Step Counting with Smartphone IMUs

Overview

Many real-world apps provide step counting as a core feature, such as Apple Health (iOS), Google Fit, Samsung Health, and activity-tracking apps like Strava. In this homework, your goal is to build the sensing core of a step-counting app using smartphone IMU data.

To keep the project focused on sensing and algorithm design (instead of full mobile app engineering), we intentionally separate **data collection** and **algorithm development**: you will collect data with phyphox, then implement and test your algorithm in Python. You will still support real-time streaming so your pipeline behaves like a live app.

You will implement a unified class interface that supports both offline testing and real-time streaming, build a real-time visualization demo, and write a short report describing your design and results. The target is a reasonable, robust, and interpretable system with solid baseline performance.

Quick Introduction: phyphox

phyphox (physical phone experiments) is a free smartphone app from RWTH Aachen University for physics and sensing experiments: `https://phyphox.org`.

In this homework, phyphox serves as your **data collection and live streaming tool**. It lets you:

- record phone sensor data (including IMU channels),
- export recordings to CSV for offline algorithm development, and
- stream live sensor data to your computer through the built-in Remote Access interface.

You will use it to mimic a real app pipeline while keeping implementation focused on signal processing and algorithm design.

Tasks

Complete all tasks below. Submit all requested deliverables.

1. **Data collection with phyphox.** You may collect and use IMU data channels (accelerometer, gyroscope, magnetometer), not only accelerometer. For standardized offline grading fairness, the test input is based on accelerometer signals (`time` + 3-axis `acc`), so your method must run correctly with accelerometer-only input. In practice, step counting usually relies on accelerometer

as the primary sensing signal. The phyphox "Accelerometer without g" experiment is recommended for cleaner motion signals: `https://phyphox.org/wiki/index.php?title=Sensor:_Acceleration_(without_g)`. Record at least one walking session for at least 1 minute (60 s or longer). Export the measurement as CSV and record the ground-truth step count by manual counting.

2. **Algorithm implementation.** Implement the required `StepCounter` class in `step_counter.py`. Your implementation must support both:
   - **Offline mode** via `run_offline(data)` for grading, and
   - **Online mode** via `update(data_chunk)` for real-time streaming.
3. **Real-time visualization demo.** Build an executable script or webpage that streams data from phyphox Remote Access (`https://phyphox.org/remote/`) and displays:
   - a real-time data curve, and
   - real-time step counting results, including immediate step detection and cumulative step count.
4. **Report.** Write a PDF report (maximum length is 5 pages) describing your data collection, algorithm,and results. Include screenshots of the demo. You must use th LaTeX template we provide.

Deliverables

Submit the following:

1. **Code:** `step_counter.py` implementing the required `StepCounter` class.
2. **Project file:** A zip file containing all files required to reproduce your results.
3. **Report:** a PDF report.
4. **Live demo:** A video demonstrating your real-time step counting and visualization.

Learning Objectives

By the end of this homework, you should be able to:

1. Apply basic signal processing to extract periodic step patterns.
2. Implement a step-counting method that works for both offline recordings and streaming data.
3. Evaluate accuracy and robustness across recordings.
4. Build a real-time visualization that demonstrates your algorithm behavior.
5. Compare different step-counting approaches and justify your design choices.

IMU Data and Baseline Step Counting Idea

What data you need

For standardized offline grading, the required sensing input is linear acceleration with timestamps:

- Timestamp t in seconds.
- Linear acceleration $a_x(t), a_y(t), a_z(t)$ in $\mathrm{m/s^2}$.

You may additionally use other IMU channels (gyroscope/magnetometer) in your own design and real-time demo, but the algorithm should still work with accelerometer-only input. In practice, accelerometer is usually the primary signal for step counting.

Linear acceleration means gravity has been removed from the raw accelerometer signal. A convenient phyphox source for this is "Accelerometer without g": `https://phyphox.org/wiki/index.php?title=Sensor:_Acceleration_(without_g)`.

Because phone orientation may change, a common orientation-robust signal is the acceleration magnitude:

$$a_{\mathrm{mag}}(t) = \sqrt{a_x(t)^2 + a_y(t)^2 + a_z(t)^2}.$$

Illustrative IMU patterns for steps

Figures 1, 2, and 3 show typical step-related IMU patterns (illustrative examples). Use them to connect signal features with algorithm choices; your own recordings may look different.

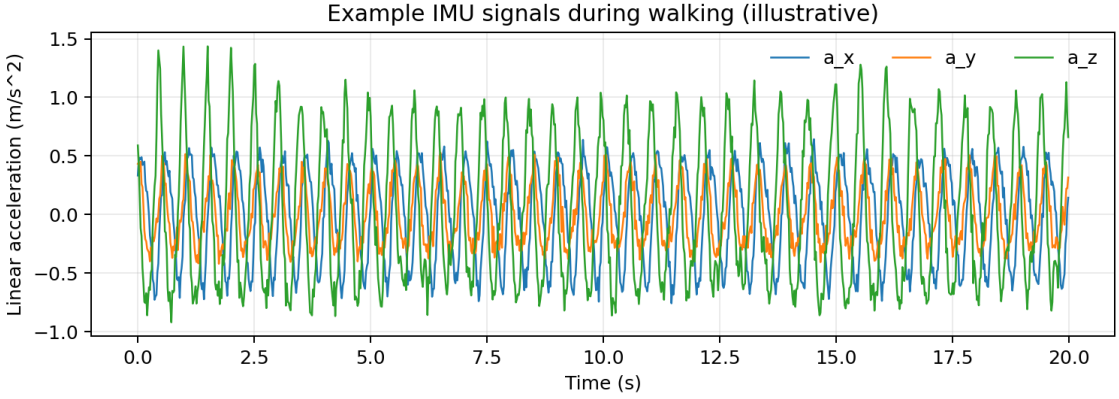

Figure 1: Illustrative tri-axis linear acceleration during walking

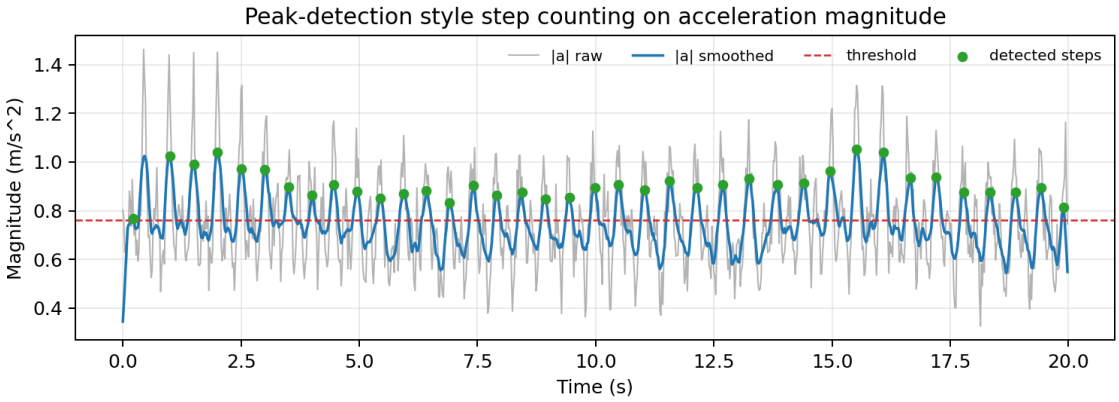

Figure 2: Acceleration magnitude, smoothing, thresholding, and detected step peaks

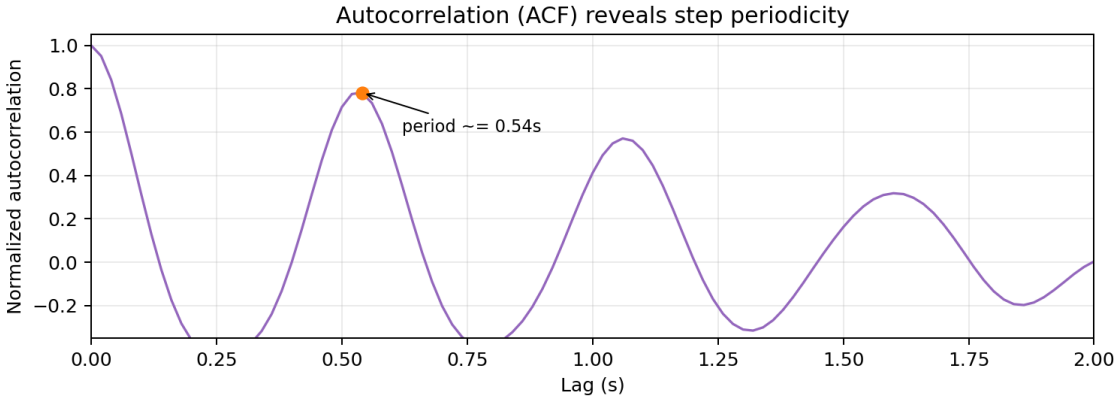

Figure 3: Autocorrelation (ACF) view showing periodic structure of walking

Common approaches you can try

- **Peak detection (time domain):** Smooth the signal, apply an adaptive threshold, and enforce a minimum peak distance.
- **Zero crossing:** Remove slow trends, then count stable crossings (often on a filtered axis or derivative) with refractory rules to avoid double counting.
- **Autocorrelation (ACF):** Estimate dominant periodicity from windowed ACF and convert period to step rate; combine with peak logic for robust timestamps.
- **IMU fusion (optional):** Use gyroscope/magnetometer to stabilize orientation estimates or reject artifacts, while keeping accelerometer-based counting as the core.
- **Neural-network methods:** As an optional extension, use a lightweight model (for example a small 1D CNN/RNN) trained from scratch on your own data. Pretrained models are not allowed, and your final system must still satisfy the required interface and real-time behavior.

Using phyphox

Why phyphox in this homework

phyphox gives practical smartphone sensor access and real-time streaming without requiring a full native iOS/Android app implementation. This design choice keeps the focus on algorithms while preserving realistic constraints:

- real sensor noise and orientation changes,
- sequential streaming updates,
- live visualization and stateful counting behavior.

App and sensor

As introduced above, phyphox is the app used for sensing and streaming in this homework. For accelerometer-based step counting, "Accelerometer without g" is recommended: `https://phyphox.org/wiki/index.php?title=Sensor:_Acceleration_(without_g)`. You may also collect gyroscope/magnetometer channels as optional auxiliary inputs.

CSV export

Record a walking session for at least 1 minute and export the measurement as a CSV file. The CSV typically contains columns like `time`, `ax`, `ay`, `az`. If you collect extra IMU channels, you may also include gyroscope and magnetometer columns. Column names may vary across devices.

Real-time access

phyphox provides a built-in web server via Remote Access: `https://phyphox.org/remote/`. Connect your phone and computer to the same Wi-Fi network, enable Remote Access, and open the URL shown by the app in a browser on your computer.

Please check the appendix for how to request the data from the server.

Showcase of phyphox

You should download the phyphox app from the App Store or Google Play Store. Example screenshots are shown below.

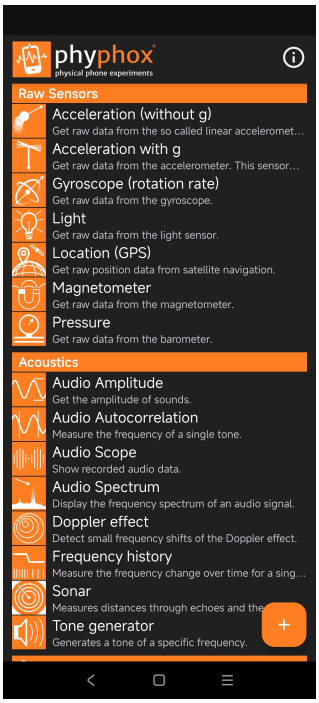

Figure 4: The main page of the phyphox app

Choose “Accelerometer without g” and click “Start” to begin recording.

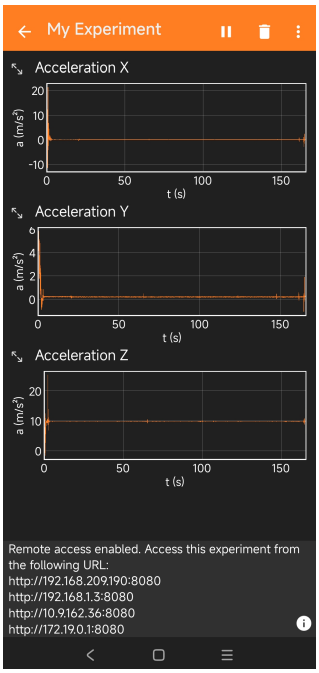

Figure 5: The “Accelerometer without g” page in phyphox

Enable “Remote Access” to view real-time data on your computer.

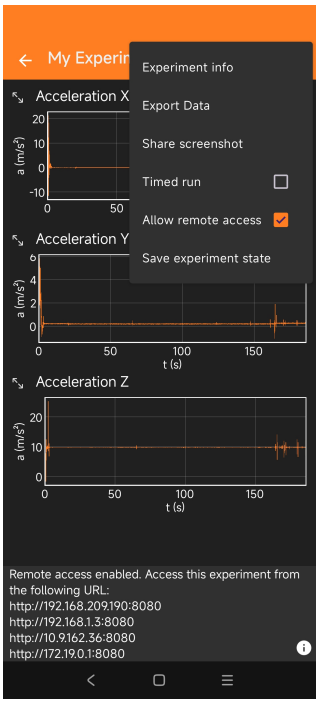

Figure 6: Remote Access settings in phyphox

Visit the address shown by the app in your browser to view real-time data.

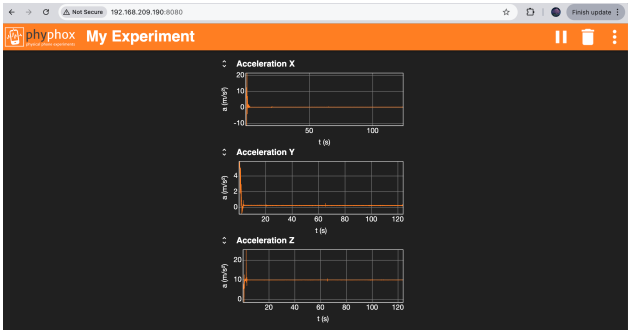

Figure 7: Real-time phyphox Remote Access page in a computer browser

Unified Framework for Algorithm Implementation

To ensure uniform grading and clean real-time demos, you must implement a single stateful class that supports both usage modes.

Required file

Create a file named `step_counter.py`.

Required class and strict output format

Your file must define the class below with the same method names and return keys. The grader will call `run_offline`.

```
import numpy as np


class StepCounter:
    """
    One step counter class for both offline and real-time usage.
    You can add any other attributes you need to the class. But you should not change the
         interface of the class.
    """

    def __init__(self):
        """
        Initialize the step counter.
        """
        raise NotImplementedError

    def reset(self) -> None:
        """
        Reset internal state such as buffers and cumulative count.
        After reset(), total_steps should be 0.
        """
        raise NotImplementedError

    def update(self, data_chunk: dict) -> dict:
        """
        Real-time update: process a chunk of new samples.

        Input
          data_chunk["time"] : numpy.ndarray with shape (M,) [required]
          data_chunk["acc"] : numpy.ndarray with shape (M, 3) in m/s^2 [required]
          data_chunk["gyro"] : numpy.ndarray with shape (M, 3) in rad/s [optional]
          data_chunk["mag"] : numpy.ndarray with shape (M, 3) in uT [optional]
          Chunks arrive sequentially.

        Output (must contain all keys)
          {
            "new_steps": int,
            "total_steps": int,
            "new_step_timestamps": np.ndarray, # shape (K,), float seconds
            "diagnostics": dict
          }
        """
        raise NotImplementedError

    def run_offline(self, data: dict) -> dict:
        """
        Offline processing: process a full recording.

        Input
```

```
        data["time"] : numpy.ndarray with shape (N,) [required]
        data["acc"] : numpy.ndarray with shape (N, 3) in m/s^2 [required]
        data["gyro"] : numpy.ndarray with shape (N, 3) in rad/s [optional]
        data["mag"] : numpy.ndarray with shape (N, 3) in uT [optional]

      Output format for grading (must contain all keys)
        {
          "step_count": int,
          "step_timestamps": np.ndarray, # shape (K,), float seconds
          "diagnostics": dict
        }

      Requirements on output:
        - "step_count" must be a Python int and must be >= 0.
        - "step_timestamps" must be a 1D NumPy array of dtype float with shape (K,).
          Each entry is a timestamp in seconds. If your algorithm does not produce
          timestamps, return an empty array with shape (0,) rather than None.
        - "diagnostics" must be a Python dict. It may be empty.
      """
      raise NotImplementedError
```

Strict requirements

- Do not change the class name or method names.
- Do not print inside any method.
- Do not read files inside any method.
- Do not modify the input dictionaries.
- Do not assume optional keys (`gyro`, `mag`) always exist.

Algorithm Requirements and Implementation Hints

Requirements

Your algorithm must satisfy the following requirements:

- Use 3-axis accelerometer data as the required main sensing input.
- You may use gyroscope/magnetometer as optional auxiliary signals.
- Work automatically on different recordings without per-file manual tuning.
- Be robust to phone orientation changes.
- Run correctly when only `time` + `acc` are provided.
- Output a non-negative integer step count in offline mode.
- Maintain correct cumulative counting in real-time mode.

Restrictions

- Do not use pretrained machine learning models.
- Do not use built-in step counter libraries or phone step APIs.
- Do not hard-code recording-specific parameters.

Standard numerical libraries such as NumPy, SciPy, and matplotlib are allowed.

Implementation hints

The items below are suggestions. You may use different approaches if they satisfy the requirements.

- Start with a peak-detection baseline on $a_{\text{mag}}(t)$.
- Use $a_{\text{mag}}(t)$ for orientation robustness.
- Apply smoothing, such as a moving average or low-pass filtering.
- In streaming mode, keep a short buffer so filtering and detection remain stable across chunk boundaries.

Streaming design tip

- Keep method-specific state (buffers, refractory timers, running thresholds, and model hidden state if any) inside the `StepCounter` object so offline and online behavior remain consistent.

Testing and Evaluation

A uniform dataset is used for standardized testing across students. It includes recordings from the OxWalk dataset from the University of Oxford:

`https://ora.ox.ac.uk/objects/uuid:19d3cb34-e2b3-4177-91b6-1bad0e0163e7`.

You do not need to worry about consistency between datasets. We will convert the standardized dataset into the same format as phyphox. Make sure your framework strictly follows the required format so that it can be used for grading.

You do not need to aim for the best performance on the standardized dataset. We mainly check whether your algorithm is robust and works for different recordings.

Standardized input/output protocol for fairness

For offline grading fairness, we will always call:

```
run_offline(data)
```

with at least:

- `data["time"]`: NumPy array of shape `(N,)`.

- `data["acc"]`: NumPy array of shape `(N, 3)` in m/s^2.

`gyro` and `mag` are not guaranteed in standardized grading input.

You may use additional IMU channels for your own system and demo, but your implementation must still produce valid results with accelerometer-only input.

Output keys and types for grading remain exactly the same as specified in the required class interface.

Visualization and Real-Time Demonstration

Your demo must use real-time data from phyphox (via Remote Access) and show:

1. A real-time data curve that updates continuously.
2. Real-time step counts, including immediate step detection and cumulative step count.

Report

Submit a PDF report that includes:

- A brief description of your data collection and how you recorded ground-truth step counts.
- A clear description of your algorithm and key parameters.
- Results on your own recordings with error metrics.
- Screenshots of your real-time visualization and an explanation of what is shown.
- If you explored multiple methods, a brief comparison of their trade-offs.
- A brief discussion of limitations and failure cases.

Grading

Total score is 100 points.

- 30% for standardized testing on the uniform dataset.
- 40% for performance on your own dataset and the visualization or demo.
- 30% for report writing quality and technical clarity.

Appendix

Accessing live data in Python using `requests`

When Remote Access is enabled, phyphox runs a small web server on the phone. All API endpoints are under the device URL shown by the app, for example: `http://192.168.0.42:8080` (typical on Android) or `http://192.168.0.42` (on iOS the server often runs on port 80, so the port can usually be

omitted). The API is documented at `https://phyphox.org/wiki/index.php/Remote-interface_communication`.

Key endpoint: `/get`. The endpoint `/get` returns a JSON object:

`{"buffer": {...}, "status": {...}}`.

You specify which buffers to fetch using query parameters:

- `/get?abc&def` returns the *last value* of buffers `abc` and `def`.
- `/get?abc=full` returns the *entire* buffer `abc` (may be inefficient).
- `/get?t=3` returns values of buffer `t` occurring after the first value exceeding `3`.

To efficiently stream only new samples, use a monotonic reference buffer (usually a time buffer) as a threshold and request other buffers aligned by that reference:

`/get?time=3&ax=3|time&ay=3|time&az=3|time`.

Here, the threshold `3` is applied to the reference buffer `time`, and the other buffers return values at the same indices as `time` values larger than `3`. Note: the character `|` must be URL-encoded as `%7C` if you construct URLs manually. Most HTTP libraries (including `requests`) handle this encoding for you.

How to find the buffer names

To request data via `/get`, you must use the correct *buffer names* (e.g., `time`, `ax`, `ay`, `az`, and optional gyroscope/magnetometer buffers). The exact names depend on the phyphox experiment and are described in the Remote Access interface.

The Remote API provides a configuration endpoint that exposes the experiment structure, including which buffers exist and their names. Open:

`http://<PHONE_IP>:<PORT>/config`

in your browser (or fetch it using Python). The returned JSON describes the experiment and includes buffer definitions and names.

Practical tip. If you are unsure, open `/config` in the browser and use the browser search function to search for the keyword `buffer`. The surrounding entries typically include the exact names you should use in `/get`.

Minimal polling example (recommended pattern). The code below shows a standard "fetch only new data" loop. You must replace `time`, `ax`, `ay`, `az` by the actual buffer names used by your phyphox experiment (these names are listed on the Remote Access page for that experiment). You may request additional buffers (for gyroscope/magnetometer) in the same way if you use them in your own demo.

```
import time
import requests
import numpy as np

BASE_URL = "http://192.168.0.42:8080" # replace with the URL shown in phyphox
TIME_BUF = "time" # replace if your experiment uses a different name
ACC_BUFS = ["ax", "ay", "az"] # replace with actual buffer names

last_time = None # last received time value (monotonic reference)

def fetch_new(last_t):
    # If last_t is None, fetch the last value only (cheap initialization)
    if last_t is None:
        params = {TIME_BUF: "", **{b: "" for b in ACC_BUFS}} # "/get?time&ax&ay&az"
    else:
        # Request only new data after last_t using time as reference.
        # For non-time buffers, use "threshold|referenceBuffer".
        params = {TIME_BUF: str(last_t)}
        for b in ACC_BUFS:
            params[b] = f"{last_t}|{TIME_BUF}"

    r = requests.get(BASE_URL + "/get", params=params, timeout=2.0)
    r.raise_for_status()
    return r.json()

while True:
    j = fetch_new(last_time)

    # Extract returned arrays. Each buffer entry contains metadata + the "buffer" array.
    tb = j["buffer"][TIME_BUF]["buffer"]
    axb = j["buffer"][ACC_BUFS[0]]["buffer"]
    ayb = j["buffer"][ACC_BUFS[1]]["buffer"]
    azb = j["buffer"][ACC_BUFS[2]]["buffer"]

    # Convert to numpy arrays (may be empty if no new data)
    t = np.asarray(tb, dtype=float)
    if t.size > 0:
        acc = np.stack([
            np.asarray(axb, dtype=float),
            np.asarray(ayb, dtype=float),
            np.asarray(azb, dtype=float),
        ], axis=1)

        last_time = float(t[-1])

        # TODO: feed into your streaming step counter:
        # out = step_counter.update({"time": t, "acc": acc})

    time.sleep(0.05) # adjust polling interval (e.g., 20--100 ms)
```

Status handling. The JSON response includes a `"status"` object with fields such as: `"session"` (unique identifier for the current session/experiment), `"measuring"` (running or paused), and timed-run information. If `"session"` changes, you should reset your buffers and internal state.

Other useful endpoints. The Remote API also supports:

- `/control` for start/stop/clear commands, e.g., `/control?cmd=start`.
- `/export` to download a full export file (CSV is one of the format indices).

See `https://phyphox.org/wiki/index.php/Remote-interface_communication` for details.