1. Project Overview

1.1 Purpose

This repository provides **open-source, simulation-based examples** of the algorithms in *Principles of Indoor Positioning and Indoor Navigation* (Chapters 2–8).

The repo is intended as a **companion to the book**: readers can read the equations and derivations in the text and then run concrete, minimal code examples that directly implement those equations on simulated datasets.

1.2 Target Users

- Master's students in navigation / robotics / geodesy / EE / CS.
- PhD students and early-career researchers entering indoor positioning.
- Engineers who want a **reference implementation** of classical algorithms before building production systems.

1.3 Goals

The repo should:

- Demonstrate key algorithms from Ch.2–8 on **small, reproducible simulations**:
    - **Ch.2:** coordinate systems & attitude.
    - **Ch.3:** LS/WLS, robust LS, KF/EKF/UKF/PF, FGO.
    - **Ch.4–7:** RF positioning, fingerprinting, PDR/sensors, SLAM.
    - **Ch.8:** practical sensor fusion.
- Provide **reproducible experiments** based on open simulation datasets.
- Offer **simple, inspectable reference implementations** rather than production systems.
- Support **per-chapter examples** that map back to the book.

New Goal: Equation-Level Traceability

For every important equation in the book that is implemented in code, there must be a **clear mapping**:

- **Book → Code**
    - For a given equation number (e.g. Eq. (3.12)), users can quickly locate:
        - The implementing function/class.
        - The tests and example notebooks that exercise it.
- **Code → Book**

    - From a function or class, users can see:
        - Which equation(s) it implements.
        - The chapter/section context.

Design constraints:

- Mapping must be:
    - **Searchable** via plain text (e.g. searching "Eq. (3.12)" in the repo).
    - **Maintainable** when new code is added.
    - Visible in **docstrings, comments, docs, and notebooks**.

1.4 Out of Scope

- Advanced techniques from Chapter 9 (crowdsourcing, collaborative, deep AI PDR, RIS) beyond small demos.
- Real-time deployment, mobile apps, or large-scale real-data pipelines.
- Full SLAM frameworks or high-performance mapping stacks.

---

2. Scope & Use Cases

2.1 Scope

The repo implements **simulation-first** reference versions of:

- Coordinate transforms (body/map/ENU/NED/LLH) and attitude representations.
- LS/WLS and robust LS; KF/EKF/UKF/PF; basic FGO wrappers.
- RF point positioning (TOA / two-way TOA / TDOA / AOA / RSS).
- Fingerprinting (Wi-Fi / magnetic / hybrid; deterministic & probabilistic).
- Proprioceptive sensor models (IMU, wheel odom, PDR, barometer, magnetometer).
- Minimal 2D SLAM & sensor fusion demos (Loosely vs Tightly coupled; observability).

2.2 Typical Use Cases

- “Run a notebook to reproduce a figure similar to chapter X.”
- “Swap KF ↔ FGO on the **same** simulated trajectory.”
- “Compare RF trilateration vs TDOA vs AOA in a toy floor.”
- “Compare k-NN fingerprinting vs a simple ML model on a synthetic RF map.”

This section sets **what problems the repo must solve**, not just what code exists.

---

3. High-Level Architecture

Repo layout:

```
ipin-examples/
  core/
    coords/
    estimators/
    rf/
    sensors/
    sim/
    eval/
  ch2_coords/
  ch3_estimators/
  ch4_rf_point_positioning/
  ch5_fingerprinting/
  ch6_dead_reckoning/
  ch7_slam/
  ch8_sensor_fusion/
  data/
    sim/
    real/    # optional, small demo logs only
  notebooks/
  docs/
  tools/
    check_equation_index.py
```

Directory roles:

- **core/**
  Reusable math & models used across chapters (never import from chX_.../):
    - coords/ – frames, LLH↔ECEF↔ENU, ENU↔NED, attitude conversions.

    - estimators/ – LS/WLS, robust LS, KF/EKF/UKF/PF, FGO wrappers.
    - rf/ – TOA/TDOA/AOA/RSS models, DOP utilities.
    - sensors/ – IMU, wheel odom, PDR, mag, barometer models.
    - sim/ – trajectory generators, scenario definitions, noise injection.
    - eval/ – error metrics, CDFs, NEES/NIS, DOP.
- **chX_.../**
  Thin chapter-specific examples:
    - Wiring code & scripts.
    - Plots and figure reproduction.
    - Minimal glue code between core/ and data.
- **data/sim/**
  Standardized **simulation datasets** (Section 5).
- **data/real/**
  Optional, very small demonstration logs (not the main focus).
- **notebooks/**
  One notebook (or small set) per chapter, referencing equations and pointing to core/ modules.
- **docs/**
  Markdown docs per chapter + equation_index.yml and usage docs.
- **tools/**
  CI/maintenance scripts, especially equation mapping checker.

---

4. Core Functional Requirements (Basic Functions)

For each core submodule, the design doc specifies:

- **Purpose**
- **Key data structures**
- **Required functions / classes** (with rough signatures)
- **Equation mapping** expectations (docstring conventions, index entries)
- **Owner**: navigation engineer vs software engineer.

4.1 core/coords

Purpose

- Implement coordinate and attitude foundations of Ch.2.

Basic functions

- Geodetic / ECEF / local frames:
    - llh_to_ecef(llh) -> np.ndarray
    - ecef_to_llh(ecef) -> np.ndarray
    - ecef_to_enu(ecef, ref_llh) -> np.ndarray
    - enu_to_ecef(enu, ref_llh) -> np.ndarray
    - enu_to_ned(enu) -> np.ndarray
    - ned_to_enu(ned) -> np.ndarray
- Attitude conversions:
    - rpy_to_rotmat(roll, pitch, yaw) -> np.ndarray
    - rotmat_to_rpy(R) -> Tuple[float, float, float]
    - quat_to_rotmat(q) -> np.ndarray
    - rotmat_to_quat(R) -> np.ndarray

Equation mapping

- All core transforms have docstrings like:
    - "Implements Eq. (2.x) in Chapter 2: ..."
- Each implemented equation appears in docs/equation_index.yml.

Unique tasks

- Support multiple map frames (multi-floor, building frames).
- Enforce consistent frame naming/metadata used by RF, PDR, SLAM.

A. Replacement text for Section 4.2 core/estimators

Use this to replace your current “4.2 core/estimators” subsection.

4.2 core/estimators

4.2.1 Purpose

This module implements the core estimation and filtering algorithms from Chapter 3 and exposes them through reusable APIs that other modules (core/rf, core/sensors, chX_...) can call.

Goals:

- **Provide reference implementations of:**

- **Least squares (LS), weighted LS (WLS), and robust LS.**
- **Linear KF, EKF, UKF, PF.**
- **Factor graph optimization (FGO) with basic numerical solvers.**

- **Maintain equation-level traceability to Chapter 3:**
  - **Each algorithm implementation references the exact equation numbers it follows.**
- **Reuse the same estimator APIs across RF positioning, PDR, SLAM, and fusion examples.**

4.2.2 Key abstractions & data structures

- **State vector**
  - **State = np.ndarray of shape (n,).**
- **Covariance matrix**
  - **Covariance = np.ndarray of shape (n, n).**
- **Measurement vector**
  - **Measurement = np.ndarray of shape (m,).**
- **ProcessModel (for KF/EKF/UKF/FGO)**
  - **Encapsulates system dynamics:**
    - **f(x, u, dt) -> x_next: nonlinear state propagation (Eq. (3.21) for EKF, Eq. (3.25) for UKF).**
    - **F(x, u, dt) -> Fk: Jacobian ∂f/∂x for EKF (used in Eq. (3.22)).**
    - **Q(x, u, dt) -> Qk: process noise covariance.**
- **MeasurementModel**
  - **Encapsulates sensor model:**
    - **h(x) -> z_pred: linear or nonlinear measurement (e.g. Eq. (3.8) for KF, Eq. (3.21) measurement part).**
    - **H(x) -> Hk: Jacobian ∂h/∂x for EKF.**
    - **R(x) -> Rk: measurement noise covariance (Σ_{w,z,k}).**
- **Factor / FactorGraph (for FGO)**
  - **Factor:**

        - **Holds residual function r(x_subset) and its Jacobian (or automatic differentiation stub).**
    - **FactorGraph:**
        - **Collection of variables and factors representing the MAP optimization problem of Eq. (3.35)–(3.38).**

All estimators share these abstractions so that changing the estimator does not require changing the sensor or motion models.

4.2.3 Least Squares (LS / WLS / Robust LS)

Relevant equations (Chapter 3)

- **Cost function:**
    - $J(x) = \Sigma (y_i - h_i(x))^2$ **→ Eq. (3.1).**
- **Linear model normal equations:**
    - $H^T H \hat{x} = H^T y$ **→ Eq. (3.2).**
    - $\hat{x} = (H^T H)^{-1} H^T y$ **→ Eq. (3.3).**
- **Nonlinear LS optimality condition:**
    - $\Sigma (y_i - h_i(x)) \nabla h_i(x) = 0$ **→ Eq. (3.4).**
- **Weighted LS cost (no explicit equation number, defined in the text in 3.1.1):**
    - $J(x) = \Sigma w_i (y_i - h_i(x))^2$.

Planned API (in core/estimators/least_squares.py)

- **def linear_least_squares(H: np.ndarray, y: np.ndarray) -> np.ndarray:**
    - **Implements Eq. (3.2)–(3.3).**
    - **Docstring must include:**
        - **“Implements Eqs. (3.2)–(3.3) (normal equations and closed-form LS solution).”**
- **def weighted_least_squares(H: np.ndarray, y: np.ndarray, W: np.ndarray) -> np.ndarray:**
    - **W is diagonal or full weight (inverse covariance).**
    - **Generalizes Eq. (3.2)–(3.3) to WLS.**
    - **Docstring: “Implements weighted least squares as defined in Section 3.1.1.”**

- **def gauss_newton_solve(h_fun, jac_fun, y, x0, max_iters, tol) -> np.ndarray:**
  - **Implements iterative solution of Eq. (3.4): residual gradient=0 via Gauss–Newton.**
  - **Used for nonlinear LS in RF / SLAM.**
- **def robust_least_squares(...) -> np.ndarray:**
  - **Wraps LS with robust loss functions (Huber, Cauchy, etc.) as described in Section 3.1.1.**
  - **Docstring references Table 3.1 + robust LS text, even if no explicit equation number.**

Equation mapping requirements

- **linear_least_squares docstring: references Eqs. (3.2)–(3.3).**
- **gauss_newton_solve: references Eq. (3.4).**
- **equation_index.yml must map:**
  - **"Eq. (3.2)" and "Eq. (3.3)" → core/estimators/least_squares.py::linear_least_squares.**
  - **"Eq. (3.4)" → core/estimators/least_squares.py::gauss_newton_solve.**

4.2.4 Kalman Filter Family (KF / EKF / UKF)

(a) Linear Kalman Filter

Relevant equations (Chapter 3)

- **Linear measurement model:**
  - **$z_k = H_k x_k + w_{z,k}$ → Eq. (3.8).**
- **Likelihood mean/covariance:**
  - **$E[z_k \mid x_k] = H_k x_k$, $Cov[z_k \mid x_k] = \Sigma_{w,z,k}$ → Eq. (3.9).**
- **Propagation model and prior mean/cov:**
  - **$x_{k,k-1} = F_k x_{k-1} + u_k + w_{x,k-1}$ → Eq. (3.11).**
  - **$P_{k,k-1} = F_k \Sigma_{x,k-1} F_k^T + \Sigma_{w,u,k}$ → Eq. (3.12).**
- **MAP derivation and posterior:**
  - **Posterior density (3.13), MAP via derivative (3.15), closed form (3.16).**
- **KF update form:**

- **$\hat{x}_{k,MAP} = x_{k,k-1} + K_k (z_k - H_k x_{k,k-1})$ → Eq. (3.17).**
- **$K_k = P_{k,k-1} H_k^T (H_k P_{k,k-1} H_k^T + \Sigma_{w,z,k})^{-1}$ → Eq. (3.18).**
- **Covariance: $\Sigma_{x,k} = P_{k,k-1} - F_k K_k H_k P_{k,k-1}$ → Eq. (3.19).**
- **Summary of "five equations" → grouped as Eq. (3.20).**

Planned API (in core/estimators/kalman.py)

- **class KalmanFilter:**
  - **__init__(self, F, Q, H, R) – constant matrices or callables.**
  - **predict(self, u: np.ndarray = None)**
    - **Implements propagation from Eqs. (3.11), (3.12) and the first two lines of (3.20).**
  - **update(self, z: np.ndarray)**
    - **Implements Eqs. (3.17)–(3.19) and full set in (3.20).**

Equation mapping requirements

- **predict docstring: "Implements the prediction step of the linear Kalman filter (Eqs. (3.11), (3.12), (3.20)).”**
- **update docstring: "Implements the update step (Eqs. (3.17)–(3.20)).”**
- **equation_index.yml:**
  - **"Eq. (3.11)", "Eq. (3.12)", "Eq. (3.17)", "Eq. (3.18)", "Eq. (3.19)", "Eq. (3.20)" → KalmanFilter.predict / KalmanFilter.update.**

(b) Extended Kalman Filter (EKF)

Relevant equations (Chapter 3)

- **Nonlinear state & measurement model:**
  - **$x_k = f(x_{k-1}, u_k) + w_k$, $z_k = h(x_k) + v_k$ → Eq. (3.21).**
- **EKF prediction:**
  - **$\hat{x}_k^- = f(\hat{x}_{k-1}, u_k)$,**
  - **$P_k^- = F_{k-1} P_{k-1} F_{k-1}^T + Q$ → Eq. (3.22).**
- **EKF update (on the next page, same section; equation number continues, but we primarily tie to Eq. (3.21)–(3.23)).**

Planned API

- **class ExtendedKalmanFilter(KalmanFilter):**
    - **Accepts ProcessModel and MeasurementModel objects instead of fixed matrices.**
    - **predict(self, u, dt):**
        - **Uses f, F, and Q → Eq. (3.21) and (3.22).**
    - **update(self, z):**
        - **Uses h, H, R and standard KF structure, referencing EKF update equations in the EKF subsection.**

Equation mapping

- **predict docstring: "Implements EKF prediction (Eqs. (3.21)–(3.22)).”**
- **update docstring: "Implements EKF update (EKF section following Eq. (3.21)).”**
- **equation_index.yml: map "Eq. (3.21)", "Eq. (3.22)" to ExtendedKalmanFilter.predict.**

(c) Unscented Kalman Filter (UKF)

Relevant equations (Chapter 3)

- **Sigma-point generation:**
    - **$\chi_0 = \hat{x}_{k-1}$, $\chi_i = \hat{x}_{k-1} + \delta_i$, $\chi_{i+n} = \hat{x}_{k-1} - \delta_i$, i=1..n → Eq. (3.24).**
- **Sigma-point propagation through f (process) and h (measurement):**
    - **$\chi_i^- = f(\chi_i, u_k)$ → Eq. (3.25).**
- **UKF update and gain:**
    - **K_k, $\hat{x}_k$, P_k defined via cross-covariances and measurement covariances → Eq. (3.30).**

Planned API

- **class UnscentedKalmanFilter:**
    - **__init__(self, process_model, measurement_model, alpha, beta, kappa)**
    - **predict(self, u, dt)**
        - **Generates sigma points (Eq. (3.24)), propagates via f (Eq. (3.25)), computes predicted mean/cov.**
    - **update(self, z)**

        - **Computes predicted measurement, cross-covariance, Kalman gain, and posterior state following Eq. (3.30).**

Equation mapping

- **Docstrings reference Eqs. (3.24), (3.25), (3.30) explicitly.**
- **equation_index.yml maps these equations to the UKF methods.**

4.2.5 Particle Filter (PF)

Relevant equations (Chapter 3)

- **Recursive Bayes update:**
    - **$p(x_k \mid z_{1:k}) \propto p(z_k \mid x_k)\, p(x_k \mid z_{1:k-1})$ → Eq. (3.32).**
- **Sampling step:**
    - **$x_k^{(i)} \sim p(x_k \mid x_{k-1}^{(i)})$ → Eq. (3.33).**
- **Weight update:**
    - **$\dot{w}_k^{(i)} = w_{k-1}^{(i)}\, p(z_k \mid x_k^{(i)})$ → Eq. (3.34).**

Planned API (in core/estimators/particle.py)

- **class ParticleFilter:**
    - **Attributes: particles (N×n), weights (N,), process_model, measurement_model.**
    - **predict(self, u, dt):**
        - **Sample new particles using the process model (Eq. (3.33)).**
    - **update(self, z):**
        - **Update weights via likelihood (Eq. (3.34)), normalize, resample.**
    - **estimate(self) -> np.ndarray:**
        - **Return weighted mean or best-weight particle.**

Equation mapping

- **predict docstring references Eq. (3.33).**
- **update docstring references Eq. (3.32)–(3.34).**
- **equation_index.yml: "Eq. (3.32)", "Eq. (3.33)", "Eq. (3.34)" → ParticleFilter.**

4.2.6 Factor Graph Optimization (FGO) & Numerical Methods

Relevant equations (Chapter 3)

- **MAP for full trajectory:**
  - **$\hat{X}_{MAP} = \text{argmax}_X\ p(X \mid Z) = \text{argmax}_X\ p(Z \mid X)\ p(X) / p(Z)$ → Eq. (3.35).**
- **Simplified MAP form:**
  - **$\hat{X}_{MAP} = \text{argmax}_X\ \ell(X; Z)\ p(X)$ with $\ell(X; Z) \propto p(Z \mid X)$ → Eqs. (3.36)–(3.37).**
- **Conversion from product of Gaussians to sum of squared residuals (negative log-posterior) → Eq. (3.38) and following.**
- **Gradient descent update:**
  - **$x_{k+1} = x_k + \alpha\ d$ → Eq. (3.42).**
- **Descent condition:**
  - **$f(x_{k+1}) = f(x_k + \alpha\ d) < f(x_k)$ → Eq. (3.43).**

Planned API (in core/estimators/fgo.py and core/estimators/optim.py)

- **class Factor:**
  - **def residual(self, x: np.ndarray) -> np.ndarray:**
  - **def jacobian(self, x: np.ndarray) -> np.ndarray:**
- **class FactorGraph:**
  - **Stores variable ordering, list of factors.**
  - **def evaluate(self, x: np.ndarray) -> Tuple[r, J]: returns stacked residuals and Jacobian.**
- **def gradient_descent_step(x, grad, alpha) -> np.ndarray:**
  - **Encodes Eq. (3.42) and Eq. (3.43) logic.**
- **def gauss_newton_step(x, J, r) -> np.ndarray:**
  - **Equivalent to LS step on residuals derived from Eq. (3.38).**
- **def levenberg_marquardt_step(...) -> np.ndarray:**
  - **LM step bridging gradient descent and Gauss–Newton, consistent with 3.4.1 discussion.**
- **def solve_fgo(graph: FactorGraph, x0, method='gn', max_iters=..., tol=...) -> np.ndarray:**
  - **High-level solver that iteratively applies GN/LM using the factor graph.**

Equation mapping

- **solve_fgo docstring references Eqs. (3.35)–(3.38) (MAP via factor graph).**
- **gradient_descent_step docstring references Eqs. (3.42)–(3.43).**
- **equation_index.yml ties these equations to the corresponding functions.**

4.2.7 Simulation data requirements (for estimators)

Although most estimators are exercised through chapter-specific scenarios, core/estimators also needs simple synthetic datasets for unit tests and minimal examples:

- **toy_ls_linear/**
  - **Small linear regression / linear system:**
    - **H (m×n), y (m), known true x_true, noise variance.**
  - **Used to test:**
    - **linear_least_squares, weighted_least_squares, robust_least_squares.**
  - **Must be documented in Section 5.2 as one of the dataset families.**
- **toy_kf_1d_cv/**
  - **1D constant-velocity (CV) model:**
    - **State: [position, velocity].**
    - **Measurements: noisy position only.**
  - **Used to test:**
    - **KalmanFilter convergence vs. analytical solution.**
- **toy_nonlinear_1d/**
  - **1D or 2D nonlinear tracking:**
    - **Nonlinear measurement, e.g. range², to exercise EKF, UKF, PF, and FGO.**
  - **Used to:**
    - **Compare KF vs. EKF vs. UKF vs. PF on the same synthetic data.**
    - **Test FGO smoothing vs. recursive filters.**

These datasets should be stored under data/sim/ (see Section 5.2) and used both by:

- **Unit tests in tests/test_estimators_*.py.**
- **A small ch3_estimators notebook that reproduces selected plots / behaviors from Chapter 3.**

4.3 core/rf

Purpose

- RF signal measurement models from Ch.4–5.

Basic functions

- Ranging / angles:
    - toa_range(tx_pos, rx_pos, c, clock_bias)
    - two_way_toa_range(...)
    - tdoa_range_diff(anchor_i, anchor_j, state, c)
    - aoa_bearing(anchor_pos, state)
- RSS:
    - rss_pathloss(tx_power, distance, n, sigma_shadow)
- DOP:
    - compute_dop(geometry, weights=None) -> dict with HDOP/VDOP/PDOP.

Equation mapping

- Each measurement model documents which equations in Ch.4 it implements (e.g. path-loss, hyperbolic TDOA, AOA geometry).

Unique tasks

- Support multiple technologies (Wi-Fi, BLE, UWB, 5G) through parameterization rather than separate code.

4.4 core/sensors

Purpose

- Proprioceptive & environmental sensor models from Ch.6.

Basic functions

- IMU:
    - Continuous/discrete error models (bias, noise, RW).
    - Simple strapdown integration for 2D/3D.
- PDR:

    - Step detection, step length models, heading from IMU/mag.
- Environmental:
    - Barometer altitude, floor detection.
    - Magnetometer heading and magnetic fingerprint stubs.
- Wheel odometry:
    - Differential drive, steering model, noise models.

Equation mapping

- IMU propagation, error models, ZUPT updates etc. point to Ch.6 equations.

Unique tasks

- Define a **unified sensor packet** structure used across simulation and logs.

4.5 core/sim

Purpose

- Shared simulation tools for all examples.

Basic functions

- Trajectories:
    - generate_2d_trajectory(shape, params, dt, duration)
    - generate_3d_trajectory(...)
- Scenarios:
    - Rooms/floor plans, beacons/anchors, obstacles.
- Sensor noise injection:
    - add_imu_noise(traj, model_params)
    - simulate_rf_measurements(traj, anchors, rf_config)

Unique tasks

- Scenario configurations in JSON/YAML so all chapters share the same definitions.

- **4.6 core/eval**
- **Purpose**
  • Evaluation and visualization utilities shared across chapters.
  • Provide a small, consistent API for:
  – computing errors and consistency metrics, and
  – generating publication-quality, vector graphics for trajectories, errors, and algorithm comparisons.

- **Basic functions (numerical evaluation)**
  • Position errors:
  – compute_position_errors(truth, est) -> dict
  ▫ Per-epoch error vectors (ENU / NED).
  – compute_rmse(errors) -> float | dict
  ▫ Scalar RMSE or per-axis RMSE.
  – compute_error_stats(errors) -> dict
  ▫ mean / median / percentiles.
  • Consistency metrics:
  – compute_nees(truth, est, cov) -> np.ndarray
  – compute_nis(innov, S) -> np.ndarray
  – helpers to check NEES/NIS against χ^2 bounds.
  • RF / geometry utilities:
  – compute_dop(geometry, weights=None) -> dict (HDOP/VDOP/PDOP), reusing core/coords for frame handling.
- 574882d9-b20d-4e2c-a4e3-c94a8d9…
- **Basic functions (visualization)**
  All plotting helpers live in core/eval/plots.py. They must:
  • accept plain NumPy arrays / dicts (no hidden global state), and
  • return a matplotlib.figure.Figure (the caller decides whether to show or save).
- Planned APIs:
  • Trajectory / map views
  – plot_trajectory_2d(truth_xy, est_xy_dict, anchors_xy=None) -> Figure
  ▫ truth vs. one or more estimates on a 2D floor plan.
  ▫ optional RF anchors / landmarks overlay.
  – plot_trajectory_3d(truth_xyz, est_xyz_dict, anchors_xyz=None) -> Figure
  ▫ simple 3D view for SLAM / multi-floor examples.
  • Error over time
  – plot_position_error_time(errors_dict, dt, axes="enu") -> Figure
  ▫ per-axis error vs. time, multiple algorithms on one plot.
  • Error distributions
  – plot_error_hist(errors_dict, bins=…) -> Figure
  – plot_error_cdf(errors_dict) -> Figure
  ▫ used heavily in Ch.4–5 to compare RF / fingerprinting methods.
- 574882d9-b20d-4e2c-a4e3-c94a8d9…
-
  • Consistency plots
  – plot_nees(nees, chi2_bounds) -> Figure
  – plot_nis(nis, chi2_bounds) -> Figure
  • RF geometry / DOP
  – plot_rf_geometry(anchors_xy, traj_xy=None) -> Figure
  – plot_dop_map(dop_grid, floorplan=None) -> Figure
  ▫ visualize how beacon layouts (or AP layouts) affect HDOP / VDOP.
  • SLAM / map views (minimal)
  – plot_occupancy_grid(grid, poses=None, landmarks=None) -> Figure
  – plot_factor_graph_skeleton(nodes, edges) -> Figure (optional, diagnostic only).
- **Standard vector export**
- To make results easy to reuse in papers and slides, every plotting helper must work with a single, shared export utility:

- • save_figure(fig, out_dir, name, formats=("svg", "pdf")) -> list[path]
  – ensures:
  ▫ vector formats by default (.svg, .pdf),
  ▫ deterministic naming, and
  ▫ creation of the output directory if needed.
- **File and naming conventions**
- • All chapter examples should save figures into:
  – chX_.../figs/ for scripts, or
  – notebooks/figs/chX_.../ for notebooks.
  • File names must follow:
  – ch3_estimators_kf_vs_pf_rmse.svg
  – ch4_rf_toa_vs_tdoa_cdf.pdf
  – ch6_pdr_drift_correction_trajectory.svg
  • Notebooks *embed* the figure inline, but also call save_figure so users get reusable vector files with a single run.
- **Visual style guidelines**
- To keep plots visually consistent across the repo:
  • Always label axes with units, e.g. Position error [m], Time [s], Heading [deg].
  • Use the same color / linestyle mapping for estimator families across chapters:
  – LS/WLS: solid blue
  – KF/EKF/UKF: solid green
  – PF: dashed orange
  – FGO / smoothing: solid red
  • Keep backgrounds white and gridlines light (MATLAB/Matplotlib default is fine).
  • Avoid heavy custom styling in core; chapter examples can adjust if needed.
- **Unique tasks**
- • Define a small set of "standard plots" per chapter (see Section 7) and implement them using core/eval/plots.py.
  • Make sure every example script / notebook:
  – prints a small numerical summary (RMSE, final bias, etc.), and
  – generates at least one vector figure via save_figure(...).
  • Ensure that evaluation APIs are reused across RF, PDR, SLAM, and fusion examples so that algorithm comparisons are meaningful and visually comparable.

---

5. Simulation Datasets & Open Data

5.1 Principles

- All primary examples must run **only on data in data/sim/**.
- Datasets must be:
  - Small enough to clone and run quickly.
  - Reproducible (fixed seeds, recorded configs).
  - Clearly licensed (e.g. CC-BY-4.0) and documented in docs/data.md.

5.2 Dataset Families

Planned families under data/sim/:

- rf_2d_floor/
    - 2D floor map, anchor positions, true trajectories, TOA/TDOA/AOA/RSS measurements.
- wifi_fingerprint_grid/
    - Reference points on a grid, RSS vectors, test trajectories.
- pdr_corridor_walk/
    - IMU time series, reference path, step labels, floor labels.
- slam_lidar2d/
    - 2D occupancy grid, range-bearing measurements, ground truth poses.

 toy_ls_linear/

- Simple linear system for least-squares testing:
    - Files: H.npy, y.npy, x_true.npy, noise_config.json.
- Used by: core/estimators/least_squares.py unit tests and ch3_estimators examples.

 toy_kf_1d_cv/

- 1D constant-velocity system:
    - Files: x_true.npy, z.npy, F.npy, H.npy, Q.npy, R.npy, dt.
- Used to validate KalmanFilter implementation and tuning.

 toy_nonlinear_1d/

- Nonlinear state/measurement example:
    - State trajectory file, nonlinear measurement config, noise statistics.
- Used by: EKF, UKF, PF, and FGO examples to illustrate behavior under non-Gaussian/nonlinear conditions.

For each dataset:

- File formats: CSV / NPZ / HDF5.
- Coordinate frames: clearly defined, link to core/coords.
- Ground truth semantics.

5.3 Optional Real Data

- data/real/ may contain **small** demo logs (not full research datasets) to show real-world quirks.
- Not a core dependency for running per-chapter examples.

---

6. Equation–to–Code Mapping Design

This section defines how equation mapping is implemented, enforced, and integrated with core/ and chapter modules.

6.1 Canonical Equation ID Format

- Equations are referenced using a **canonical string**:
    - Eq. (C.NN) for Chapter C, Equation NN.
    - If the book uses section-based numbering, adapt to Eq. (C.SS).

Examples

- Chapter 2, equation 5 → "Eq. (2.5)"
- Chapter 3, equation 12 → "Eq. (3.12)"

Rules

- This exact string must appear in:
    - Docstrings.
    - Comments (for line-level mapping).
    - Notebooks (markdown cells).
    - docs/equation_index.yml.

That way, a simple repo search for "Eq. (3.12)" finds all relevant code/tests.

6.2 Code Conventions (Docstrings & Comments)

Requirement

Every function/class that directly implements a book equation must mention it in its **docstring**.

Example: LS

```
def linear_least_squares(A: np.ndarray, b: np.ndarray) -> np.ndarray:
    """
    Solve the linear least squares problem x* = argmin ||Ax - b||_2.
```

Implements Eq. (3.12) in Chapter 3 of the book:

x* = (A^T A)^{-1} A^T b

"""

...

Example: EKF predict

def predict(self, u: np.ndarray) -> None:

"""

Extended Kalman filter prediction step.

Implements Eqs. (3.25)–(3.27) in Chapter 3:

- State propagation: Eq. (3.25)

- Jacobian: Eq. (3.26)

- Covariance propagation: Eq. (3.27)

"""

...

Line-level comments

If a specific line encodes an equation:

Eq. (3.27): P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k

P_pred = F @ P @ F.T + Q

Design rule

- Whole function = one equation (or block) → docstring reference.
- Single expression or key step = equation → inline comment with the exact Eq. (C.NN) reference.

6.3 Central Equation Index File

Create docs/equation_index.yml (or .json).

Purpose

From the book side, you can look up "Eq. (3.12)" and see:

- Which modules/objects implement it.
- Which tests/notebooks exercise it.

YAML structure (example)

```
- eq: "Eq. (2.3)"
  chapter: 2
  description: "ECEF to ENU coordinate transformation"
  files:
    - path: "core/coords/frames.py"
      object: "ecef_to_enu"
    - path: "ch2_coords/examples/coord_demo.py"
      object: "demo_ecef_to_enu"
  tests:
    - "tests/test_coords.py::test_ecef_enu_roundtrip"


- eq: "Eq. (3.12)"
  chapter: 3
  description: "Linear least squares closed form"
  files:
    - path: "core/estimators/least_squares.py"
      object: "linear_least_squares"
  notebooks:
    - "notebooks/ch3_estimators/ls_vs_ekf.ipynb"
```

Design choices

- One entry per implemented equation.
- Keys:
    - eq – canonical ID ("Eq. (3.12)").
    - chapter
    - description
    - files – list of {path, object}.
    - Optional: tests, notebooks, notes.

6.4 Notebooks & Docs Conventions

Notebooks

At the top of each chapter notebook, add an “Equations used” list, e.g.:

Equations in this notebook

- Eq. (3.12): Linear least squares solution.
- Eqs. (3.25)–(3.27): EKF prediction equations.

And reference equations in text:

We now implement the linear least squares estimator from Eq. (3.12).

Docs

In docs/ch3_estimators.md:

Equation map

- Eq. (3.12): `core/estimators/least_squares.py::linear_least_squares`
- Eqs. (3.25)–(3.27): `core/estimators/kalman.py::ExtendedKalmanFilter.predict`

This gives users three navigation routes:

1. Search "Eq. (3.12)" in the repo.
2. Check docs/equation_index.yml.
3. Read chapter docs listing mappings.

6.5 Tooling: Equation Index Checker Script

Add tools/check_equation_index.py:

Responsibilities:

- Parse docs/equation_index.yml.
- For every {path, object} entry:
    - Verify the module and object exist.
    - Verify the docstring or comments contain the mapped "Eq. (C.NN)".

Sketch (already provided):

- Use importlib to load modules.
- Inspect obj.__doc__.
- Fail CI if any mismatch.

Integrate in CI:

- Run pytest.
- Run python tools/check_equation_index.py.

6.6 Integration into Epics / Phases

For each epic, “definition of done” includes:

- Code & tests written.
- Example notebook added.
- **Equation mapping completed**:
    - Docstring references added.
    - equation_index.yml updated.
    - Notebook/docs list relevant equations.

Concretely:

- **Epic 1 – Coordinates (Ch.2)**
    - All transforms tagged with Eq. (2.x) where applicable.
- **Epic 2 – Estimators (Ch.3)**
    - LS, KF, EKF, UKF, PF, FGO all mapped to Ch.3 equations.
- **Epic 3 – RF positioning (Ch.4)**
    - TOA/TDOA/AOA/RSS and DOP models mapped.
- Subsequent epics (fingerprinting, PDR, SLAM, fusion) follow the same pattern.

---

7. Per-Chapter Example Modules (Unique Tasks)

For each chX_... folder, list:

- Example scripts / notebooks.
- Which core/ functions they exercise.
- Which equations they highlight.

Standard visualization outputs (all chapters)
For each chX_... module, at least one example must:
• call core/eval metrics to compute errors / statistics, and
• call core/eval/plots.py to generate:
– a trajectory or map view, and
– at least one error/time or error/CDF plot.

All examples should save figures via save_figure(...) into chX_.../figs/ (scripts) or notebooks/figs/chX_.../ (notebooks) so that:
• readers can quickly compare different algorithms or datasets by opening the SVG/PDF files, and
• instructors can directly reuse the vector figures in slides and course materials without manual re-plotting.

7.1 ch2_coords/

Examples

- demo_frames.py
    - Build body/map/ENU/NED frames and print transforms.
- Simple plotting script to reproduce or echo Ch.2 figures.

Dependencies

- core.coords, core.sim, core.eval.

7.2 ch3_estimators/

Examples

- ls_vs_robust_ls.py
    - Outlier demo: LS vs Huber vs Cauchy.
- kf_vs_ekf_vs_ukf_vs_pf_vs_fgo.ipynb
    - 1D/2D toy system comparing estimator families.

Unique tasks

- Compare convergence, RMSE, computation time.
- Show qualitatively how different estimators behave under same conditions.

7.3 ch4_rf_point_positioning/

Examples

- TOA trilateration (LS/WLS).
- TDOA Chan & Fang algorithms.
- AOA OVE / PLE examples.

Dependencies

- core.coords, core.rf, core.estimators, core.eval.

7.4 ch5_fingerprinting/

Examples

- Synthetic grid map:
    - Generate RSS fingerprints from core.rf and core.sim.
- Deterministic methods:
    - k-NN, weighted k-NN.
- Probabilistic:
    - Naive Bayes / simple Gaussian likelihood.
- Optional:
    - Small MLP (e.g. 2–3 layers) as a “modern” baseline.

7.5 ch6_dead_reckoning/

Examples

- Foot-mounted PDR with ZUPT:
    - Walk along corridor, accumulate drift, correct with constraints.
- Vehicle odom + IMU:
    - Simple car-like motion on a map.

7.6 ch7_slam/

Examples

- Minimal 2D LiDAR / range-bearing SLAM:
    - Use FGO with pose + landmark factors.
- Loop closure toy example.

7.7 ch8_sensor_fusion/

Examples

- Loosely vs tightly coupled fusion examples:
    - GNSS+IMU style but adapted to indoor sensors.
- Observability demo:
    - Show drift/unobservability vs properly fused sensors.
- Calibration:
    - Simple intrinsic/extrinsic calibration toy.

For each chapter section, the design doc should separate:

- **Core functions reused** from core/.

- **Chapter-specific "unique tasks"** and plots.

---

8. Non-Functional Requirements

8.1 Language & Tooling

- Python 3.x.
- NumPy, SciPy, matplotlib, optional JAX/PyTorch for ML examples (kept light).
- Packaging via pyproject.toml / poetry or similar.

8.2 Quality

- Unit tests for all core functions (tests/).
- pytest-based CI.
- Equation index checker included in CI.

8.3 Performance

- All examples should run on a typical laptop in **minutes**, not hours.
- Datasets and notebooks: no GPU dependency required.

8.4 Documentation

- docs/:
    - Per-chapter docs (overview, how to run examples).
    - docs/equation_index.yml and usage instructions.
- README.md:
    - Explain structure, target audience, and how to map book ↔ repo.

---

9. Implementation Roadmap

Use epics as a roadmap for you + contributing engineers.

9.1 Epic 0 – Repo & Infrastructure

- Initialize repo structure.
- Set up:
    - core/, chX_.../, data/, notebooks/, docs/, tools/.
    - CI with pytest + equation checker.

9.2 Epic 1 – Coordinates (Ch.2)

- Implement core/coords.
- Add tests for:
    - LLH↔ECEF↔ENU round-trip.
    - ENU↔NED conversions.
- Implement ch2_coords examples.
- Add Ch.2 equations to equation_index.yml.

9.3 Epic 2 – Estimators (Ch.3)

- Implement LS/WLS + robust LS in core/estimators.
- Implement KF/EKF/UKF/PF skeletons.
- Add simple FGO wrapper.
- Unit tests (simple linear systems).
- ch3_estimators example notebook.
- Update equation_index.yml for Ch.3.

9.4 Epic 3 – RF Positioning (Ch.4)

- Implement measurement models and DOP in core/rf.
- Build RF simulation tools in core/sim.
- ch4_rf_point_positioning examples:
    - TOA, TDOA (Chan/Fang), AOA demos.
- Add Ch.4 equations to equation_index.yml.

9.5 Later Epics – Fingerprinting, Sensors, SLAM, Fusion

- Similar pattern:
    - Spec → core implementation → tests → examples → docs → equation mapping.

---

10. Working Effectively: SW vs Navigation Engineers

Throughout the design doc, separate:

- **Algorithm / modeling decisions** (navigation engineer):
    - Which equation(s) in the book to implement.
    - What approximations and parameters to use.

- **Implementation details** (software engineer):
    - File layout, function signatures, performance, tests, CI.

The equation mapping system bridges the two:

- Nav engineers can **check correctness** by tracing Eq. → code.
- SW engineers can **preserve traceability** by following docstring + index conventions and the equation checker.