ChatGPT

PX4 SITL EKF2 Guide (Simulated GPS-Only)

This guide provides a step-by-step walkthrough to run a PX4 Software-In-The-Loop (SITL) simulation with Gazebo, MAVROS2 (ROS 2 Jazzy), and a Python EKF analytics/mission script. The goal is to ensure **EKF2** uses **only the simulated GPS** (no real GPS input) and that the estimator's local and global position outputs are valid. We include explicit PX4 shell (`pxh>`) commands for configuring the system, as well as verification steps at each stage. Finally, an appendix explains how the provided Python script waits for EKF health via ROS2 topics.

Terminal 1: Launch PX4 SITL (Gazebo X500) and Configure EKF2

1. **Start PX4 SITL with Gazebo:** Open a terminal and launch the PX4 SITL simulation using the X500 drone model in Gazebo. For example:

```
cd ~/PX4-Autopilot
make px4_sitl gz_x500
```

This command builds/starts PX4 SITL and Gazebo. Wait for Gazebo to load and for the PX4 shell prompt (`pxh>`) to appear in the terminal.

1. **Enable the simulated GPS (fake GPS) module:** At the `pxh>` prompt, ensure PX4 is using the simulated GPS data (and not waiting for any real GPS). Set the following parameters in the PX4 shell:

```
pxh> param set SENS_EN_GPSSIM 1
pxh> param set SENS_EN_MAGSIM 1
```

The X500 airframe config typically enables these by default, but this double-checks that the **GPS simulator** and **magnetometer simulator** are active [1]. With `SENS_EN_GPSSIM=1`, PX4 will simulate GPS readings by acquiring the vehicle's true pose from Gazebo and feeding it through the `sensor_gps_sim` module [2]. This ensures the **fake GPS** is the **only** GPS source used by PX4. *(The magnetometer is similarly simulated, while the barometer comes from a Gazebo plugin in this model.)*

> **Note:** By default, the EKF2 uses the barometric altitude as its height source (parameter `EKF2_HGT_MODE = 0`, barometer). This is ideal in simulation since a barometer reading is available. Using a consistent height source (barometer vs. GPS altitude) avoids jumps in the origin altitude. Ensure you haven't changed this parameter, so the EKF's altitude reference remains stable.

1. **(Optional) Manually seed EKF origin:** PX4 normally sets the home position/EKF origin automatically once the GPS acquires a fix. If you want to explicitly set the global origin (for example, to ensure consistency across runs or multiple vehicles), you can use:

```
pxh> commander set_ekf_origin 47.397742 8.545594 488
```

This command seeds the EKF2 origin to latitude 47.397742 °N, longitude 8.545594 °E, altitude ~488 m [3] (which is the default Zurich Irchel Park location used in PX4's SITL simulation). In most cases, this step isn't required for a single vehicle, but it guarantees the EKF's global reference matches the known simulation location.

- If you change origin or certain EKF parameters at runtime, you may **restart the EKF2** to ensure a clean initialization. To do so, enter:

  ```
  pxh> ekf2 stop
  pxh> ekf2 start
  ```

  This will reset the estimator state using the new settings/origin. (Give the system a few seconds after restart to regain a GPS fix and initialize.)

- **Verify only simulated GPS is in use:** Use the PX4 uORB listener to inspect GPS data:

```
pxh> listener sensor_gps
```

This will display the latest GPS message. You should see **one** instance of `sensor_gps` data, corresponding to the fake GPS. In the output, look for:
- `fix_type` – should be **3** or higher (3 = 3D fix).
- **Latitude/Longitude** – should be near **47.3977° N** and **8.5456° E** (the default SITL coordinates) [3].
- **# of satellites** – a non-zero count (e.g., 10+ sats, since the simulation typically provides a perfect or high-quality fix).

For example, the listener might show:

```
TOPIC: sensor_gps ...  (1 instance)
 sensor_gps_s
    timestamp: 123456789  (.... seconds ago)
    latitude_deg: 47.397742
    longitude_deg: 8.545594
    altitude_msl_m: 488.0
    fix_type: 3
    satellites_used: 10
    ... (other fields) ...
```

This confirms the simulated GPS is providing a valid 3D fix at the expected location. **No second GPS** should be listed. (If you saw multiple instances, e.g., `sensor_gps 2 instances`, that would indicate an unwanted second source [4] – in our case we expect only one.) The latitude ~47.3977 and longitude

~8.5456 match the known SITL home position [3], confirming that **no real GPS data is being used**, only the Gazebo/PX4 fake GPS feed.

1. **Check EKF2 status for position validity:** Now verify that the estimator has initialized both local and global position estimates. Run:

```
pxh> ekf2 status
```

PX4 will report the status of the EKF2 estimator. In the output, find lines for **"local position"** and **"global position."** Both should be marked **"valid"** (or show as `1`). For example:

```
INFO  [ekf2] local position: valid
INFO  [ekf2] global position: valid
```

This indicates that EKF2 is using the simulated sensors and has a healthy position estimate in both the local NED frame and global (latitude/longitude) frame. In numeric form, PX4 may show `local position: 1, global position: 1` (1 meaning true/valid) [5]. **Do not proceed** unless both are valid (if you see "invalid" or `0`, the EKF is not yet ready). If they are `invalid`, ensure the GPS fix is good (see step 4) and give the system a few more seconds; the EKF might still be initializing.

1. **Inspect the local position and origin (optional):** You can further examine the EKF's internal state:

```
pxh> listener vehicle_local_position
```

This shows the local position estimate message. Key things to check:

- **Ref**: Look at `ref_lat`, `ref_lon`, `ref_alt`. These are the latitude/longitude/altitude that the EKF is using as the origin for the local frame. They should correspond to the home position (around 47.3977, 8.5456 in our case) and altitude ~488 m [6].
- **Position**: The `x, y, z` fields are the NED coordinates of the vehicle relative to the origin. Right after startup (before takeoff), `x` and `y` should be near 0 (the vehicle hasn't moved from the origin). The `z` will be small (close to 0) or slowly changing if there's slight barometer drift; note `z` is NED, so positive *down*.
- **Validity flags**: Check that `xy_valid` and `z_valid` are `True`, and similarly `v_xy_valid` / `v_z_valid` (velocity validity) are `True` [7]. Also, `xy_global` and `z_global` should be true, indicating the filter has aligned local frame to global reference.

For example, an output might look like:

```
ref_lat: 47.397742,  ref_lon: 8.545594,  ref_alt: 488.0
x: 0.021,  y: -0.015,  z: -0.002   (vehicle is roughly at origin)
vx: 0.0, vy: 0.0, vz: 0.0         (vehicle is stationary)
```

```
xy_valid: True, z_valid: True, v_xy_valid: True, v_z_valid: True
xy_global: True, z_global: True, dead_reckoning: False
```

This detailed check confirms that the EKF's local frame is properly set (no large offsets) and the position data is valid and globally referenced.

<small>(*If any validity flags are false or the ref coordinates look wrong, there may be an issue with origin setting or sensor data. In a properly configured SITL with simulated GPS, these should all look correct.*)</small>

With Terminal 1 setup complete, the PX4 SITL is running with **EKF2** fully initialized (local_position and global_position are valid) using **only the simulated sensors** (GPS, mag, baro). We can now bridge to ROS2 to observe these states and run missions.

Terminal 2: Launch MAVROS (ROS 2 Bridge) and Verify Data

Open a second terminal for ROS 2 and MAVROS. This will connect to the SITL via MAVLink and expose data/topics we can use to monitor the EKF and control the vehicle.

1. **Start MAVROS and connect to PX4:** Source your ROS 2 environment (e.g., `source /opt/ros/jazzy/setup.bash`) if not already done. Then launch the MAVROS node with a UDP link:

```
ros2 run mavros mavros_node --ros-args -p fcu_url:=udp://0.0.0.0:14540@
```

Explanation: This command runs the MAVROS container and sets the `fcu_url` parameter. We bind to UDP port **14540**, which is the port PX4 uses for offboard API connections (like MAVROS) [8]. The `@` in the URL tells MAVROS to autodetect the remote endpoint. PX4 SITL, by default, broadcasts MAVLink on port 14540 and listens for responses on port 14557 [8]. When MAVROS starts, it will hear PX4's broadcast and establish the connection (PX4 will see MAVROS on the network and communicate).

You should see MAVROS log messages indicating a successful connection. For example, in the MAVROS terminal output:

- A line like `[ INFO] ... MAVROS connected to FCU` (showing the connection to PX4).
- Home position information being received (PX4 usually sends a home position message on connect).
- The current flight mode and other status echoed by MAVROS.

(If MAVROS does not connect, ensure PX4 is running and that the UDP ports are correct. The above command uses default ports; no additional configuration should be needed in SITL. The ROS 2 MAVROS node is analogous to the ROS1 `px4.launch` *with* `fcu_url:=udp://:14540@`*.)* [9]

1. **Verify MAVROS topics and EKF data in ROS 2:** Now that MAVROS is running, we can use ROS 2 topic tools to confirm data is flowing from PX4:

2. **Check connection state:**

```
ros2 topic echo /mavros/state --once
```

This will print one message from the `mavros/state` topic (of type `mavros_msgs/State`). Look for `connected: True` in the output, which indicates MAVROS is indeed connected to the PX4 FCU. You'll also see fields like `armed` (should be false at this point) and `mode` (likely `MANUAL` or `POSCTL` by default in SITL). For example:

```
connected: True
armed: False
mode: "POSCTL"
```

If `connected` is `False`, MAVROS hasn't linked to PX4 – recheck the fcu_url or PX4 status before proceeding.

3. **Check local position topic:**

```
ros2 topic echo /mavros/local_position/pose --once
```

This echoes one message from the `/mavros/local_position/pose` topic (type `geometry_msgs/PoseStamped`). This is the vehicle's position in the ROS ENU frame, as reported by the EKF via MAVROS. You should receive a message with coordinates corresponding to the origin. For instance:

```
header:
  frame_id: "map"
pose:
  position:
    x: ~0.0
    y: ~0.0
    z: ~0.0
  orientation: [...]
```

Here `(x~0, y~0, z~0)` indicates the vehicle is at the origin (ENU frame) – which matches the NED origin we saw on the PX4 side (the slight differences or sign inversion in z are due to NED->ENU conversion by MAVROS). The key is that we are **receiving data** on this topic, meaning the EKF's position is being published. If this topic returns no data or hangs, then MAVROS might not be receiving the local position (which would be a problem – ensure EKF2 is valid as per step 1 and that you launched MAVROS after EKF was ready).

4. **(Optional) Check global position:** You can similarly echo `/mavros/global_position/global` (or `.../raw/fix`) to see the global latitude/longitude. It should match ~47.3977, 8.5456 in a NavSatFix message. This is another confirmation that the EKF's global position is being communicated.

5. **Monitor continuously (optional):** Rather than `--once`, you can run `ros2 topic echo /mavros/local_position/pose` without `--once` to see a stream of pose updates. You should see the pose values updating at the rate PX4 publishes (~50 Hz). Initially the values will hover around zero (just noise) since the vehicle is sitting still at the origin.

6. **EKF health via MAVROS:** At this point, from the ROS side, we have: the FCU is connected and the local position is publishing. This implies the EKF is up and giving data (MAVROS doesn't publish `/local_position/pose` unless PX4's EKF has a position lock). We have effectively double-checked EKF health through ROS. You can also inspect the `/mavros/local_position/odom` topic (which includes velocities and pose covariance) or `/mavros/estimator_status` (if available in MAVROS2) for more insight. For a basic pipeline, the checks above are sufficient.

With MAVROS running, we can now run our analytics or mission script that uses these topics. The script will typically wait for the EKF to be ready (which we've essentially confirmed) before commanding a mission.

Terminal 3: Run the EKF Integration / Mission Script

In a third terminal, we'll execute the Python script (`waypoint_based_ekf_integration.py`) which automates the mission upload and monitoring. Make sure this terminal has the ROS 2 workspace sourced as well, if needed.

1. **Launch the Python EKF/mission script:** Run the script using ROS 2 or Python as appropriate. For example, if it's packaged in a ROS 2 node:

```
ros2 run <your_package_name> waypoint_based_ekf_integration.py
```

(If it's not installed as a ROS 2 node, you might run it directly: `python3 ~/path/to/waypoint_based_ekf_integration.py` – ensure you have `ros2` Python environment set up so it can connect to ROS topics.)

This script likely does the following: it waits for the vehicle to be ready, then sends a mission and commands the drone. At start, you should see some log output from the script indicating it's waiting for conditions. For instance, it may print messages like "Waiting for FCU connection..." or "Waiting for local position..." – essentially verifying the same things we did manually. Once those checks pass, the script will proceed.

1. **Script sets mission and arms the drone:** After the script confirms EKF is healthy and PX4 is ready, it will upload the mission waypoints (probably via a MAVROS service or topic) and then arm the vehicle and start the mission. Watch the terminals for the following:

2. In **Terminal 1 (PX4 shell)**: You should see status messages as the vehicle mode changes to **AUTO.MISSION** or **Offboard**, and an arming message. PX4 might log something like `[commander] Armed by command` and `[navigator] Mission received` etc., depending on how the script operates. You can also run `listener mission_result` or observe `mission_started` messages in PX4 if curious.

3. In **Terminal 2 (MAVROS)**: MAVROS will report mode changes (e.g., "Mode changed to AUTO.MISSION") and other info like "Waypoint mission received". It will also reflect the armed state (`armed: True` in `/mavros/state`).
4. In **Gazebo**: The drone model should take off and start executing the mission waypoints after a short delay (once armed and in Auto mode). You can visually confirm it following the expected path.

Throughout the mission, the EKF should remain healthy since we've ensured a strong GPS fix and sensor data. You can periodically run `ekf2 status` in the PX4 shell to ensure it still says local and global position valid (it should). Also, monitor `/mavros/local_position/pose` – the values will change as the drone moves, reflecting its position. This closes the loop: the script is using the EKF output to decide when to start, and once started, the EKF output (position) drives the mission progress.

> **Troubleshooting Tip:** If at any point the PX4 shell shows an error like **"EKF2 rejecting position"** or you see the mode switch get rejected with **"EKF not happy"**, it means the estimator didn't consider itself fully ready. Ensure you waited for the EKF status to be valid (steps above). In our case, using only simulated sensors, this should not occur as long as the GPS fix was present before arming.

Post-Run Checklist – Verifying Simulated GPS-Only EKF Functionality

Before concluding, go through this quick checklist to verify that everything worked as intended with **EKF2 using only the simulated GPS**:

- [ ] **Single GPS source in use:** The PX4 `sensor_gps` topic shows **only one** GPS instance, with a good fix. (Checked via `listener sensor_gps` – only one instance printed, with `fix_type >= 3` and coordinates ~47.3977 N, 8.5456 E [4].) No secondary or real GPS feed is interfering.
- [ ] **EKF2 has valid local & global position:** The `ekf2 status` output confirms `local position: valid` and `global position: valid` (i.e., both are 1/true) [10]. This means the EKF is fusing the fake GPS and other sensors correctly. There are no warnings about poor GPS or vision fusion; the estimator is in the nominal state.
- [ ] **EKF origin and frame alignment are correct:** The `vehicle_local_position` reference latitude/longitude (`ref_lat/ref_lon`) matches the known simulation home coordinates (within expected precision) and `ref_alt` is reasonable. The local position (x,y) at the start was near (0,0), and global flags `xy_global`, `z_global` are true – confirming the local frame is tied to the global frame. Height is consistent (barometer providing altitude, no jumps).
- [ ] **MAVROS is receiving EKF data:** The ROS topic `/mavros/local_position/pose` is publishing the drone's pose. During the mission, this should reflect the movement (the values changed as the drone moved). This indicates the EKF's output is successfully reaching the ROS layer.
- [ ] **Mission script respected EKF readiness:** The Python script waited for the connection and for the pose data to be valid before arming. (You would have seen its log output stall until those conditions were true, then proceed.) This ensured a smooth start with no "EKF still initializing" errors when the drone took off.
- [ ] **No real GPS influence:** At no point did a real/host GPS feed into the simulation. (In SITL, this typically isn't an issue unless HIL features are enabled. Our checks and the use of `SENS_EN_GPSSIM=1` guarantee only the simulated data was used [1].)

If all the above are confirmed, you have a fully working PX4 SITL environment where **EKF2** is reliably using the **simulated GPS as the sole position source**, with a consistent origin and stable altitude reference. The vehicle should be able to arm and fly in simulation without EKF position errors.

Appendix: Script Logic for Waiting on EKF Health

The `waypoint_based_ekf_integration.py` script is designed to ensure the drone only starts its mission when it's safe (i.e., when the EKF is ready). Below is a brief outline of how it utilizes ROS (MAVROS) topics to achieve that:

- **Monitoring** `/mavros/state`**:** The script subscribes to the `mavros/state` topic to get the connection and arming status of the vehicle. It will typically loop until `State.connected` is `True` – meaning MAVROS is connected to PX4 – before doing anything else. It may also check `State.armed` (expecting `False`) and perhaps the mode (e.g., waiting for manual mode or some initial mode) just to ensure the drone is in a standby state. Essentially, this guarantees that the script is communicating with PX4 properly.

- **Monitoring** `/mavros/local_position/pose`**:** The script also subscribes to the local position pose topic, which reflects the EKF's output. It will wait until this topic shows signs of a valid position. In practice, this could mean:

- Waiting for the first message on `/mavros/local_position/pose` (indicating the EKF has initialized and started publishing).
- Potentially checking the content of the pose message – for example, ensuring that the position is not `(0,0,0)` for an extended period or that there is a reasonable timestamp. Often, just receiving messages regularly is enough to assume the EKF is OK.

- (In more advanced cases, one might subscribe to `/mavros/estimator_status` or `/mavros/local_position/odom` to inspect the status flags or covariances, but using the pose topic is a simpler proxy for "EKF position ready".)

- **Waiting loop:** The script likely uses a loop or callback mechanism that only proceeds when both of the above conditions are met – i.e., **connected to PX4** and **local position is being published**. This is the crucial gating function: it prevents sending any mission or arming commands while the EKF might still be initializing. In effect, it's checking that `EKF2` has **"local_position = 1" and "global_position = 1"** (as we saw via `ekf2 status`) before moving on. It's doing so indirectly through the ROS topics.

- **Mission upload and arming:** Once the EKF is deemed healthy, the script uses MAVROS services or topics to upload the mission waypoints (e.g., calling a service like `/mavros/mission/push` with a list of waypoints). Then it likely switches the PX4 mode to **AUTO.MISSION** (or sends an Offboard command, depending on implementation). Finally, it sends an arm command (e.g., via `/mavros/cmd/arming`). MAVROS forwards these to PX4 (as MAVLink commands). Because the script waited, PX4 accepts the mode change and arm command without complaining about the EKF. The drone then takes off and follows the mission.

- **During flight:** The script might continue to monitor the state (for example, to detect if the mission is complete or if any failsafe occurs, etc.). But the critical part related to EKF was the pre-flight check. Once in the air, if the EKF stays good (which it should in our simulated GPS-only scenario), the script's job is essentially done aside from perhaps logging progress.

In summary, the **script ensures the vehicle does not arm or start its mission until the EKF2 is fully initialized with a valid position**. It does this by listening to the same MAVROS topics we examined manually. By waiting for `/mavros/state.connected == True` and a steady stream of `/mavros/local_position/pose` messages, the script guarantees that **only simulated GPS (with a proper fix)** is guiding the EKF and that the EKF's outputs (local and global position) are ready to be trusted for navigation. This integration of waypoint control with EKF status is vital for robust autonomous missions, and our steps above achieved exactly that. Safe flights! [2] [8]

---

[1] [2] Missing sensors for a Gazebo simulation - PX4 Autopilot - Discussion Forum for PX4, Pixhawk, QGroundControl, MAVSDK, MAVLink
https://discuss.px4.io/t/missing-sensors-for-a-gazebo-simulation/32787

[3] rotors_gazebo_plugins: gazebo_mavlink_interface.cpp Source File
http://docs.ros.org/en/melodic/api/rotors_gazebo_plugins/html/gazebo__mavlink__interface_8cpp_source.html

[4] [5] [6] [7] No global position in ekf2 - PX4 Autopilot - Discussion Forum for PX4, Pixhawk, QGroundControl, MAVSDK, MAVLink
https://discuss.px4.io/t/no-global-position-in-ekf2/43590

[8] 시뮬레이션 · PX4 Developer Guide
https://jalpanchal1.gitbooks.io/px4-developer-guide/content/kr/simulation/

[9] Can not trigger subscriber for mavros/local_position/pose, ... topics - ROS Answers archive
https://answers.ros.org/question/413445/

[10] Calibrate Sensors | ModalAI Technical Docs
https://docs.modalai.com/px4-calibrate-sensors/