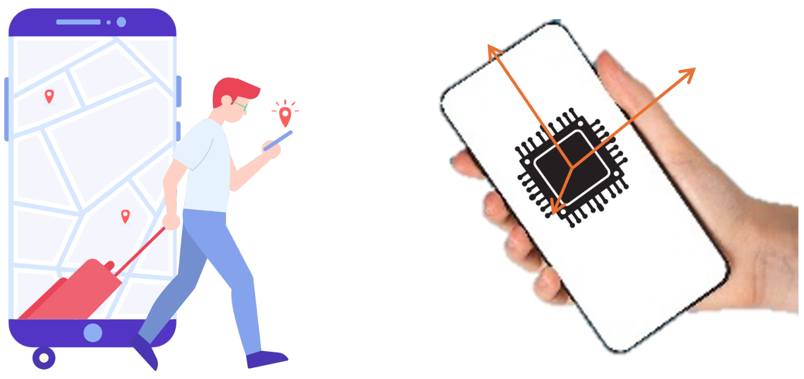


Figure 1: Smartphone-based pedestrian dead-reckoning.

2 Theoretical Background

2.1 Pedestrian Dead-Reckoning

PDR is a step-based navigation technique that estimates a pedestrian's position by detecting individual steps and calculating the displacement based on step length and heading direction. Starting from a known initial position, PDR incrementally updates the user's location in a 2D local navigation frame (e.g., East-North coordinates). Unlike continuous integration methods that directly accumulate raw sensor data, PDR is more robust to sensor drift because it operates on discrete step events, making it computationally efficient for indoor navigation.

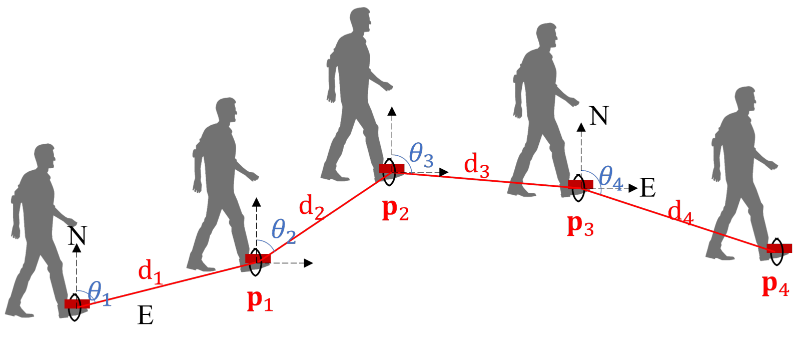


Figure 2: Illustration of PDR.

The core of PDR is a recursive update process. At the completion of the k-th step, the pedestrian's position $\mathbf{p}_k = [x_k, y_k]^T$ is obtained from the previous position $\mathbf{p}_{k-1}$ as follows:

$$\mathbf{p}_k = \mathbf{p}_{k-1} + \begin{bmatrix} d_k \sin(\theta_k) \\ d_k \cos(\theta_k) \end{bmatrix}, \tag{1}$$

where:

- $\mathbf{p}_{k-1}$ is the estimated position after the $(k-1)$-th step.
- d_k is the estimated length of the k-th step.
- θ_k is the estimated heading direction (yaw) during the k-th step, typically measured clockwise from North.

Therefore, an effective PDR algorithm requires three key modules:

- **Step Detection:** Identifying when a step occurs using accelerometer data.
- **Step Length Estimation:** Calculating the distance covered in each step.
- **Heading Estimation:** Determining the direction of movement using gyroscope data combined with orientation filtering.

2.2 Step Detection Algorithm

Human walking exhibits a characteristic acceleration pattern: during each gait cycle, the body undergoes a forward acceleration when a step is initiated and a deceleration upon landing. This periodic dynamic behavior can be effectively captured using inertial measurements. Therefore, step detection is typically performed on the magnitude of the acceleration vector, which eliminates orientation dependence and highlights the cyclic nature of human locomotion.

To remove the effect of device orientation, the raw three-axis accelerometer signals are combined into a single scalar magnitude:

$$a(t) = \sqrt{a_x(t)^2 + a_y(t)^2 + a_z(t)^2}, \tag{2}$$

where $a_x(t)$, $a_y(t)$, and $a_z(t)$ are the accelerations measured along the device's local x-, y-, and z-axes at time t. The resulting magnitude $a(t)$ captures the overall dynamic intensity of motion. During walking, alternating acceleration and deceleration phases produce distinguishable peaks and valleys in $a(t)$, which can be exploited for robust step detection.

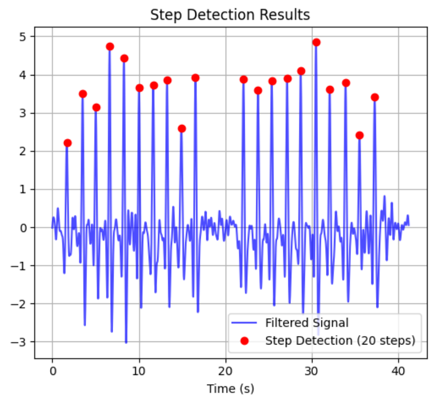


Figure 3: Step detection result.

The main processing pipeline is as follows:

a) **Band-pass Filtering:** The raw acceleration magnitude $a(t)$ is filtered to isolate frequencies characteristic of human walking (typically 0.5 Hz to 3 Hz). This removes the quasi-static gravity component and high-frequency noise. Let the filtered signal be $a_{\text{filtered}}(t)$.

b) **Peak Detection with Dynamic Thresholding:** Steps are identified as peaks in the $a_{\text{filtered}}(t)$ signal. To adapt to different walking speeds and device placements, a dynamic threshold $T_p(t)$ is calculated based on the signal's local statistics within a sliding window:

$$T_p(t) = \mu_{\text{local}}(a_{\text{filtered}}) + \alpha \cdot \sigma_{\text{local}}(a_{\text{filtered}}), \tag{3}$$

where μ_{local} and σ_{local} are the mean and standard deviation of the signal in the local window, and α is a sensitivity coefficient. A peak is a candidate for a step if its magnitude exceeds this threshold.

c) **Temporal Constraints:** To reject spurious peaks caused by random jitter, a temporal constraint is applied. The time interval between two consecutive detected steps, t_k and t_{k-1}, must be greater than a minimum physiological limit, $T_{\min}$ (e.g., 250 ms).

$$t_k - t_{k-1} \geq T_{\min}. \tag{4}$$

2.3 Step Length Estimation Model

The length of a step, d_k, is closely related to the dynamics of the user's movement. A commonly used empirical model estimates the step length based on the variation of the filtered acceleration magnitude within that step. This model can be expressed as:

$$d_k = d + s \cdot a_{\text{peak},k}, \quad d_k \in [0.3, 1.2]\text{m}, \tag{5}$$

where:

- $a_{\text{peak},k}$ denotes the peak value of the filtered acceleration magnitude, $a_{\text{filtered}}(t)$, observed during the k-th step cycle.
- d is the base step length, representing the minimum step size independent of acceleration variation, typically set to $d = 0.4$ m.
- s is a scaling coefficient that determines how strongly the acceleration variation affects the step length, typically set to $s = 0.03$.

Traditional PDR implementations often assume a fixed step length, which can lead to large cumulative errors because actual step sizes vary with walking speed, style, and terrain. In contrast, the above empirical model provides an **adaptive** step length estimation: more vigorous walking (larger acceleration swings) corresponds to longer steps, while slower walking produces shorter steps. This adaptability improves the robustness of PDR across different users and walking conditions.

2.4 Heading Estimation

The pedestrian's heading can be estimated by integrating the z-axis angular velocity measured by the gyroscope:

$$\theta_k = \theta_{k-1} + \int_{t_{k-1}}^{t_k} \omega_z(t)\, dt, \tag{6}$$

where:

- θ_{k-1} is the estimated heading at the previous step $(k-1)$,
- $\omega_z(t)$ is the angular velocity around the vertical axis in the sensor frame,
- t_{k-1} and t_k are the timestamps of the $(k-1)$-th and k-th steps, respectively.

This integration provides the cumulative heading change between consecutive steps, allowing the algorithm to track directional changes during walking. The **trapezoidal rule** ensures a simple yet accurate numerical approximation for discrete sensor measurements.

3 Laboratory Procedure

3.1 Programming with AI Tools

This section introduces practical methods for using AI tools to assist in programming tasks for PDR algorithm implementation. Emphasis is placed on transforming theoretical formulas into executable code, and on verifying and refining AI-generated code to ensure correct functionality.

3.1.1 Available AI Models for Programming Assistance

The following are several readily accessible AI models, which are either free to use or provide a free usage quota. Some of them also offer limited reasoning and online search capabilities. Each tool has unique features that can assist in programming, code generation, and debugging.

- **PolyU GenAI** – Internal AI platform provided by The Hong Kong Polytechnic University. `https://genai.polyu.edu.hk`
- **DeepSeek** – Web-based AI coding assistant developed in China, utilizing reinforcement learning techniques to improve code suggestions. `https://www.deepseek.com`
- **Grok** – AI coding assistant developed by xAI for code generation and intelligent suggestions. `https://www.grok.com`
- **GitHub Copilot** – AI-powered code completion tool developed by GitHub, integrated into mainstream IDEs. `https://github.com/copilot`
- **ChatGPT** – General-purpose large language model from OpenAI, capable of code generation and debugging; may require VPN. `https://chat.openai.com`
- **Gemini** – AI assistant supported by Google, specializing in programming and technical reasoning; may require VPN. `https://gemini.google.com`

Note that the above list is not exhaustive; many other AI models exist that can support programming tasks.

3.1.2 Preparation Prior to AI-Assisted Programming

Clear planning is essential before interacting with AI. This preparation ensures alignment between the student and the AI assistant. AI assistants serve as a support tool for planning and coding, but critical thinking and verification remain the responsibility of the student.

1) Selecting Programming Tools

- **Programming Language:** Python is recommended due to its scientific computing libraries such as NumPy (arrays), SciPy (scientific calculations), and Matplotlib (visualization). MATLAB is also suitable for academic and engineering applications. Students may also consult AI assistants, describing their specific requirements and application scenarios, to obtain recommendations for the most suitable programming language.

- **Development Environment (IDE):** Choose a comfortable IDE, e.g., VS Code, PyCharm, or Jupyter Notebook, to facilitate testing and debugging of AI-generated code. In addition to installing the environment, students should familiarize themselves with essential operations, including code compilation and execution, as well as installation and import of required dependencies (e.g., libraries or header files). These procedures can also be clarified with AI guidance.

2) Clarifying Goal: Inputs, Process, and Outputs

- **Input:** Specify the data required by the algorithm, e.g., accelerometer readings in a NumPy array with timestamps.
- **Process:** Describe the computational steps of the PDR algorithm, including step detection, step length estimation, heading estimation, and position updating.
- **Output:** Define the expected results, e.g., a list of (x, y) coordinates or a visualization of the walking trajectory.

3.1.3 Effective Prompt Engineering

The quality of AI-generated code is highly dependent on the clarity and completeness of the prompt provided. A vague or incomplete prompt may result in code that is irrelevant, inefficient, or incorrect. To ensure meaningful AI responses, it is important to clearly define the task, the context, and the expected output. A well-structured prompt typically contains the following key components:

- **Assign a Role:** Instruct the AI to act as an expert in a specific domain, e.g., scientific computing in Python.
- **Provide Context:** Describe the project, its objectives, and the current stage of development.
- **Define the Task:** Clearly specify the functionality or computation to be implemented.
- **Provide Specific Details:** Include all necessary information such as mathematical formulas, variable names, and data structures.
- **Specify Output Format:** Indicate the desired format of the code, such as a function, class, or complete script.
- **Add Additional Requirements:** Request the use of specific libraries, inclusion of comments, or explanation of the code logic.

Prompt Example

Consider the implementation of a simple position update algorithm for PDR. The core formulas are:

$$x_k = x_{k-1} + L_k \cdot \sin(\theta_k), \quad y_k = y_{k-1} + L_k \cdot \cos(\theta_k)$$

Bad Prompt (Vague): *"Write code for a PDR algorithm."*
This prompt lacks details regarding the programming language, input variables, computation steps, and expected output, which may lead to irrelevant or incorrect code generation.

Good Prompt (Specific & Actionable):
[**Role**] Act as an expert in Python for scientific computing.
[**Context**] I am writing code for a Pedestrian Dead Reckoning project. I already have the step length (L) and heading angle (theta) for each step. I now need a function to calculate and update the pedestrian's position based on this data.
[**Task**] Please write a Python function named `update_position`.
[**Specifics**] This function should accept the following arguments:

a) `prev_pos`: A tuple containing the previous position coordinates `(x, y)`.

b) `step_length`: The step length for the current step, L_k (a float).

c) `heading`: The heading angle for the current step, θ_k (a float, in radians).

The function should calculate the new position (x_k, y_k) based on the formulas:

$$x_k = x_{k-1} + L_k \cdot \sin(\theta_k), \quad y_k = y_{k-1} + L_k \cdot \cos(\theta_k)$$

[**Output Format**] The function should return a new tuple containing the calculated new position (x_k, y_k).
[**Extra Requirements**] Please use the NumPy library for the calculations and add detailed comments to the code explaining each step.

3.1.4 Collaborative Debugging with AI

AI-generated code is not always perfect. Encountering errors is a normal part of the process; the key is learning how to use the AI to help you fix them.

1) The Golden Rule of AI Debugging: Provide Full Context

When you encounter an error, don't just tell the AI, "Your code is wrong." You must provide three key pieces of information:

- **The Code You Ran:** Especially the part that caused the error.
- **The Full Error Message:** Copy the entire error traceback from beginning to end.
- **Your Intention:** Briefly explain what you expected the code to do.

2) Debugging in Action: An Example

Suppose the AI gave you the following code to process a list of step lengths, but with a subtle bug:

```python
import numpy as np

def calculate_trajectory(initial_pos, step_lengths, headings):
  path = [initial_pos]
  current_pos = initial_pos
  # Bug is in the next line
  for i in range(len(step_lengths) + 1):
    current_pos = update_position(
    current_pos, step_lengths[i], headings[i])
    path.append(current_pos)
  return path
```

When you run this with data like `step_lengths = [0.7, 0.75]`, it crashes with an `IndexError`:

```
Traceback (most recent call last):
  File "pdr_test.py", line 20, in <module>
    calculate_trajectory((0,0), step_lengths, headings)
  File "pdr_test.py", line 12, in calculate_trajectory
    current_pos = update_position(current_pos,
      step_lengths[i], headings[i])
IndexError: list index out of range
```

How to ask the AI for help:

I ran the `calculate_trajectory` function you provided, but I encountered an error.

[Here is the code I ran]

```python
def calculate_trajectory(initial_pos, step_lengths, headings):
  path = [initial_pos]
  current_pos = initial_pos
  for i in range(len(step_lengths) + 1):
    current_pos = update_position(
    current_pos, step_lengths[i], headings[i])
    path.append(current_pos)
  return path
```

[Here is the full error message]

```
Traceback (most recent call last):
  File "pdr_test.py", line 12, in calculate_trajectory
    current_pos = update_position(current_pos,
      step_lengths[i], headings[i])
IndexError: list index out of range
```

[My Intention] I expected this function to iterate through the `step_lengths` and `headings` lists to calculate the full trajectory. It seems like the loop is running one too many times. Can you please help me fix it?

3.1.5 Verification and Refinement

Even after AI-assisted code generation and initial debugging, the task remains incomplete. Rigorous verification and systematic refinement are essential to ensure correctness, reliability, and maintainability.

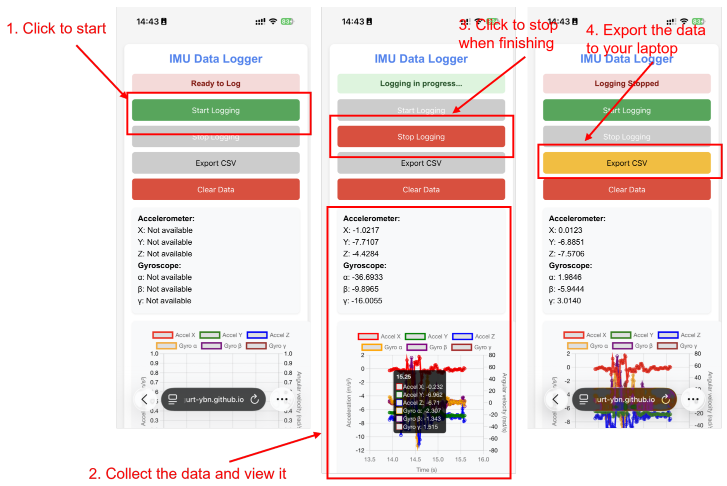
- **Verification of Test Data:** Use a simple dataset with known expected outputs to validate the correctness of the algorithm, either manually or automatically.
- **Comparison of Results:** Compare the program output with theoretical expectations or manually computed results to ensure consistency and correctness. Any discrepancies should prompt a review of the algorithm and its implementation.
- **Code Refinement and Optimization:** After confirming correctness, students may request further improvements, such as optimizing for performance, refactoring for readability, or enhancing maintainability.
- **Documentation and Summary:** Maintain a detailed record of prompts, AI responses, and final working code. This serves both as project documentation and a valuable learning resource.

Following these practices allows students to effectively leverage AI assistance while ensuring reliable, accurate, and well-documented code.

3.2 Collecting IMU Data with Smartphone

To facilitate the collection of IMU data, a simple web-based tool is provided at AAE4203Lab-IMU Data Logger. By opening this website in a mobile browser, sensor data from the smartphone can be recorded and exported for further processing.

3.2.1 Using the Smartphone to Collect Data



Accelerometer:
X: Not available
Y: Not available
Z: Not available
Gyroscope:
α: Not available
β: Not available
γ: Not available

Accelerometer:
X: -1.0217
Y: -7.7107
Z: -4.4284
Gyroscope:
α: -36.6933
β: -9.8965
γ: -16.0055

Accelerometer:
X: 0.0123
Y: -6.8851
Z: -7.5706
Gyroscope:
α: 1.9846
β: -5.9444
γ: 3.0140

Figure 4: Smartphone web interface for IMU data collection.

The interface provides several buttons for data collection (see Figure 4):

- **Start logging**: begin recording IMU data.

- **Stop logging**: terminate the current recording session.
- **Export data**: save the collected data in `CSV` format. Please specify the target folder to ensure proper storage.
- **Clear data**: erase the temporary buffer of recorded data.

3.2.2 Guidelines for Walking Data Collection

To ensure the quality and usability of the collected dataset, please follow these steps:

- **Preparation**
  - **Path design:** Choose simple and repeatable walking patterns, such as:
    - a 5×5 meter square,
    - a 10-meter straight line (walking back to the starting point),
    - or L-shaped / rectangular trajectories.
  - **Ground truth:** For basic verification, return to the starting point after completing the path. For straight-line walking, a simple linear interpolation of distance can be used as reference.
  - **Phone handling:** Hold the phone consistently (e.g., in hand, pocket, or fixed on the body) and maintain a stable orientation during the entire walk.
- **Data Collection**
  - Open the webpage using the smartphone browser.
  - Tap **Start Logging** to begin recording IMU data.
  - Perform the walking experiment along the pre-designed path.
  - Tap **Stop Logging** once the trial is complete.
  - Use **Export Data** to download the recorded sensor data as a CSV file. Remember to specify a proper filename and storage location.
  - If necessary, use **Clear Data** to reset the log before starting a new trial.

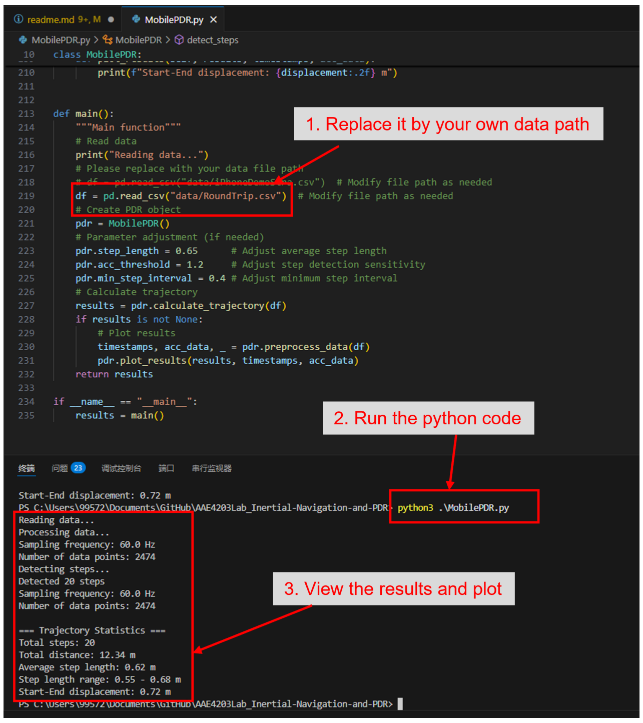
3.2.3 Alternative Option: Demo Data

If collecting data with a smartphone is difficult, a set of demo data is available at AAE4203Lab-Inertial-Navigation-and-PDR. This dataset can be used directly for practice and testing.

3.3 Implementing the PDR Algorithm

A reference implementation of the PDR algorithm is provided in the following GitHub repository: AAE4203Lab-Inertial-Navigation-and-PDR. Students can download this codebase and run the provided Python scripts directly.

After downloading, students need to replace the default dataset path with the location of their exported IMU data. Running the scripts with this data will generate results and plots, as illustrated in Figure 5. This procedure allows students to verify the algorithm and observe the estimated trajectory corresponding to their walking experiment.


readme.md 9+, M
MobilePDR.py
MobilePDR.py > MobilePDR > detect_steps
1. Replace it by your own data path
2. Run the python code
终端
问题 23
调试控制台
端口
串行监视器
3. View the results and plot

```python
class MobilePDR:
            print(f"Start-End displacement: {displacement:.2f} m")


def main():
    """Main function"""
    # Read data
    print("Reading data...")
    # Please replace with your data file path
    # df = pd.read_csv("data/iPhoneDemoS[illegible]a.csv")  # Modify file path as needed
    df = pd.read_csv("data/RoundTrip.csv")  # Modify file path as needed
    # Create PDR object
    pdr = MobilePDR()
    # Parameter adjustment (if needed)
    pdr.step_length = 0.65      # Adjust average step length
    pdr.acc_threshold = 1.2     # Adjust step detection sensitivity
    pdr.min_step_interval = 0.4 # Adjust minimum step interval
    # Calculate trajectory
    results = pdr.calculate_trajectory(df)
    if results is not None:
        # Plot results
        timestamps, acc_data, _ = pdr.preprocess_data(df)
        pdr.plot_results(results, timestamps, acc_data)
    return results

if __name__ == "__main__":
    results = main()
```

```
Start-End displacement: 0.72 m
PS C:\Users\99572\Documents\GitHub\AAE4203Lab_Inertial-Navigation-and-PDR> python3 .\MobilePDR.py
Reading data...
Processing data...
Sampling frequency: 60.0 Hz
Number of data points: 2474
Detecting steps...
Detected 20 steps
Sampling frequency: 60.0 Hz
Number of data points: 2474

=== Trajectory Statistics ===
Total steps: 20
Total distance: 12.34 m
Average step length: 0.62 m
Step length range: 0.55 - 0.68 m
Start-End displacement: 0.72 m
PS C:\Users\99572\Documents\GitHub\AAE4203Lab_Inertial-Navigation-and-PDR>
```

Figure 5: Procedure for running the PDR code locally.

A key aspect of working with the PDR algorithm is **parameter tuning and performance optimization**. The accuracy of trajectory estimation depends strongly on sensor characteristics and walking style, so students should adjust parameters based on the collected data to improve performance.

The algorithm includes several important parameters that influence its behavior:

- **Step Detection Sensitivity:** Controls how sensitive the algorithm is to accelerometer peaks. If set too low, extra steps may be falsely detected; if too high, some steps may be missed.

```
pdr. acc_threshold = 1.2
# Peak detection threshold
pdr. min_step_interval = 0.4
# Min time between steps
```

- **Step Length Parameters:** Determines how step length is estimated from the signal. These parameters should be tuned to match the walking pace and stride of different users.

```
pdr. step_length = 0.65
# Average step baseline
min_peak_height = 1.0
# Absolute minimum threshold
```

- **Signal Processing:** Includes settings such as filtering thresholds or smoothing factors. Proper tuning reduces noise and improves the stability of the estimated trajectory.

```
    window_size = int (0.5 * fs)
    # Dynamic threshold window
```

Careful tuning of these parameters can significantly reduce errors and enhance the stability of the estimated trajectory.

4 Thinking Questions

a) Does the orientation of the smartphone during data collection affect PDR algorithm performance? Why or why not?

b) How does heading estimation affect the performance of the PDR algorithm? What methods can be used to improve the accuracy of heading estimation?

References

[1] W. Kang and Y. Han, "SmartPDR: Smartphone-Based Pedestrian Dead Reckoning for Indoor Localization," *IEEE Sensors Journal*, vol. 15, no. 5, pp. 2906-2916, May 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6987239

[2] AAE4203 Lab Repository. [Online]. Available: https://github.com/RuijieXu0408/AAE4203Lab_Inertial-Navigation-and-PDR

[3] GitHub Docs, "Basic writing and formatting syntax," GitHub. [Online]. Available: https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax