ChatGPT

Understanding TF Relationships in the Planning Pipeline

Coordinate Frames and Reference Origin

In this system, a **global reference frame** (named `"reference"`) is defined based on a fixed GPS latitude/longitude. The launch file sets a reference latitude/longitude (default `37.540190, 127.076488`) which serves as the origin of the `"reference"` frame. All global positions (e.g. the vehicle's position and map waypoints) are expressed relative to this origin in a local Cartesian coordinate system (East-North-Up): - `reference` **frame:** A fixed world frame anchored at the specified reference GPS coordinates. Positions in this frame are in meters (X eastward, Y northward, Z up).
- `gps_antenna` **frame:** A moving frame attached to the vehicle's GPS receiver. This frame's pose (position and orientation) is updated relative to the `reference` frame, representing the vehicle's location.
- `os_sensor` **frame:** A sensor frame (e.g. LiDAR or camera) fixed relative to the GPS antenna. In the code, it's offset 0.1 m downward from the GPS antenna (no rotation offset). This represents the sensor's coordinate system on the vehicle.

The **TF tree** thus has `reference` as the root, with `gps_antenna` as a child, and `os_sensor` as a child of `gps_antenna`. In summary:

```
reference  —(dynamic transform)—>  gps_antenna  —(static transform)—>
os_sensor
```

This means the vehicle's **GPS antenna pose is given in the** `reference` **frame**, and the sensor is fixed relative to the GPS. These transforms allow converting data between the world and sensor frames as the vehicle moves.

GPS to Local Cartesian Conversion (`gps_to_local_cartesian_node`)

The `gps_to_local_cartesian` **node** converts raw GPS latitude/longitude from the receiver into local XY coordinates in the `reference` frame. It subscribes to the GPS fix topic (e.g. `/ublox_gps_node/fix`, a `sensor_msgs/NavSatFix`). On each fix message, it computes the planar offset from the reference lat/lon using a simple equirectangular projection:

```
double x = (lon_r - ref_lon_r) * cos(ref_lat_r) * R_EARTH;
double y = (lat_r - ref_lat_r) * R_EARTH;
```

Here `R_EARTH` is the Earth's radius (6,378,137 m) and `ref_lat_r`, `ref_lon_r` are the reference lat/lon in radians. This yields an approximate Cartesian coordinate (x East, y North) of the current GPS

reading relative to the origin. Altitude is ignored (the node sets z=0). The node then publishes these coordinates as a `geometry_msgs/PointStamped` on the topic `/local_xy`.

Frame convention: The published `PointStamped` carries the same header as the incoming GPS message. In practice, we treat this (x,y) as coordinates in the `"reference"` frame (the world ENU frame). Essentially, `/local_xy` provides the vehicle's current position in the reference frame.

Vehicle Pose TF Broadcaster (`vehicle_tf_broadcaster_node`)

The `vehicle_tf_broadcaster` **node** listens to the vehicle's position and heading, and broadcasts the corresponding TF transforms: - **Position:** It subscribes to the `PointStamped` from `/local_xy` (the output of the GPS converter). The callback simply caches the latest X, Y (and Z) position of the vehicle in the reference frame. - **Orientation (Yaw):** It also subscribes to a `/global_yaw` topic (std_msgs/Float32) which provides the vehicle's heading angle (orientation around Z). This yaw could come from an IMU or GPS-IMU fusion (in degrees or radians as configured – likely radians here).

Using this data, the node publishes two transforms: 1. **Dynamic transform:** `"reference" -> "gps_antenna"`. This represents the vehicle's pose in the world. Each broadcast (20 Hz) updates the `gps_antenna` frame's position to the latest X, Y coordinates (with Z, which might be a constant or altitude if available, set to the cached z) and orientation set by the yaw angle. In code, this is done by populating a `geometry_msgs/TransformStamped`:

```
t.header.frame_id = "reference";
t.child_frame_id  = "gps_antenna";
t.transform.translation.x = x_;
t.transform.translation.y = y_;
t.transform.translation.z = z_;
// Set rotation from yaw:
tf2::Quaternion q;
q.setRPY(0, 0, yaw_);
t.transform.rotation = {q.x(), q.y(), q.z(), q.w()};
tf_broadcaster_.sendTransform(t);
```

This means the `gps_antenna` frame is located at `(x_, y_, z_)` in the `reference` frame, and rotated by `yaw_` around Z relative to the reference frame axes. In other words, the vehicle's **forward direction** (GPS antenna's x-axis) is rotated `yaw_` radians from East (reference X-axis). This transform allows any point in the vehicle frame to be related to global coordinates. For example, if the vehicle is at (100, 50) in `reference` with yaw 90°, the transform places `gps_antenna` at that position with a 90° rotation. 2. **Static transform:** `"gps_antenna" -> "os_sensor"`. This is broadcast once at startup. In the code, `broadcastStaticSensorTF()` sets:

```
t.header.frame_id = "gps_antenna";
t.child_frame_id  = "os_sensor";
t.transform.translation.z = -0.1;
t.transform.rotation.w = 1.0; // no rotation
static_broadcaster_.sendTransform(t);
```

This declares that the `os_sensor` frame is fixed relative to the GPS antenna: it's 0.1 m below the GPS (translation in z), with no rotation offset [1]. All other translation components are zero, meaning `os_sensor` is directly beneath the GPS. The orientation being identity (`w=1.0`) means `os_sensor` shares the same orientation as `gps_antenna`.

With these two transforms, the TF tree is established. Now any point in the sensor's coordinate system can be transformed to the world frame (`reference`) by chaining `os_sensor -> gps_antenna -> reference`. Similarly, one can transform global points into the sensor frame (useful for local planning) by using the inverse (which the TF library provides via lookup transforms, as we'll see). The `vehicle_tf_broadcaster` essentially provides the vehicle's pose to the rest of the system.

Note: The `gps_antenna` frame here effectively serves a role similar to a vehicle base frame. If the GPS antenna isn't exactly at the vehicle's reference point (e.g. center of axle), this approach assumes the difference is negligible (except the vertical offset). In a more refined model, one might introduce a `base_link` frame at the vehicle's center and a static offset from base_link to gps_antenna, but in this code `gps_antenna` is used directly as the moving base frame.

Global Path Publishing (`status_colored_path_publisher_node`)

The `status_colored_path_publisher` node loads a predefined path (from a CSV file) and publishes it as a visualization Marker in the world frame. This represents the course or track that the vehicle should follow globally. Key points: - The CSV contains waypoints in latitude/longitude (and a covariance value for each). The node reads each line, converts lat/lon to local (x, y) coordinates using the same reference lat/lon origin as before. This ensures the path points align in the `reference` frame coordinates. Each point is stored along with its covariance (`cov`). - It then publishes a ROS Marker of type `LINE_STRIP` on topic `/global_path_marker`. In the code's timer callback, we see:

```
mk.header.frame_id = "reference";
mk.type = Marker::LINE_STRIP;
// ... fill mk.points with path coordinates ...
// Set color for each point depending on covariance vs threshold
if (p.cov > cov_threshold_) { col = red; } else { col = green; }
mk.colors.push_back(col);
marker_pub_->publish(mk);
```

The marker's `header.frame_id` is `"reference"`, meaning all the points in the path are interpreted in the global reference frame. The path is drawn as a continuous line through these points. Segments with higher covariance (error > threshold, e.g. >5 cm if threshold=0.0025) are colored red, others green, for debugging accuracy.

Publishing the global path in the `reference` frame allows other nodes to compare sensor data (cones, etc.) to this fixed path.

At this stage, we have: - The vehicle broadcasting its pose (`gps_antenna` in `reference`). - A global path defined in `reference` coordinates (as `/global_path_marker`). - Sensors (like cone detections) publishing in the `os_sensor` frame (explained next).

All these exist in the TF ecosystem, so we can transform between them as needed.

Cone Detection Visualization (`visualize_cones.py`)

(This script is optional and for visualization, but helps understand frames of sensor data.)

A perception system (not fully shown in the code) detects cones and publishes them on `/sorted_cones_time` as a custom message (`ModifiedFloat32MultiArray`). Essentially, it is a list of cone coordinates in the sensor frame. The `visualize_cones.py` node subscribes to `/sorted_cones_time` and visualizes each cone as a gray sphere in RViz. Important details: - The cones message's header frame is set to `"os_sensor"` (the script uses a parameter default `frame_id = 'os_sensor'`). This implies the (x, y, z) values in `msg.data` for each cone are relative to the sensor's origin and orientation (typically, the sensor is on the vehicle, so this is a **vehicle-relative measurement**). - The visualizer node simply takes triples of floats from the message and creates MarkerArray of spheres in the `os_sensor` frame for display. It doesn't transform them – it just uses the frame as given. So if the TF tree is correct, those cones will appear in the right place in the world when viewed in RViz (because RViz will use the `os_sensor->reference` TF to position them).

The main takeaway is that **cone detections are reported in the `os_sensor` frame**. This matches our TF setup – the cones' coordinates are relative to the sensor/car. We will need to relate these to the global path in order to classify and use them for planning.

Cone Side Classification (`classify_cones_by_side` node)

The `classify_cones_by_side` node receives the global path and the detected cones, and determines which cones are on the left or right side of the path. This is done by leveraging TF to transform coordinates between frames: - **Input:** It subscribes to `/global_path_marker` (the Marker published earlier) and caches the sequence of path points (in `reference` frame). It also subscribes to `/sorted_cones_time` (the cone list in `os_sensor` frame). - **Transform lookup:** When a new cones message arrives, the node first obtains the latest transform from the sensor frame to the reference frame using TF2. The code calls:

```
auto tf = tf_buffer_.lookupTransform("reference", "os_sensor", rclcpp::Time(0));
```

This requests the transform **from source `os_sensor` to target `reference`** at the current time. In effect, it gives the position and orientation of the sensor in the world (which comes from the dynamic `gps_antenna` transform plus the static offset). This transform is critical because it allows us to convert the cone coordinates from the vehicle's frame to the global frame. The node converts the transform into an Eigen rotation matrix `R` and translation vector `t` for calculation. - **Transform cones to global frame:** For each cone in the incoming list (each (x, y, z) in `os_sensor` frame), it applies the transform to get coordinates in the `reference` frame:

```
Eigen::Vector3d ps(x, y, 0.0);
Eigen::Vector3d pr = R * ps + t;
cones_ref.emplace_back(pr.x(), pr.y());
```

(Z is set to 0.0 here since we only care about ground projection for side classification.) After this, `cones_ref` contains all cone positions in the same frame as the path (reference frame). - **Nearest path segment projection:** The node then iterates through each transformed cone point and finds the closest segment of the global path line. It projects the cone onto that segment (essentially finding the perpendicular foot of the cone point onto the path line). This gives a projected point on the path and identifies which segment of the path is nearest. - **Left or right determination:** Using the path segment's direction vector and the vector from the projected point to the actual cone position, it computes a 2D cross product (essentially the **signed area / z-component of the cross product**). If the cross product's z-component is positive, the cone lies to the left of the path (by right-hand rule, assuming path direction as forward); if negative, to the right. Each cone is sorted into `left_out` or `right_out` list accordingly. - **Output:** The node publishes two Marker topics: - `/left_cone_marker` – a `visualization_msgs/Marker` of type SPHERE_LIST in frame `"reference"`, containing all left-side cone positions as blue spheres. - `/right_cone_marker` – a similar Marker containing all right-side cones as yellow spheres. In the code, when publishing, they set `marker.header.frame_id = "reference"` for both, and populate `marker.points` with the cone positions (x,y,0). This way, the classified cones are now expressed in the global frame, colored by side. RViz (or other nodes) can interpret their positions correctly as they share the frame with the path.

At this point, we have the cones categorized by side in the **reference/world frame**. The next step is to plan a path between them for the vehicle, which is done in the vehicle's local frame.

Local Reference Path Planning (`reference_path_planning.py`)

The **Reference Path Planner** node generates a smooth centerline path for the vehicle to follow, using the left/right cone markers. The trick here is that the planner will output a path **relative to the vehicle's current pose** (in the sensor frame) for local driving, while utilizing the global frame data. Key steps: - **Subscribe to cone markers:** The planner listens to `/left_cone_marker` and `/right_cone_marker` (the outputs of the classifier). Each time it receives an update (especially on the right cones, where it calls `plan_path()` at the end of the callback), it stores the cone positions. For example, in `cb_left` / `cb_right`, it does:

```
self.left_ref = [(p.x, p.y) for p in mk.points]   # left cones in reference frame
self.right_ref = [(p.x, p.y) for p in mk.points] # right cones in reference frame
```

These lists `left_ref` and `right_ref` now hold the coordinates of cones in the global reference frame (as published). - **Lookup transform (reference -> sensor):** Before planning the path, the node needs to bring these cone positions into the vehicle's coordinate frame (so that the path can be planned relative to the car's current position). It uses TF2 to get the transform from the reference frame to the sensor frame at the current time. In code:

```
tf = self.tf_buf.lookup_transform(self.sensor_frame, self.ref_frame, rclpy.time.Time(),
timeout=0.2)
```

where `self.sensor_frame` is `"os_sensor"` and `self.ref_frame` is `"reference"`. This call returns the transform that will convert a point in the `reference` frame to the `os_sensor` frame. Essentially, it's the **inverse** of the transform used in the classifier node. (If the car is at position (X, Y) and yaw θ in reference frame, this transform will translate by -(X,Y) and rotate by -θ, aligning the world such that the car becomes the origin.) - **Transform cones to sensor frame:** The planner applies this transform

to all left and right cone coordinates. Using a helper, it multiplies each point by the rotation and then adds the translation:

```
xs, ys = R_sr @ [px, py, 0] + t_sr
```

This yields `left_s` and `right_s` lists: the cone positions expressed in the **vehicle's sensor frame**. Now, in these coordinates, the car (sensor) is at the origin (0,0), facing along its +X axis. The cones are positioned relative to the car's current location and heading. - **Delaunay triangulation & midpoint extraction:** The core of the path planning algorithm is to find a middle path between the left and right cones. It performs a Delaunay triangulation on all cone points (left + right) in the sensor frame. From the triangulated mesh, it filters only the "bridge" edges that connect a left cone to a right cone. The midpoint of each such left-right edge is computed (these should lie roughly down the center between cones). All midpoints are then sorted in a logical order (starting from the nearest in front of the vehicle, chaining through nearest neighbors). - **Spline fitting:** A B-spline is fit through the sequence of midpoints to smooth the path. If spline fitting fails (e.g. not enough points), it falls back to linear interpolation. The result is a smooth centerline in continuous (parametric) form. - **Even spacing (resampling):** The spline is then sampled at a fixed interval (e.g. every 0.5 m of arc length) to produce a series of waypoints that are evenly spaced. This results in a list of `Waypoint` objects (each with x, y in sensor frame). Initially a default speed is assigned to each (which can be updated by a speed profile later). - **Publish local path and waypoints:** The planner publishes multiple outputs: - `/local_planned_path`: a `nav_msgs/Path` message in frame `"os_sensor"`. Each Pose in this path represents a point on the planned centerline, in the sensor (vehicle-relative) coordinates. The header stamp corresponds to the time of planning, and `header.frame_id = "os_sensor"`. This path can be used by a controller which is likely operating in the vehicle frame. - `/final_waypoints`: a `MarkerArray` of small spheres (magenta colored) for each waypoint, also in the sensor frame. This is mainly for visualization in RViz, showing the discrete waypoints along the path. Each marker has `ns="final_wp"` and frame "os_sensor". - `/waypoint_speed_bars`: a `MarkerArray` of vertical bars indicating the speed at each waypoint (green semi-transparent columns whose heights correspond to the waypoint speeds). Also in the sensor frame. This gives a quick visual indication of the planned speed profile along the path. - (Additionally, for debugging, it publishes `/delaunay_internal_lines` and `/delaunay_internal_midpoints` in the sensor frame, which show the bridging lines and their midpoints from the Delaunay step.)

Because the final path and waypoints are published in the `os_sensor` (vehicle) frame, they are **ego-centric**: the vehicle is always at or near the origin of this frame. As the vehicle moves and new cone data comes in, the planner will recompute a new local path relative to the updated position. Downstream controllers can then consume `/local_planned_path` or `/final_waypoints` to guide the vehicle along this path. The use of the TF transform ensures that at the moment of planning, the global positions of cones are correctly converted to the vehicle's instantaneous frame, so the local path is accurate.

Integrating GPS-IMU Fusion into the TF Chain

Given the above structure, a **GPS+IMU fusion package** (such as a Kalman filter from `robot_localization` or a custom EKF) can be integrated to improve the accuracy of the `reference -> gps_antenna` transform. In the current setup, the `vehicle_tf_broadcaster` is

directly using raw GPS and a separate yaw source. A fusion algorithm would typically produce a more stable and accurate pose estimate. Here's how to apply it:

- **Provide fused pose in the `reference` frame:** Configure the fusion output to use the same frames (`reference` as the global frame, and `gps_antenna` or equivalent as the vehicle frame). For example, if using `robot_localization`, you can set the `world_frame` to "reference" and the `base_link_frame` to "gps_antenna" (or whatever frame you consider the vehicle's reference). This way the filter will output the vehicle's pose (position and orientation) in the `reference` frame coordinates.
- **Replace or augment `vehicle_tf_broadcaster`:** Instead of manually broadcasting the GPS pose, you can have the fusion node publish the transform. Many fusion packages output a `nav_msgs/Odometry` or directly broadcast a TF. You should ensure it publishes a transform equivalent to `reference -> gps_antenna`. If the fusion node uses a different frame name (e.g. `base_link`), you have two options:
- **Match the frame names:** Adjust the fusion node's parameters to use `"reference"` as the global frame and `"gps_antenna"` as the vehicle frame. This is ideal because it slots directly into the existing pipeline — the `classify_cones` and `reference_path_planning` nodes will find the correct transform in TF without any code changes. For instance, if the fused odometry has `header.frame_id = "reference"` and `child_frame_id = "gps_antenna"`, it directly replaces the static/dynamic transform from `vehicle_tf_broadcaster` with a hopefully smoother one.
- **Use standard frames and adapt:** If you prefer standard ROS frames (like `"map"` for global and `"base_link"` for vehicle), you can still use them but would need to adjust the other components. You could publish a static TF alias if necessary (e.g., make `"reference"` a duplicate of `"map"`, or have a static transform `"gps_antenna"` coincident with `"base_link"`). The simplest might be to modify the launch or parameters so that `classify_cones_by_side` and `reference_path_planning` look for `"map"` and `"base_link"` if those are used by the fusion output. However, this requires changing the frame strings in those nodes or remapping frames, so aligning names upfront is easier.
- **Incorporate IMU yaw:** The fused orientation from IMU/GPS will provide a more robust yaw than a standalone compass or GPS heading. The fusion node will output a full orientation (quaternion). When using it, ensure that the orientation is properly applied to the TF. If the fusion directly broadcasts TF, it will handle that. If it outputs an Odometry or Pose, you might need a small node or use ROS TF static publisher utilities to broadcast the transform from that pose. Essentially, you would discontinue the manual `global_yaw` Float32 topic and instead use the fused orientation.
- **Maintain static sensor offset:** The static transform from `gps_antenna` to `os_sensor` remains valid. You should still broadcast the `gps_antenna -> os_sensor` transform (10 cm offset) unless your fusion or URDF setup already includes the sensor frame. In practice, you can keep the `vehicle_tf_broadcaster_node` but disable its dynamic broadcasting (since fusion will do it) and only use it (or `static_transform_publisher`) to send the static offset.

Summary: The GPS-IMU fusion package should take in the raw GPS and IMU data and output the **vehicle's pose in the `reference` frame**. By configuring its frames consistently (using `"reference"` and `"gps_antenna"` or providing the appropriate static transforms), its output can replace the direct GPS-based transform. This will plug into the existing pipeline as follows: - The classifier will use the fused TF (`reference -> gps_antenna -> os_sensor`) when transforming cones, improving accuracy (especially orientation). - The planner will use the same fused transform to place cones in the correct local coordinates. - The rest of the logic (global path in reference frame, sensor data in sensor frame) remains unchanged.

By clearly mapping the fusion output into the `reference` / `gps_antenna` frames, you ensure all parts of the system are speaking the same TF language. The attached code already establishes the necessary frame relationships; a fusion algorithm simply needs to maintain those, while providing a more accurate transform based on combined GPS/IMU inputs. In short, **make the fusion node publish the vehicle's pose as** `reference -> gps_antenna`, and the rest of the pipeline will directly benefit from the improved localization without additional modifications.

---

1 [PDF] Crop Rows Constraints for Agricultural SLAM - POLITesi
https://www.politesi.polimi.it/retrieve/6855166f-e84c-431c-82a6-c389083be251/final_thesis_chiara_relandini.pdf