State Estimation in ROS 2

ROSCon UK '25
EDINBURGH

ROSCon UK '25 EDINBURGH

WiFi: `Visit-Ed` / `682-rosftw`

Slides: https://tinyurl.com/rosconuk-se-slides

Docker container:

```
$ sudo apt install docker.io docker-compose-plugin
$ wget https://tinyurl.com/rosconuk-se-docker -O docker-compose.yaml
$ docker compose run rosconuk2025                # Runs the docker container and attaches to it
$ docker compose exec rosconuk2025 /bin/bash     # We'll need to attach another console for bags
```

Repository:

```
$ mkdir -p rosconuk_ws/src
$ cd rosconuk_ws/src
$ git clone https://github.com/locusrobotics/roscon-uk-2025-se-workshop
$ git clone https://github.com/locusrobotics/fuse -b $ROS_DISTRO
$ git clone https://github.com/cra-ros-pkg/robot_localization -b $ROS_DISTRO-devel
$ cd ..
$ rosdep install --from-paths src --ignore-src
$ colcon build --symlink-install
$ source install/setup.bash
```

Setup

Shortcuts and Help

Convenience environment variables:

- `$taskN` (with `N` = `1…8`) - Directory for the task in question
- `$bags` - The bag directory
- `$ws` - The `colcon` workspace root directory

Convenience `bash` aliases:

- `cb` - Builds the workspace and returns to your current directory
- `s` - Sources the workspace after building
- `vscode` - Runs Visual Studio Code with all the flags needed to get started (also have `vim`, `emacs`, `nano`, and `gedit`)

Help

All of the solutions to the tasks are kept in the `answers` branch of the repository. If you get stuck, try running `git diff answers <file in question>`.

Task instructions are also available in the repository.

Tom Moore
Vice President, Robotics Software at Locus Robotics
MSc in Artificial Intelligence, University of Edinburgh

Author of `robot_localization`
(Infrequent) contributor to `fuse`

LOCUS ROBOTICS

Dr. Stephen Williams
Locus Robotics

Dr. Bence Magyar
Locus Robotics

Thank you!

`robot_localization`
- GitHub Repository
- ROSCon 2015 Talk
- Paper

`fuse`
- GitHub Repository
- ROSWorld 2021 Talk

Other resources
- ROS 2 Navigation Survey Paper

Bag Datasets

- **Task 4**: Liang, J. et al. (2024). Global Navigation Dataset. *George Mason University Dataverse* https://dataverse.orc.gmu.edu/dataset.xhtml https://doi.org/10.13021/ORC2020/JUIW5F
- **Task 5**: Mallios, A.; Vidal, E.; Campos, R.; Carreras, M. (2017). Underwater caves sonar data set. *The International Journal of Robotics Research* 36, 1247-1251 https://cirs.udg.edu/caves-dataset/ https://doi.org/10.1177/0278364917732838

Relevant ROS REPs
- REP-103
- REP-105

References

Kalman Filter: algorithm that optimally estimates a system's state over time by combining noisy measurements with a predictive model

Model (Single Variable)

State: x_k
State transition model: f
Process noise: $w_k \sim N(0, q_k)$
State evolves as: $x_k = fx_{k-1} + w_k$

Observation: z_k
Observation model: h
Observation noise: $v_k \sim N(0, r_k)$
Observation is given as: $z_k = hx_k + v_k$

Kalman Filter Algorithm

Predict

$$x_k^{pred} = fx_{k-1}$$
$$p_k^{pred} = f^2 p_{k-1} + q_k$$

Correct

$$y_k = z_k - hx_k^{pred}$$
$$k_k = p_k^{pred} h / (h^2 p_k^{pred} + r_k)$$
$$x_k = x_k^{pred} + k_k y_k$$
$$p_k = (1 - k_k h) p_k^{pred}$$

Single Variable Example

State x_k is the linear velocity of our robot at time k

Initial State:

$x_{k-1} = 2.0$ // Initial velocity
$p_{k-1} = 0.1$ // Initial variance
$q = 0.05$ // Process noise variance (not time-dependent)
$f = 1.1$ // State transition model says the state should always increase by a factor of 1.1
$h = 1.0$ // Observation model says that the sensor measures velocity directly

Observation:

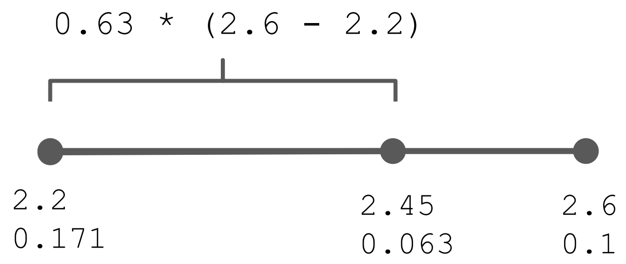
$z_k = 2.6$ // Sensor generates a value of 2.6
$r_k = 0.1$ // Measurement noise

Predict:

$x_k^{pred} = fx_{k-1} = 1.1 * 2.0 = 2.2$ // Project the state forward
$p_k^{pred} = f^2p_{k-1} + q = 1.1 * 1.1 * 0.1 + 0.05 = 0.171$ // Project the variance

Correct:

$y_k = z_k - hx_k^{pred} = z_k - x_k^{pred} = 2.6 - 2.2 = 0.4$ // Innovation (residual)
$k_k = p_k^{pred}h / (h^2p_k^{pred} + r_k) = p_k^{pred} / (p_k^{pred} + r_k) = 0.171 / (0.171 + 0.1) = 0.63099631$ // Kalman gain
$x_k = x_k^{pred} + k_ky_k = 2.2 + 0.63099631 * 0.4 = 2.452398524$ // Corrected state
$p_k = (1 - k_kh) * p_k^{pred} = (1 - k_k) * p_k^{pred} = (1.0 - 0.6399631) * 0.171 = 0.063099631$ // Corrected variance



Model (Multivariate)

State: $\mathbf{x}_k$
State transition model: $\mathbf{F}$
Process noise: $\mathbf{w}_k \sim N(0, \mathbf{Q}_k)$
State evolves as: $\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{w}_k$

Observation: $\mathbf{z}_k$
Observation model: $\mathbf{H}$
Observation noise: $\mathbf{v}_k \sim N(0, \mathbf{R}_k)$
Observation is given as: $\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k$

Kalman Filter Algorithm

Predict

$$\mathbf{x}_k^{pred} = \mathbf{F}\mathbf{x}_{k-1}$$
$$\mathbf{P}_k^{pred} = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q}$$

Correct

$$\mathbf{y}_k = \mathbf{z}_k - \mathbf{H}\mathbf{x}_k^{pred}$$
$$\mathbf{K}_k = \mathbf{P}_k^{pred}\mathbf{H}(\mathbf{H}\mathbf{P}_k^{pred}\mathbf{H}^T + \mathbf{R}_k)^{-1}$$
$$\mathbf{x}_k = \mathbf{x}_k^{pred} + \mathbf{K}_k\mathbf{y}_k$$
$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^{pred}$$

Extended Kalman Filter: a Kalman filter that linearises nonlinear state transition and observation models around the current estimate to perform prediction and correction.

Model

State: $\mathbf{x}_k$
State transition model: $f(\mathbf{x})$
Process noise: $\mathbf{w}_k \sim N(0, \mathbf{Q}_k)$
State evolves as: $\mathbf{x}_k = f(\mathbf{x}_{k-1}) + \mathbf{w}_k$

Observation: $\mathbf{z}_k$
Observation model: $h(\mathbf{x})$
Observation noise: $\mathbf{v}_k \sim N(0, \mathbf{R}_k)$
Observation is given as: $\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k$

Kalman Filter Algorithm

Predict

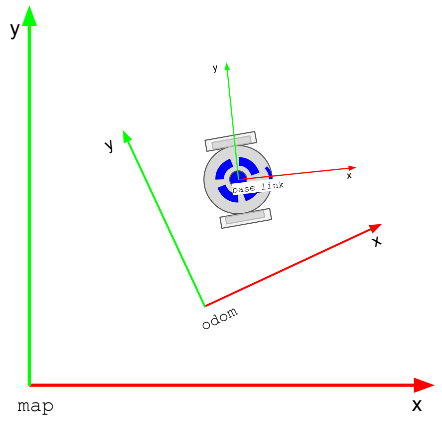
$$\mathbf{x}_k^{pred} = f(\mathbf{x}_{k-1})$$
$$\mathbf{F}_k = df/d\mathbf{x}_k$$
$$\mathbf{P}_k^{pred} = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^T + \mathbf{Q}_k$$

Correct

$$\mathbf{y}_k = \mathbf{z}_k - h(\mathbf{x}_k^{pred})$$
$$\mathbf{H}_k = dh/d\mathbf{x}_k$$
$$\mathbf{K}_k = \mathbf{P}_k^{pred} \mathbf{H}_k (\mathbf{H}_k \mathbf{P}_k^{pred} \mathbf{H}_k^T + \mathbf{R}_k)^{-1}$$
$$\mathbf{x}_k = \mathbf{x}_k^{pred} + \mathbf{K}_k \mathbf{y}_k$$
$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}) \mathbf{P}_k^{pred}$$

REP-103: Standard Units of Measure and Coordinate Conventions

- Right-handed coordinate system
- Axis orientation
- SI units: metres, radians, etc.
- Geographic locations using east north up (ENU) standard

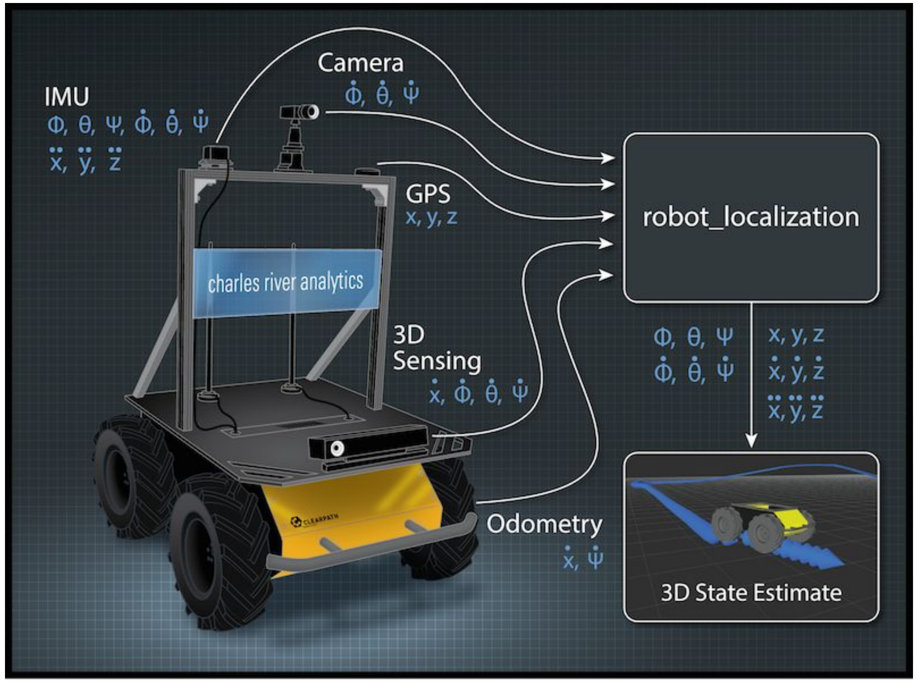


REP-105: Coordinate Frames for Mobile Platforms

- Establishes principal coordinate frames and their relationships
- `base_link`: rigidly attached to some point on the robot base, typically the centroid
- `odom`: world-fixed frame that is continuous (no discrete jumps), but subject to drift
- `map`: world-fixed frame that is not subject to drift, but may not be continuous
- `earth`: allows robots with differing map frames to interact, not typically used

Relevant REPs

robot_localization



- Spiritual successor to `robot_pose_ekf`
- Contains implementations of an Extended Kalman Filter and an Unscented Kalman Filter
- Generic omni-directional 3D state transition model
- Can support any number of inputs
- Supported input message types:
  - `nav_msgs/Odometry`
  - `geometry_msgs/PoseWithCovarianceStamped`
  - `geometry_msgs/TwistWithCovarianceStamped`
  - `sensor_msgs/Imu`
- Currently in use on tens of thousands of robots worldwide
- Does not actually perform localisation (just state estimation)

Kinematic (State Transition) Model

- Omnidirectional model
- Uses Euler angles (so be careful of gimbal lock!)
- State vector is 15D: $[x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \ddot{x}, \ddot{y}, \ddot{z}]$

$$\hat{x} = x + \dot{x}cos(\psi)cos(\theta)\Delta t + \dot{y}(cos(\psi)sin(\theta)sin(\phi) - sin(\psi)cos(\phi))\Delta t + \dot{z}(cos(\psi)sin(\theta)cos(\phi) + sin(\psi)sin(\phi))\Delta t$$

$$\hat{y} = y + \dot{x}sin(\psi)cos(\theta)\Delta t + \dot{y}(sin(\psi)sin(\theta)sin(\phi) + cos(\psi)cos(\phi))\Delta t + \dot{z}(sin(\psi)sin(\theta)cos(\phi) - cos(\psi)sin(\phi))\Delta t$$

$$\hat{z} = z - \dot{x}sin(\theta)\Delta t + \dot{y}cos(\theta)sin(\phi)\Delta t + \dot{z}cos(\theta)cos(\phi)\Delta t$$

$$\hat{\phi} = \phi + \dot{\phi}\Delta t + \dot{\theta}sin(\phi)tan(\theta)\Delta t + \dot{\psi}cos(\phi)tan(\theta)\Delta t$$

$$\hat{\theta} = \theta + \dot{\theta}cos(\phi)\Delta t - \dot{\psi}sin(\phi)\Delta t$$

$$\hat{\psi} = \psi + \dot{\theta}\frac{sin(\phi)}{cos(\theta)}\Delta t + \dot{\psi}\frac{cos(\phi)}{cos(\theta)}\Delta t$$

Basic Configuration

```yaml
ekf_filter_node:
    ros__parameters:
        frequency: 30.0                 # How frequently we publish (even if filter has not updated)
        sensor_timeout: 0.1             # If no sensor data in this time, we do a prediction (without correction)
        two_d_mode: false               # If enabled, 3D dimensions (z, roll, pitch, and their derivatives) are forced to 0
        transform_timeout: 0.0          # How long to wait for required transforms to be available
        print_diagnostics: true         # Whether to publish diagnostics
        publish_tf: true                # Whether to publish the output of the filter to /tf

        map_frame: map                  # The name of your REP-105 map frame (not needed if world_frame == odom_frame)
        odom_frame: odom                # The name of your REP-105 odom frame
        base_link_frame: base_link      # The name of your REP-105 base_link frame. Will be the child_frame_id in the output.
        world_frame: odom               # The world frame that will be the frame_id in the output.

        odom0: example/odom                      # Topic type + number (odomN, poseN, twistN, imuN) and name
        odom0_config: [false, false, false,      # x, y, z
                       false, false, false,      # roll, pitch, yaw
                       true,  true,  false,      # x velocity, y velocity, z velocity
                       false, false, true,       # roll velocity, pitch velocity, yaw velocity
                       false, false, false]      # x acceleration, y acceleration, z acceleration
```

Task 1: Basic Planar Robot

In the first task, we will help R2-D2 to estimate his state as he attempts to navigate an Imperial warehouse.

- R2's planar bag data is stored here: `$bags/planar/planar.db3`.
- Analyse the bag with `ros2 bag info $bags/planar/planar.db3`.
  - Play the bag with `ros2 bag play $bags/planar/planar.db3 --clock`.
  - Get a sense of the available topics
  - Look at the transforms that are available in `/tf_static` (make note of the `base_link`->`imu` transform)
  - We are going to build our state one input at a time. The bag contains:
    - Wheel encoder odometry
    - IMU data
    - Visual odometry
    - Map-relative pose data

Task 1a: Odometry Only

1. Edit the file `$task1/config/odometry.yaml`
   - We want to make our first odometry (as in `nav_msgs/Odometry`) input our wheel encoder odometry. Set the topic for `odom0` accordingly.
   - For this exercise, we will fuse the `x` velocity, the `y` velocity, and the `yaw` velocity from the wheel encoders
     i. If R2-D2 is a differential drive robot, why are we fusing the y velocity?
2. Run the filter and `rviz2` with:
   Terminal 1: `ros2 launch task1 ekf.launch.xml`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
   - For comparison, we show the raw wheel encoder data alongside the EKF output.
   - What do you note about the output?
3. The bag starts and ends at the same location. Use the `rviz2` measurement tool to measure the distance from the robot’s first pose to its last. Make a mental note of the value!

Task 1b: Odometry + IMU

1. Edit the file `$task1/config/odometry_imu.yaml`
   - We will now be adding IMU sensor data to our filter
   - The wheel encoder odometry configuration has been provided for you
   - You need to now fill out the configuration for the IMU topic. We want to fuse `yaw` velocity and `x` acceleration from the sensor.
   - R2's holographic projector is bulky and made mounting the IMU difficult, so his designers mounted the IMU such that `+X` points to the ground, `+Y` points to R2's right, and `+Z` points towards his back.
     i. **This will have ramifications for the sensor configuration!**
2. Run the filter and `rviz2` with:
   Terminal 1: `ros2 launch task1 ekf.launch.xml include_imu:=True`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
   - The launch file runs two instances: one has our previous odometry-only config, and one has odometry + IMU. Raw wheel encoder data is also displayed. What do you note about them?
   - Note the distance from the last pose to the first!

Task 1c: Odometry + IMU + VO

1. Edit the file `$task1/config/odometry_imu_vo.yaml`
    a. We will now add visual odometry data as an input to the filter
    b. The wheel encoder odometry and IMU configurations have been provided for you
    c. As with wheel encoder odometry, we want to fuse x, y, and yaw velocities into the filter
2. Run the filter and `rviz2` with:
    Terminal 1: `ros2 launch task1 ekf.launch.xml include_imu_vo:=True`
    Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
    a. We now have three EKF instances running:
        i. One with just wheel encoder data
        ii. One with wheel encoder + IMU data
        iii. One with wheel encoder, IMU, and visual odometry data
        iv. We are also still displaying the raw wheel encoder data
    b. What do you note about the output?

Advanced Configuration

```
ekf_filter_node:
    ros__parameters:
        # ...basic parameters here...

        transform_time_offset: 0.0  # Offset we can use to future-date the transform

        debug: false                              # Produces an absurd amount of debug output
        debug_out_file: /path/to/debug/file.txt   #

        odom0_queue_size: 2         # ROS queue size for the topic odom0
        odom0_differential: false   # Converts consecutive pose measurements to velocity data
        odom0_relative: false       # All pose data is reported relative to the first received message
        odom0_pose_rejection_threshold: 5.0    # Mahalanobis distance thresholds that can be used to reject outliers
        odom0_twist_rejection_threshold: 1.0   #

        imu0_remove_gravitational_acceleration: true  # Removes gravitational acceleration if your IMU doesn't

        # The initial covariance (P) for the filter state
        initial_estimate_covariance: [1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9, 1e-9]

        # The Q matrix
        process_noise_covariance: [0.05, 0.05, 0.06, 0.03, 0.03, 0.06, 0.025, 0.025, 0.04, 0.01, 0.01, 0.02, 0.01, 0.01, 0.015]
```

Advanced Configuration

```
ekf_filter_node:
    ros__parameters:
        # ...basic parameters here...
        # ...other advanced parameters here...

        smooth_lagged_data: true        # Whether to allow the filter to handle out-of-sequence measurements
        history_length: 1.0             # The length of the history stored for out-of-sequence measurement handling

        predict_to_current_time: true   # Whether or not we always predict to the current time before publishing

        dynamic_process_noise_covariance: true  # Whether we scale Q based on the robot's velocity
```

Task 2a: Process Noise

Tuning the process noise covariance matrix can produce very different results.

1. Edit the file `$task2/config/odometry_modified_pnc.yaml`
    - Recall that in Task 1a, we fused just wheel encoder odometry, but our output state estimate did not very closely match the input wheel encoder data.
        - Why?
    - Edit the `process_noise_covariance` for the wheel encoder odometry by increasing the values for x velocity, y velocity (not really necessary), and yaw velocity.
2. Now run
   Terminal 1: `ros2 launch task2 ekf.launch.xml modified_pnc:=True`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
    - The output shows the original configuration alongside your updated configuration. What do you note about the output from the updated configuration?

Task 2b: Differential Mode

Sometimes, a topic contains only pose data, but you may not want to fuse that into your state estimate (e.g., if you have two pose sources, or the pose data is too infrequent).

1. Even though our VO data produces pose and velocity data, we’re going to pretend it only contains pose data, and that we don’t want to use it.
2. Edit the file `$task2/config/odometry_vo_diff.yaml`
3. The `odom1` sensor should have a topic of odometry_visual, and we should be fusing x, y, and yaw (NOT velocity!)
4. Enable differential mode for `odom1`
5. After editing the config, run the following:
   Terminal 1: `ros2 launch task2 ekf.launch.xml differential:=True`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
6. The `rviz2` output also shows odometry + VO data that is fusing only velocity (note: none of our estimates are using the IMU). What do you note?

"Two-tier" Setup

```
ekf_filter_node_tier1:
    ros__parameters:
        # ...other configuration...

        map_frame: map
        odom_frame: odom
        base_link_frame: base_link
        world_frame: odom

        # Continuous input sources

ekf_filter_node_tier2:
    ros__parameters:
        # ...other configuration...

        map_frame: map
        odom_frame: odom
        base_link_frame: base_link
        world_frame: map

        # Continuous input sources
        # Global pose source(s)
```

Task 3: Two-tier Setup

It turns out that R2-D2 has a map of the warehouse! He's going to use it to localise himself.

1. Edit the file `$task3/config/two_tier.yaml`
    - The config for the `odom`->`base_link` instance has been provided for you.
    - Add parameters for a second node to the same config file. The node's name is `ekf_node_tier2`.
    - The `ekf_node_tier2` should have a `world_frame` of map.
    - It should have the exact same inputs as the `ekf_node_tier1`
    - It should also have a new input for a topic called `pose_global`. That topic contains poses in the map frame that provide an absolute reference for the filter. We want to fuse x, y, and yaw from this source.
    - We want the filter to trust the pose data, but not absolutely. Tune your `process_noise_covariance` accordingly.
2. After editing the config, run the following:

   Terminal 1: `ros2 launch task3 ekf.launch.xml`

   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`

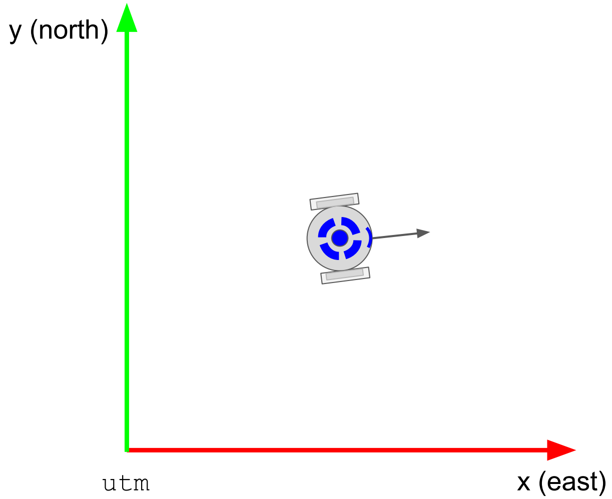
Working with GPS Data

```
$ ros2 interface show sensor_msgs/msg/NavSatFix

std_msgs/Header header

NavSatStatus status

float64 latitude
float64 longitude

float64 altitude

float64[9] position_covariance

uint8 position_covariance_type
```

Working with GPS Data

We need a way to transform GPS coordinates to our world frame (typically `map`).

The Universal Transverse Mercator (UTM) coordinate system provides a (mostly) convenient way to accomplish this. It divides the earth into 60 zones, with each zone having a portion of the earth ellipsoid projected onto it. We can then treat each zone as a two-dimensional coordinate frame with metre units. navsat_transform_node uses headers from an open source library to carry out this conversion.



Working with GPS Data

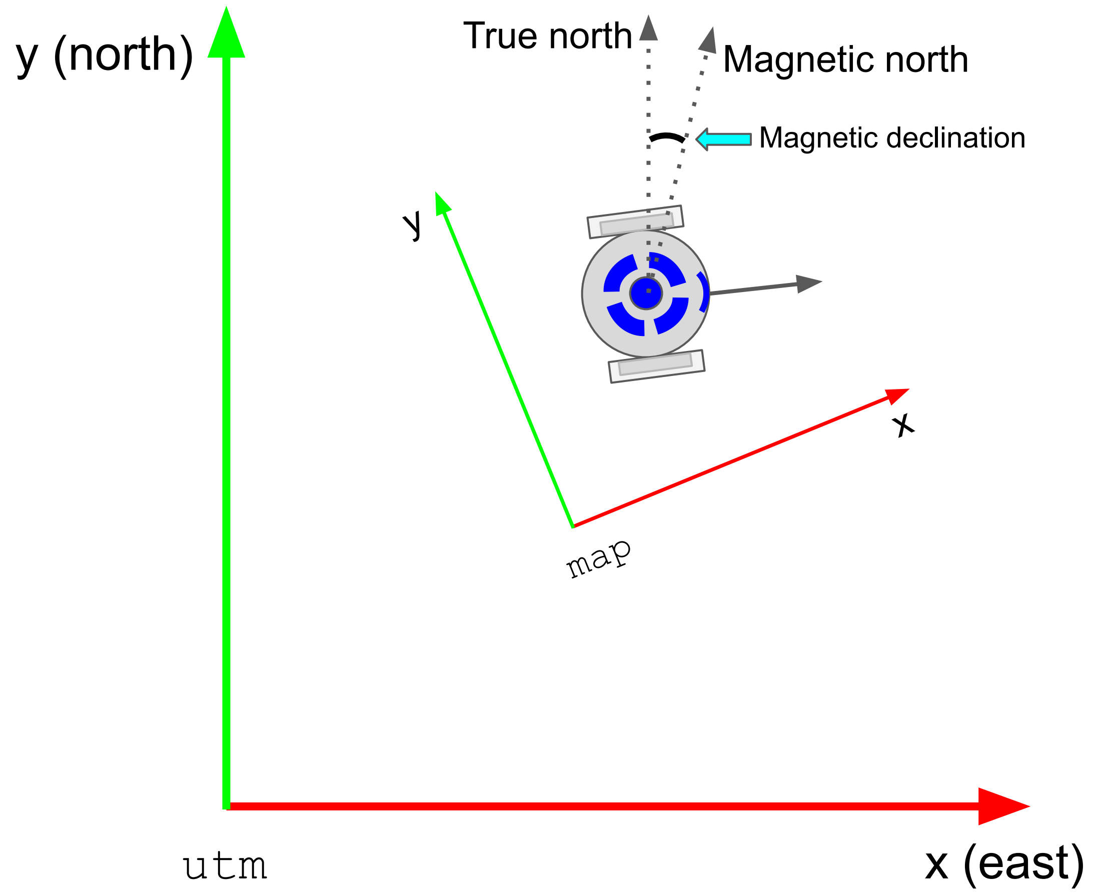
But we’re not quite done. In order to compute a transform from the UTM grid to our world frame, we need two things:

1. Our pose in the map frame (position and orientation)
2. Our pose in the UTM frame

For (2), we need to know our orientation within the UTM grid! The best way to obtain this is via a magnetometer, as provided by many IMU devices.

But now we have an additional problem: most commercial IMU devices report a heading of 0 at *magnetic north*, and not *geographic east*. In order to compensate for this, we need to know the magnetic declination for the robot’s location. This allows us to obtain geographic (“true”) north, and then we can trivially compute geographic east.

Working with GPS Data



navsat_transform_node

Converts GPS coordinates to poses that can be fused into the state estimate in our `map`-frame EKF.

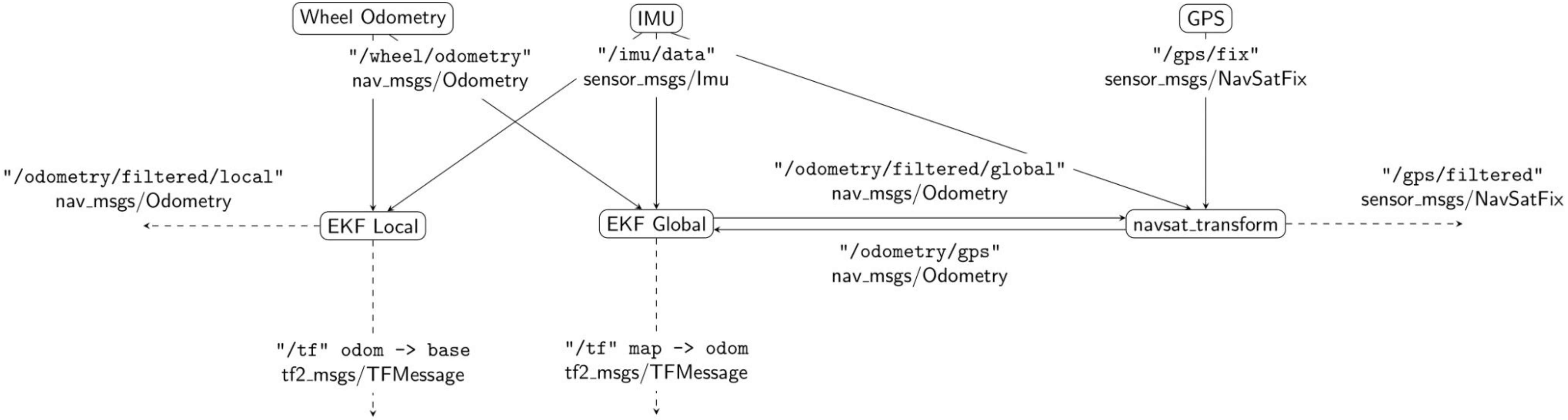
Inputs

- EKF posterior pose at the start of `navsat_transform_node`'s execution
- Earth-referenced orientation (can come from IMU *or* EKF, if the EKF's orientation is earth-referenced)
- GPS coordinates

Outputs

- GPS poses that have been transformed to the `map` coordinate frame
- (Optional) `utm`->`map` transform
- (Optional) EKF `map`-frame poses that have been transformed back into GPS coordinates

Data Flow


Wheel Odometry
"/wheel/odometry"
nav_msgs/Odometry
IMU
"/imu/data"
sensor_msgs/Imu
GPS
"/gps/fix"
sensor_msgs/NavSatFix
"/odometry/filtered/local"
nav_msgs/Odometry
EKF Local
EKF Global
"/odometry/filtered/global"
nav_msgs/Odometry
navsat_transform
"/gps/filtered"
sensor_msgs/NavSatFix
"/odometry/gps"
nav_msgs/Odometry
"/tf" odom -> base
tf2_msgs/TFMessage
"/tf" map -> odom
tf2_msgs/TFMessage

Configuring navsat_transform_node

```
navsat_transform_node:
    ros__parameters:
        frequency: 30.0                         # Frequency of main run loop
        delay: 3.0                              # How long we wait until we compute the utm->world (usually map) frame transform

        magnetic_declination_radians: 0.0       # Obtain value from http://www.ngdc.noaa.gov/geomag-web/, convert to radians
        yaw_offset: 0.0                         # If the IMU doesn't report 0 facing east after correcting for magnetic declination,
                                                # enter the value needed to get the IMU to report 0 when facing east here

        zero_altitude: false                    # Zeros out the altitude that gets reported in the output
        broadcast_utm_transform: false          # Whether to publish the utm->world transform
        publish_filtered_gps: false             # Publishes our EKF output as GPS coordinates

        use_odometry_yaw: false                 # If your EKF node already has an earth-referenced orientation, you can use it

        wait_for_datum: false                   # Tells the node to wait until we manually specify a datum (utm-frame origin)
        datum: [55.944904, -3.186693, 0.0]      # If wait_for_datum is true, we will use this value. If wait_for_datum is true and
                                                # this parameter is not specified, we will wait for a service call.
```

Two-tier Setup with GPS Data

```
ekf_filter_node_tier1:
    ros__parameters:
        # ...familiar configuration...

ekf_filter_node_tier2:
    ros__parameters:
        # ...familiar configuration...

        odom1: odometry/gps
        odom1_config: [true,  true,  false,
                       false, false, false,   # If operating in 3D, we would fuse Z (altitude) here
                       false, false, false,
                       false, false, false,
                       false, false, false]

navsat_transform_node:
    ros__parameters:
        frequency: 30.0
        delay: 3.0
        magnetic_declination_radians: -0.2413
        yaw_offset: -1.570796327
        zero_altitude: true
```

Task 4: GPS Data

In this task, R2-D2 will use his GPS sensor to keep his state estimate from drifting. The bag contains a (probably) unintentional, but useful, outage in sensor data. We will see how this affects our state estimate.

- R2's GPS bag data is stored here: `$bags/gps/gps.db3`.
- Analyse the bag with `ros2 bag info $bags/gps/gps.db3`.
  - Play the bag with `ros2 bag play $bags/gps/gps.db3 --clock`.
  - Get a sense of the available topics
  - Look at the transforms that are available in `/tf_static`

Task 4: GPS Data

1. Navigate to the bags/gps directory
2. Run `ros2 bag play gps.bag` in one terminal
3. In another terminal, run `ros2 topic echo /r2d2/gps`
4. Note the first reported GPS position
5. Go to https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml
6. Obtain the magnetic declination. Remember that you must convert it to radians, and that counter-clockwise is *positive*.
7. Edit the file `$task4/config/gps.yaml`
8. Add the GPS sensor. Remember that we are only fusing `x` and `y` position.
9. Run:
   Terminal 1: `ros2 launch task4 ekf.launch.xml`
   Terminal 2: `ros2 bag play $bags/gps/gps.db3 --clock`

Task 5: Operating in 3D

So far, we've operated with `two_d_mode` set to `true`. But many robots operate in 3D. In this task, R2-D2 is going for a swim!

- R2's undersea adventure bag data is stored here: `$bags/subsea_3d/subsea_3d.db3`.
- Analyse the bag with `ros2 bag info $bags/subsea_3d/subsea_3d.db3`.
  - Play the bag with `ros2 bag play $bags/subsea_3d/subsea_3d.db3 --clock`.
  - Get a sense of the available topics
  - Look at the transforms that are available in `/tf_static`

Task 5: Operating in 3D

1. Edit the file `$task5/config/subsea_3d.yaml`
2. We have three sensors/input topics: `velocity`, `depth`, and `imu`.
    a. For the velocity sensor, we want to fuse only the linear velocity dimensions
    b. For the depth sensor, we want to fuse only `z` position
    c. For the IMU, we want to fuse orientation *and* angular velocity
3. Fill out the missing values (search for ‘`?`’)
4. Run:
   Terminal 1: `ros2 launch task5 ekf.launch.xml`
   Terminal 2: `ros2 bag play $bags/subsea_3d/subsea_3d.db3 --clock`

Time to wake up!

Factor Graph: graphical representation of a function factorisation (here, a probability distribution)

Can frame state estimation as finding most likely/optimal value, X*, of the variables X, over a joint distribution of those variables and observations, Z

$$X^* = \underset{X}{arg\ max}\ P(X, Z)$$

The joint distribution can be factored into a product of measurement probabilities

$$P(X, Z) \propto \prod_i P(z_i | X)$$

If we assume each measurement probability is Gaussian…

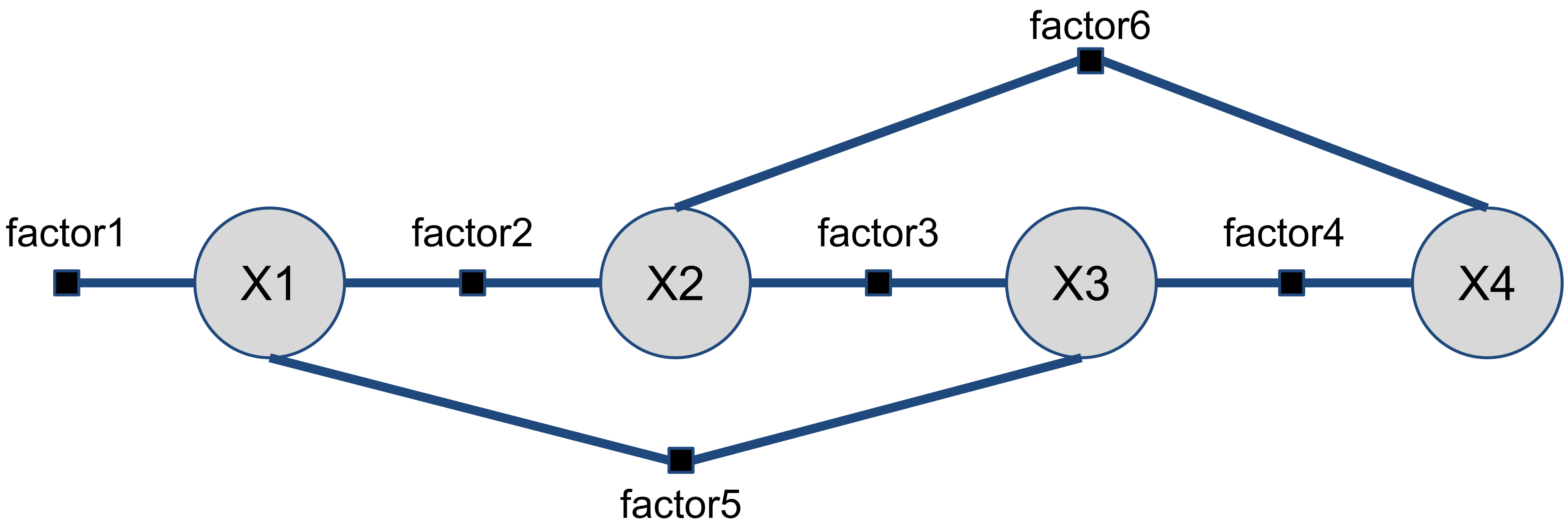
$$P(z_i|X) = -\frac{1}{2}\exp\left(\left(z_i - h(X)\right)^T \cdot \Sigma^{-1} \cdot \left(z_i - h(X)\right)\right)$$

Then we can use the “negative log” trick and express the optimisation as a least-squares problem

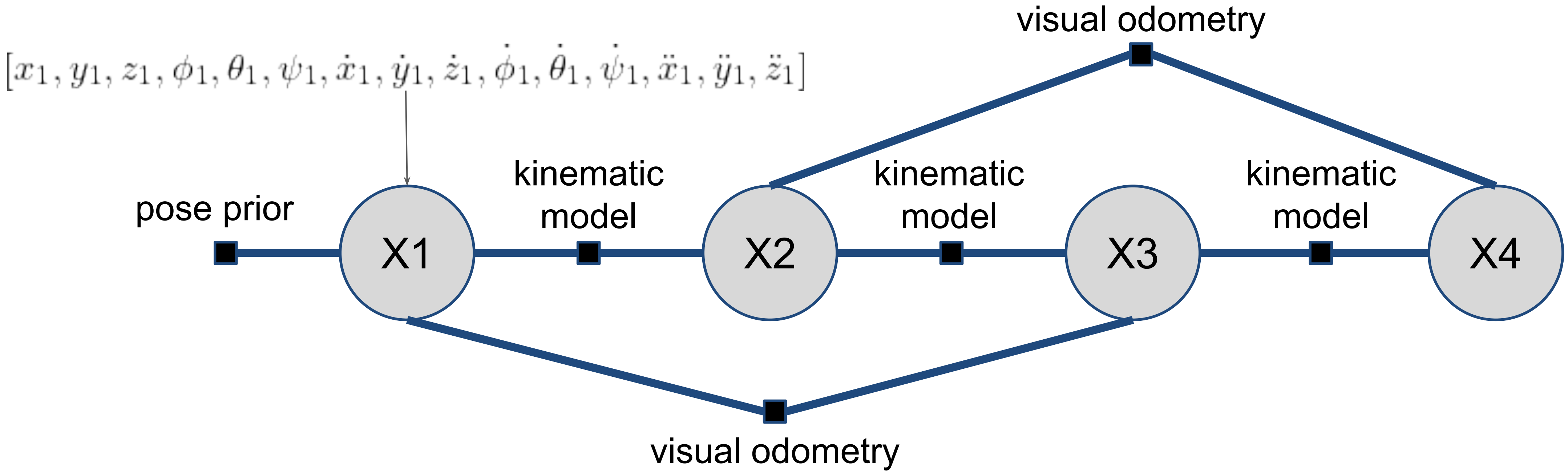
$$X^* = \underset{X}{arg\ min} \sum_i \left(\frac{z_i - h(X)}{\Sigma^{\frac{1}{2}}}\right)^2$$

Factor Graph: graphical representation of $\prod_i P(z_i|X)$

Pictures > LaTeX



Pictures > LaTeX



FUSE

fuse

- Spiritual successor to `robot_localization`
- Main nodes are the `fixed_lag_smoother` node and the `batch_optimizer` node
- Contains a slew of sensor and motion model constraints
- Completely plugin-based (highly extensible):
  - Optimisers
  - Variables
  - Sensor models (constraints)
  - Motion models (constraints)
  - Publishers
  - Loss functions
- Core optimisation is built using Google Ceres
- Currently in use on tens of thousands of robots worldwide

Why Use `fuse`?

Fuse has numerous advantages over `robot_localization`, some of which are limitations of EKFs in general, and others are due to the rigidity and assumptions of the EKF implementation in `robot_localization`.

- Can have any variables you want in your state vector (no faking 2D needed!)
- Sensor models can be added, and don't assume direct measurement of the state
- Relative measurements are possible, because the factor graph allows us to tie multiple states together with any given constraint
- Can apply `fuse` to problems in state estimation, mapping, sensor calibration, or even local planning!
- Quality-of-life improvements
  - Can specify an "ignition" sensor to avoid race conditions
  - General parameter setting is much better designed and cleaner

Why Use `robot_localization`?

Reasons are few and shrinking

- Can have lower CPU overhead, especially for very high-rate applications
- You need to work in 3D*
- You need to work with GPS data*

Key Objects in `fuse`

- Sensor Models
  - Model sensor measurements
  - Typically have the form $\underbrace{\Sigma^{-\frac{1}{2}}}_{\text{Covariance}} \cdot (\underbrace{z_i}_{\text{Measurement}} - \underbrace{h(X)}_{\text{Measurement model}})$

Covariance Measurement Measurement model

Key Objects in `fuse`

- Motion Models
  - Model the kinematics of the robot
  - Typically have the form $\Sigma^{-\frac{1}{2}} \cdot (x_j - f(x_i))$

Covariance

Current state

Previous state

Kinematic model

Key Objects in `fuse`

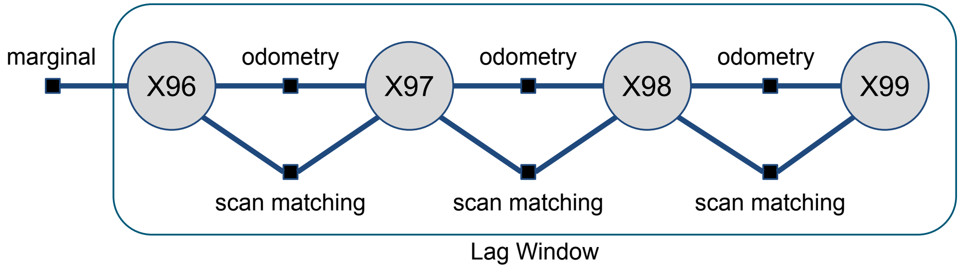
- Publishers
  - Extract information from the graph
  - Broadcast it to ROS
  - Can publish transforms (e.g., `odom->base_link`)

Key Objects in `fuse`

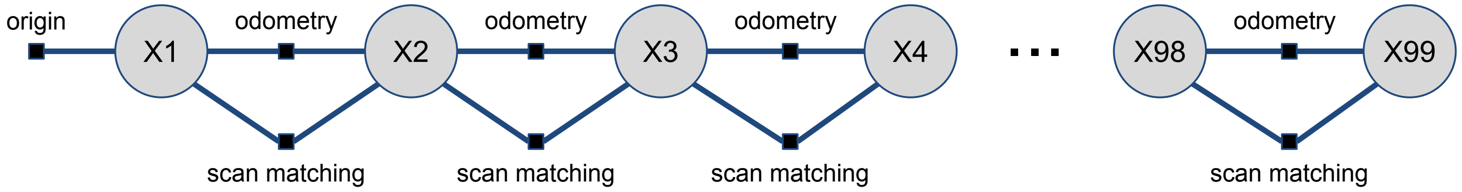
- Optimisers
  - “Central” objects (the main nodes typically are just wrappers around these)
  - Coordinate sensors, motion models, and publishers
  - Carry out the numerical optimisation to produce the optimal variable values

`fixed_lag_smoother_node` and `batch_optimization_node`

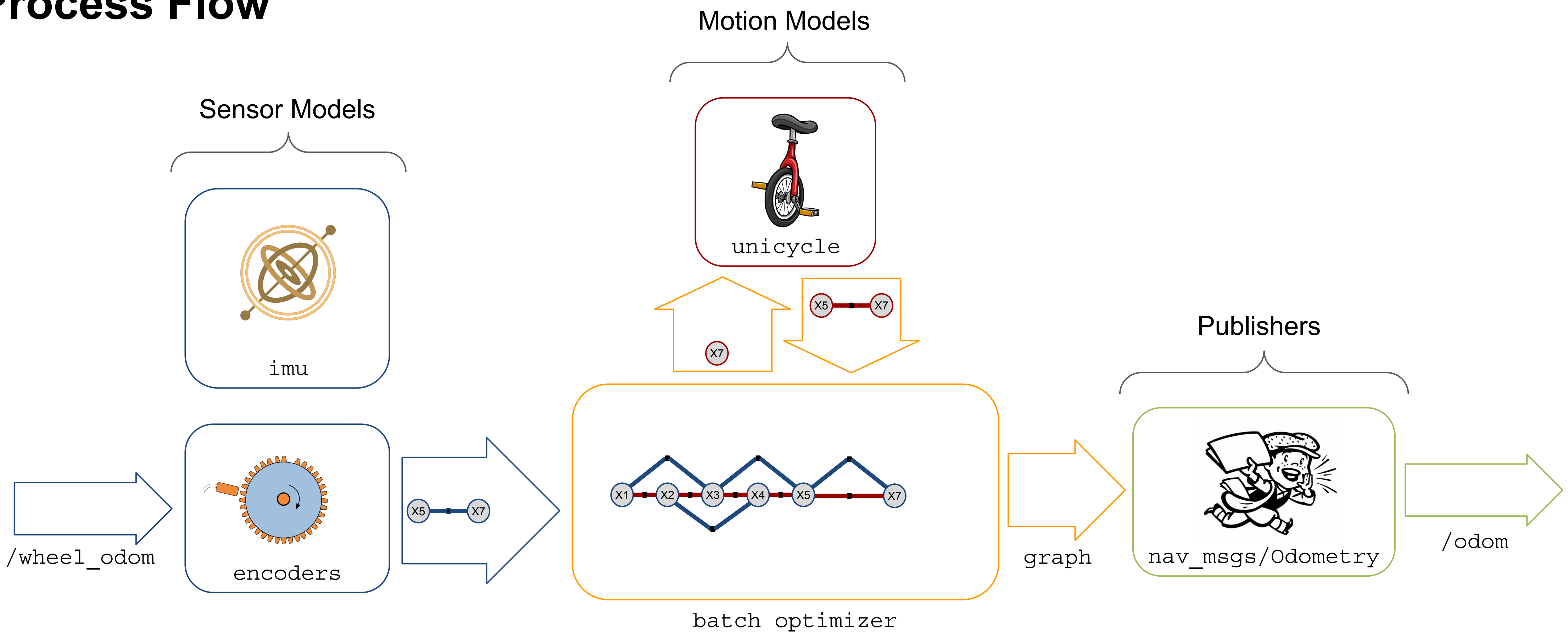
- Difference mainly comes down to graph retention policy
  - The `fixed_lag_smoother_node` retains a short rolling window history of state variables and constraints so as to produce a constantly updated state estimate



  - The `batch_optimization_node` retains all variables and constraints, and is most useful for applications like SLAM



Process Flow



Basic Configuration for the `fixed_lag_smoother_node`

```
optimization_frequency: 20  # How many times we carry out optimisation per second
transaction_timeout: 0.01   # If adding a transaction fails, and this amount of time elapses, we will drop it
lag_duration: 0.5           # How much of a state variable history to keep

motion_models:              # Motion model declarations (usually one)
  unicycle_motion_model:
    type: fuse_models::Unicycle2D

unicycle_motion_model:      # Parameters specific to the motion model we've selected
  process_noise_diagonal: [0.100, 0.100, 0.100, 0.100, 0.100, 0.100, 0.1, 0.1]  # Same as Q matrix in the EKF

sensor_models:                                # Sensor model declarations
  initial_localization_sensor:                # Ignition sensor (provides start pose)
    type: fuse_models::Unicycle2DIgnition
    motion_models: [unicycle_motion_model]    # Which motion model to use to tie constraints together
    ignition: true
  odometry_sensor:                            # Wheel odometry sensor
    type: fuse_models::Odometry2D
    motion_models: [unicycle_motion_model]
  imu_sensor:                                 # IMU sensor
    type: fuse_models::Imu2D
    motion_models: [unicycle_motion_model]
```

Basic Configuration for the `fixed_lag_smoother_node`

```
...

initial_localization_sensor:  # Ignition sensor-specific parameters
  publish_on_startup: true
  #                x      y      yaw    vx     vy     vyaw   ax     ay
  initial_state: [0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000]
  initial_sigma: [0.100, 0.100, 0.100, 0.100, 0.100, 0.100, 0.100, 0.100]

odometry_sensor:  # Odometry sensor-specific parameters
  topic: 'odom'
  twist_target_frame: 'base_link'
  linear_velocity_dimensions: ['x', 'y']  # Replaces the boolean vector that r_l uses
  angular_velocity_dimensions: ['yaw']    #

imu_sensor:       # IMU sensor-specific parameters
  topic: 'imu'
  twist_target_frame: 'base_link'
  angular_velocity_dimensions: ['yaw']    # Replaces the boolean vector that r_l uses
```

Basic Configuration for the `fixed_lag_smoother_node`

```
...

publishers:
  filtered_publisher:
    type: fuse_models::Odometry2DPublisher

filtered_publisher:                # Identical concepts to the way r_l manages these
  topic: 'odom_filtered'
  base_link_frame_id: 'base_link'
  odom_frame_id: 'odom'
  map_frame_id: 'map'
  world_frame_id: 'odom'
  publish_tf: true
  publish_frequency: 10
```

Task 6: Back to the Planar

In this task, we will revisit the first planar robot task that we carried out with `robot_localization`, but we will now use the `fixed_lag_smoother_node` in `fuse`.

- Reminder: bag data is stored here: `$bags/planar/planar.db3`.
- We will go through the same progression of adding sensor data as we progress

Task 6a: Odometry Only

1. Edit the file `$task6/config/odometry.yaml`
   - We want to make our first odometry (as in `nav_msgs/Odometry`) input our wheel encoder odometry.
   - For this exercise, we will fuse the `x` velocity, the `y` velocity, and the `yaw` velocity from the wheel encoders.
2. Run the filter and `rviz2` with:
   Terminal 1: `ros2 launch task6 fls.launch.xml`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
   - For comparison, we show the raw wheel encoder data and EKF output alongside the `fixed_lag_smoother` output.
3. The bag starts and ends at the same location. As in Task 1, use the `rviz2` measurement tool to measure the distance from the robot's first pose to its last. Make a mental note of the value!

Task 6b: Odometry + IMU

1. Edit `$task6/config/odometry_imu.yaml`
    - We will now be adding IMU sensor data to our smoother
    - The wheel encoder odometry configuration has been provided for you
    - You need to now fill out the configuration for the IMU topic. We want to fuse `yaw` velocity and `x` acceleration from the sensor.
    - R2's holographic projector is bulky and made mounting the IMU difficult, so his designers mounted the IMU such that `+X` points to the ground, `+Y` points to R2's right, and `+Z` points towards his back.
        i. This will have ramifications for the sensor configuration!
2. Run the filter and `rviz2` with:
   Terminal 1: `ros2 launch task6 fls.launch.xml include_imu:=True`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
    - The launch file runs two instances: one has our previous odometry-only config, and one has odometry + IMU. Raw wheel encoder data is also displayed, as is the EKF output for the same config

Task 6c: Odometry + IMU + VO

1. Edit `$task6/config/odometry_imu_vo.yaml`
    a. We will now add visual odometry data as an input to the smoother
    b. The wheel encoder odometry and IMU configurations have been provided for you
    c. As with wheel encoder odometry, we want to fuse `x`, `y`, and `yaw` velocities into the filter
2. Run the smoother and `rviz2` with:
    Terminal 1: `ros2 launch task6 fls.launch.xml include_imu_vo:=True`
    Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
    a. We now have three Fixed Lag Smoother instances running:
        i. One with just wheel encoder data
        ii. One with wheel encoder + IMU data
        iii. One with wheel encoder, IMU, and visual odometry data
        iv. We are also still displaying the raw wheel encoder data
        v. We are also running the EKF with the same “all sensors” configuration
    b. Run `top` and compare the EKF and “all sensors” Fixed Lag Smoother instances

Task 7: Déjà Two (-tier Setup)

1. Edit the file `$task7/config/two_tier.yaml`
    - The config for the `odom`->`base_link` instance has been provided for you.
    - Add parameters for a second node to the same config file. The node's name is `fls_node_tier2`.
    - The `fls_node_tier2` should have a `world_frame` of `map`.
    - It should have the exact same inputs as the `fls_node_tier1`
    - It should also have a new input for a topic called `pose_global`. That topic contains poses in the map frame that provide an absolute reference for the filter. We want to fuse `x`, `y`, and `yaw` from this source.
    - We want the filter to trust the pose data, but not absolutely. Tune your kinematic model's `process_noise_diagonal` accordingly.
2. After editing the config, run the following:
   Terminal 1: `ros2 launch task7 fls.launch.xml`
   Terminal 2: `ros2 bag play $bags/planar/planar.db3 --clock`
3. Note the difference with the EKF output

Writing a `fuse` Plugin

fuse supports plugins for all of its key object types. We'll focus on sensor models.

Our sensor model will be that of a beacon range sensor. The sensor wirelessly receives messages from beacons placed at regular intervals throughout the warehouse environment that we've already seen.

In this case, we need to develop three classes:

1. `SensorModel:` we need a class that is derived from the `SensorModel` base class in `fuse_core`. The job of this class is to receive sensor data and create an instance of a…
2. `Constraint`: we will derive a class whose instances will be added to the actual factor graph. The constraint math itself will be wrapped in a…
3. `CostFunctor`: This is not actually a base class, but a class that must wrap a `()` operator. Ceres uses this method to compute both the residual and, if using auto differentiation, Jacobian matrices.

Example Sensor Model Header

```
class MySensorModel : public fuse_core::AyncSensorModel
{
...

protected:

// Must override. This is where the sensor reads parameters, subscribes to non-sensor topics, etc.
void onInit() override;

// You will likely be receiving sensor data. Within this method, you will likely create transactions
// with constraints on variables and relevant time stamps, and then call sendTransaction().
void dataCallback(const whatever_msgs::msg::SensorMessage & message);

// Can optionally implement this method if your sensor model requires variable values from the graph.
void onGraphUpdate(Graph::ConstSharedPtr graph) override;

// This is where we typically subscribe to sensor data
void onStart() override;

// This is where we typically unsubscribe from sensor data
void onStop() override;
```

[fuse_core::AsyncSensorModel]

Example Constraint Header

```
class MyConstraint : public fuse_core::Constraint
{
public:
  // Your constraint will involve one or more variables, along with measurements (or priors) of their values
  MyConstraint(const std::string & source,
               const fuse_variables::Pose2DStamped & robot_pose,
               const Eigen::Vector3d& measurement_mean,
               const Eigen::Matrix3d& measurement_covariance);

  // Useful for debugging
  void print(std::ostream & stream = std::cout) const override;

  // This method must be overridden, and must return a pointer to a Ceres cost function. The returned object is what
  // computes the actual residual.
  ceres::CostFunction * costFunction() const override;

private:
  fuse_core::Vector3d measurement_mean_;  // The measured/prior mean vector for this variable
  fuse_core::Matrix3d measurement_sqrt_information_;  // The square root information matrix
```

[fuse_core::Constraint]

Example Ceres Cost Function

```
class MyCostFunctor
{
public:
  // Your cost functor receives any values it will need to compute the residual
  MyCostFunctor(const Eigen::Vector3d& measurement_mean, cost Eigen::Matrix3d& measurement_sqrt_information_)
  : measurement_mean_(measurement_mean), measurement_sqrt_information_(measurement_sqrt_information_)

  template<typename T>
  bool operator()(const T * const robot_pose, T * residuals) const  // Sizes of these arrays are defined in the constraint
  {
    residuals[0] = measurement_mean_[0] - robot_pose[0];
    ...

    // Map it to Eigen, and weight it
    Eigen::Map<Eigen::Matrix<T, 3, 1>> residual_map(residual);
    residual_map.applyOnTheLeft(measurement_sqrt_information_.template cast<T>());
  }
```

Example

Task 8: Your First Sensor Model

1. The code files that we need to modify have already been generated. Take some time to review them:
   - `$task8/include/beacon_sensor_model.hpp` and `$task8/src/beacon_sensor_model.cpp`
   - `$task8/include/range_constraint.hpp` and `$task8/src/range_constraint.cpp`
   - `$task8/include/range_cost_functor.hpp`
2. Analyse the sensor message type we will be using:
   - `ros2 interface show workshop_msgs/msg/BeaconRangeArray.msg`
3. Edit `$task8/include/beacon_sensor_model.hpp`
   - A subscriber and callback method have been created, but need the message type to be added.
4. Edit `$task8/src/beacon_sensor_model.cpp`
   - Edit the creation of the subscriber on line 45 by adding the correct type
   - Add the correct variable to the call on line 87
   - Add the correct container for iteration on line 97
   - Note the call on line 99!

Task 8: Your First Sensor Model

1. Edit `$task8/include/range_constraint.hpp`
   - Add the correct variable type on line 74
2. Edit `$task8/src/range_constraint.cpp`
   - Add the correct variable type on line 29
   - Add the correct template parameter values on line 62 (see the comments above)
3. Edit `$task8/include/range_cost_functor.hpp`
   - On line 98 and 99, add the correct values so that we are computing the difference between the robot's position variable components and the beacon's reported position
4. Run `cb` to build the workspace (recall that this will handle directory changes before and after building)
5. Run `s` to source the workspace (in terminal 1)
6. Edit the file `$task8/config/two_tier.yaml`
7. Add the beacon sensor to your configuration
8. Run:
   Terminal 1: `ros2 launch task8 fls.launch.xml`
   Terminal 2: `ros2 bag play $bags/planar.db3 --clock`

ROSCon UK '25
EDINBURGH

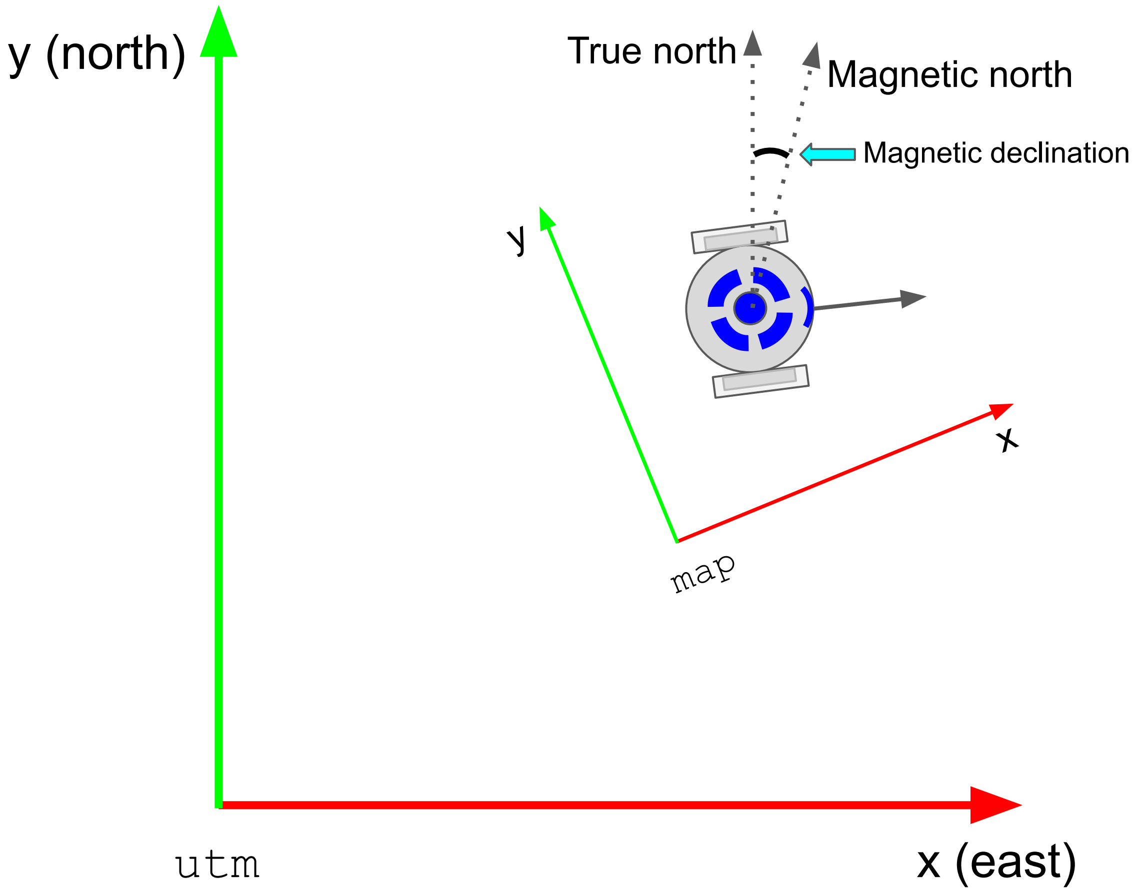
Additional Task 1: GPS Data in 3D

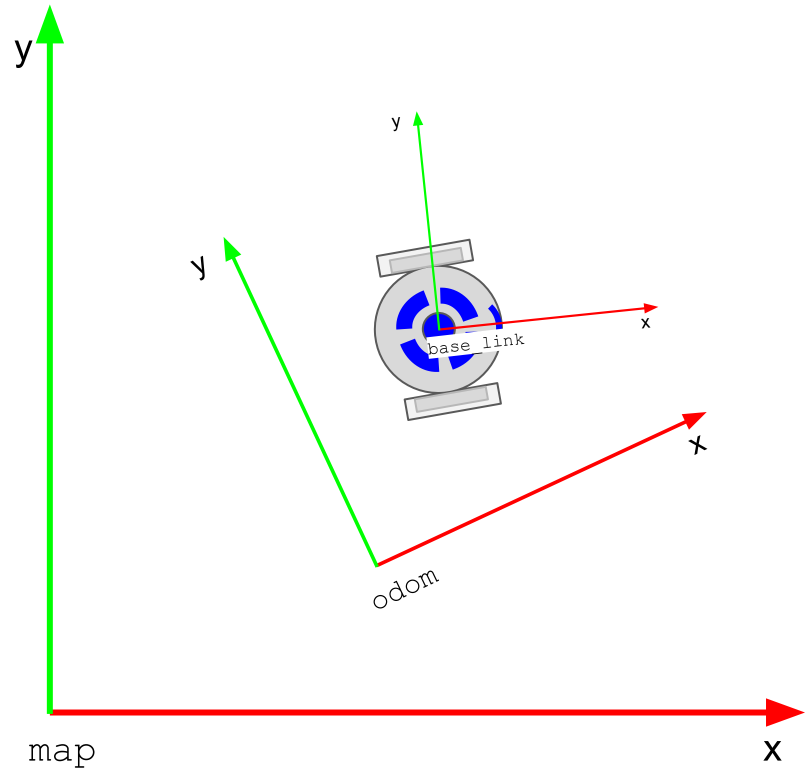
1. Change the Task 4 configuration so that we are fusing 3D dimensions as well

Additional Task 2: Drone Bag

R2-D2 can fly! But he'll need your help.

1. The bag you need is in the bags directory, under `aerial_3d`.
2. No assistance on this one! Analyse the bag, create a config and launch file (feel free to just use an existing task package), and see how it goes!
   a. Note: if you want to use the R2 model, your body frame will need to be `r2d2/base_link`.


y (north)
True north
Magnetic north
Magnetic declination
y
x
map
utm
x (east)


y
y
y
x
base_link
x
odom
map
x