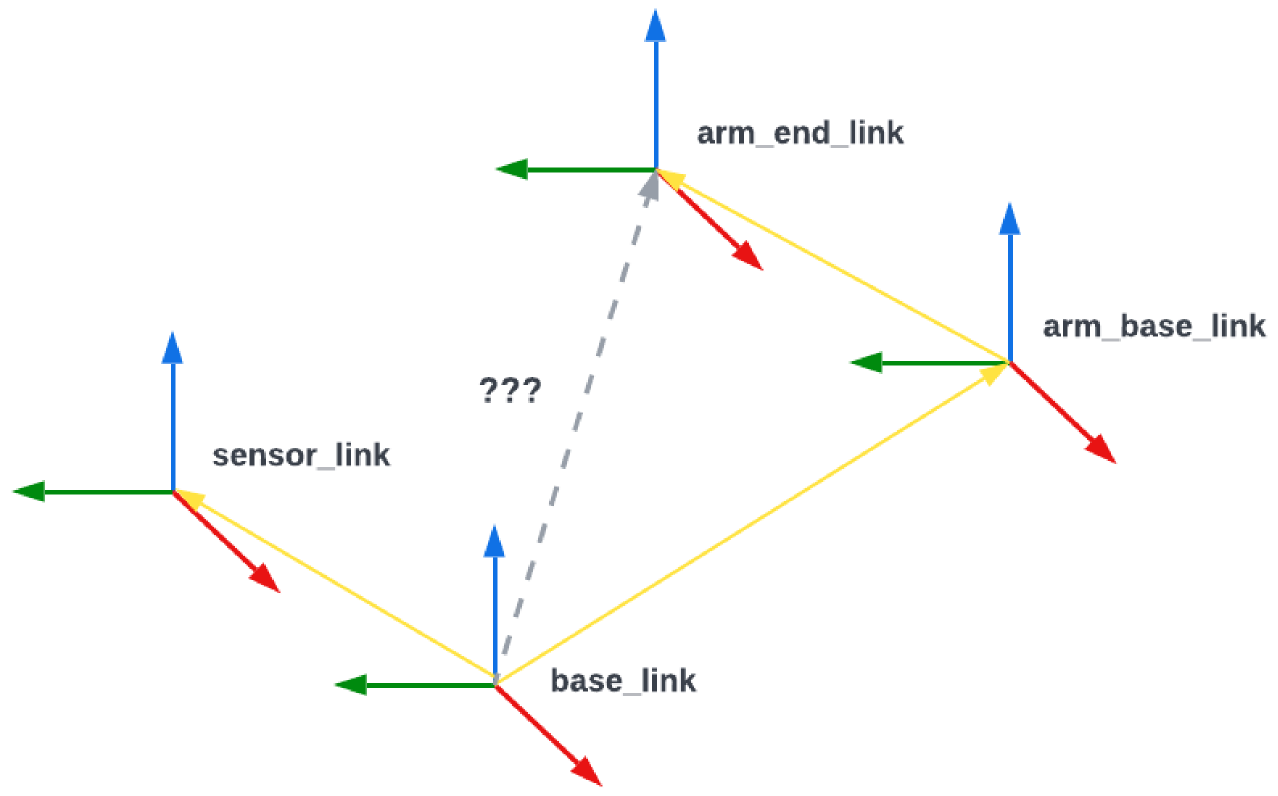
Understanding ROS Transforms

Defining how objects in a robot's world relate to each other

Esther Weon · José L. Millán · 5 min read
Published December 21, 2022



Robots are compound systems navigating complex worlds. To understand how they interact with other players in their environments, we can use mathematical operations called transformations. **Transformations (or more simply, "transforms") express an object's position and orientation in relation to another.**

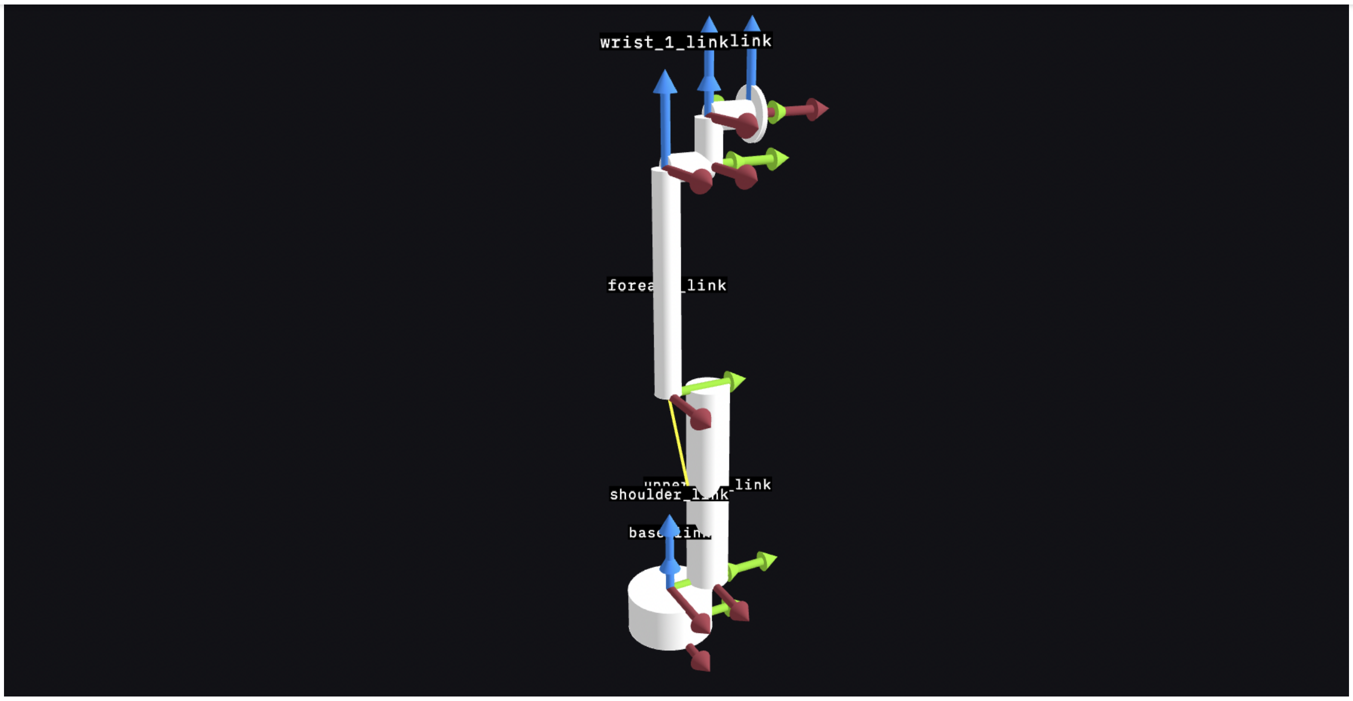
navigate – not only by positioning their components in relation to each other, but also in relation to the objects they encounter.

Situating objects with frames

In robotics, **a frame refers to the coordinate system describing an object's position and orientation, typically along x, y, and z axes**. ROS requires each frame to have its own unique `frame_id`, and a typical scene consists of multiple frames for each robot component (i.e. limb, sensor), detected object, and player in the robot's world.

A scene always has one **base frame** – usually named `world` or `map` – that is an unmoving constant. All other frames in the scene – including the robot's own frame, which ROS typically calls `base_link` – are children to this base frame, and are positioned relative to this parent in some way.

The robot's `base_link` frame usually starts at its base and branches off into child frames for limbs and sensors:



Visualizing the UR5 collaborative robot arm in Foxglove Studio's 3D panel. Each frame's x, y, and z axes are denoted by red, blue, and green arrows.

Each sensor's frames can also branch off into further child frames for the objects they detect. In this way, all objects in a given scene are somehow defined in relation to the base frame and its child frames.

Connecting frames with transforms

Transforms define the translations and rotations needed to get from a source frame to a target frame – whether it's parent-to-child, child-to-parent, or across multiple "generations" of frames. A complete set of a scene's transforms, from the base frame to all its related children, constitutes a **transform tree**.

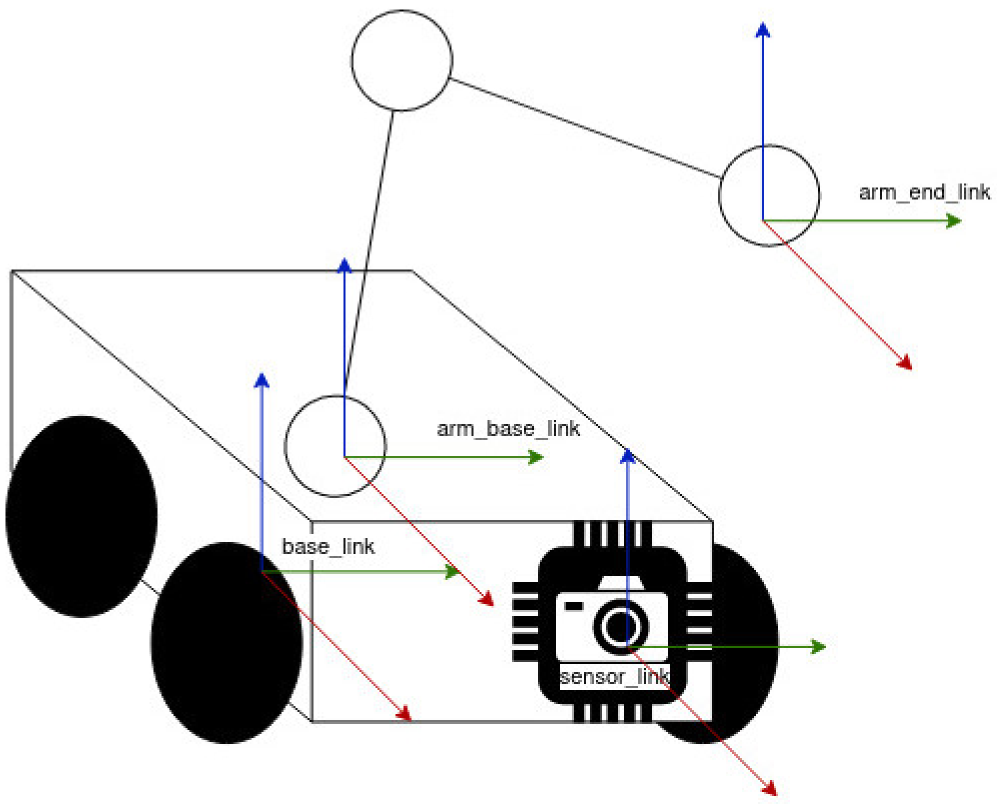
With a transform tree, you can quickly find the position and orientation of any given frame in the scene, no matter how many levels removed you are from the base. Transforms can be either static or mobile – **static transforms** (e.g.

transforms (e.g. between `base_link` and either a robotic arm mounted to the base or a separate detected object) can change as the world does.

In ROS, [transform messages]() are broadcast on two topics – `/tf_static` (for static transforms) and `/tf` (for mobile transforms). This separation improves bandwidth and reduces the number of transforms being published. ROS also provides the [`tf2`]() library to help us calculate transformations between frames.

In the real world

Let's imagine a rover with a camera and robotic arm, navigating a flat maze:



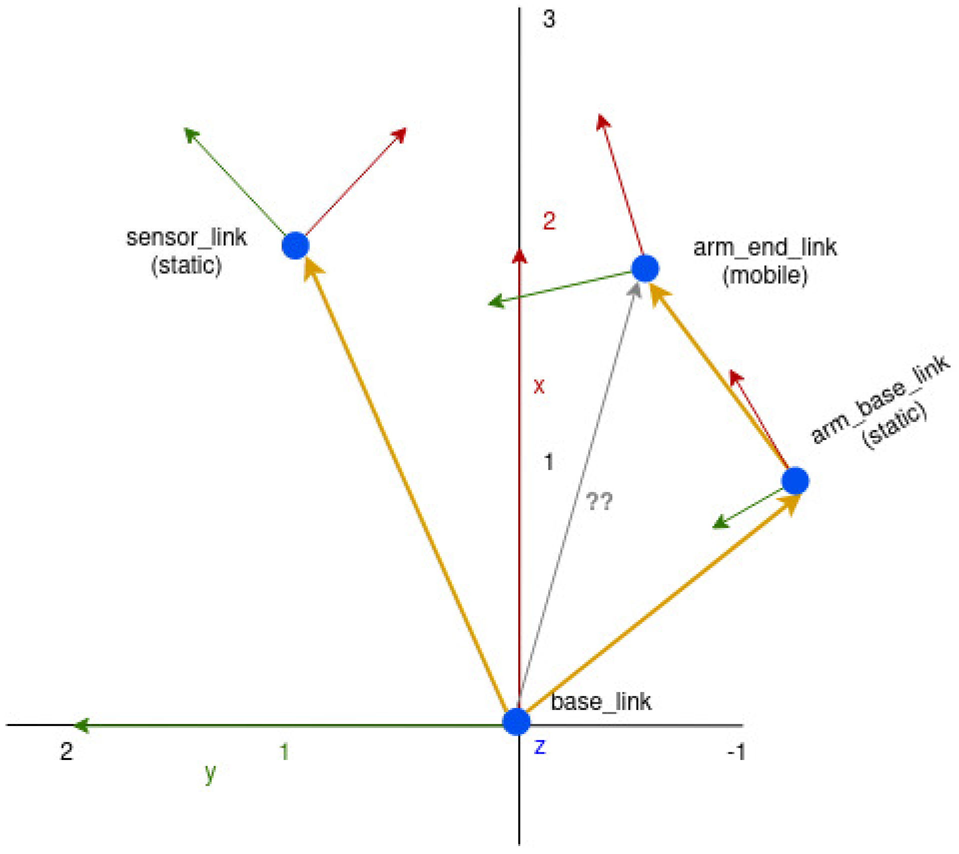
We can see that the rover's `base_link` frame has static transforms to two child frames – a `sensor_link` frame representing the unmoving camera mounted to the front, and an `arm_base_link` frame representing the unmoving arm base mounted to the top.

Building the transform tree

While ROS's `tf2` library provides an easy way for us to calculate transforms, let's take a peek at what this library is actually doing under the hood.

Each child frame has a transform that represents its position vector and rotation in relation to its parent frame:

- **Parent:** `base_link`
  - `sensor_link – (2,1)` position, `-45°` (`π/4`) rotation
  - `arm_base_link – (1,-1)` position, `30°` (`π/6`) rotation
- **Parent:** `arm_base_link`
  - `arm_end_link – (1,1)` position, no rotation



arm_end_link's position in the base_link frame.

Calculating an object's position using transforms

To start traversing our transform tree (from `base_link` to `arm_base_link` to `arm_end_link`), we need two pieces of information:

- `T` - 2D transform matrix between `base_link` and `arm_base_link` (translation of `dx` over `x` and `dy` over `y`, rotation over `z`)
- `p` – Position vector for `arm_end_link` in the `arm_base_link` frame

$$T = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & d_x \\ \sin(\phi) & \cos(\phi) & d_y \\ 0 & 0 & 1 \end{bmatrix} \qquad \vec{p} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Let's set the following values in `T` matrix:

- `Φ = π/6` – Rotation of `arm_base_link` in `base_link`
- `dx = 1` – Position (x) of `arm_base_link` in `base_link`
- `dy = -1` – Position (y) of `arm_base_link` in `base_link`

And the following values in the `p` matrix:

- `x = 1` – Position (x) of `arm_end_link` in `arm_base_link`
- `y = 1` – Position (y) of `arm_end_link` in `arm_base_link`

Finally, let's multiple these matrices together to get `arm_end_link`'s position in the `base_link` frame:

$$p_{base}^{arm\,end} = T_{base}^{arm\,base} \cdot p_{arm\,base}^{arm\,end} = \begin{bmatrix} \sin\left(\frac{\pi}{6}\right) & \cos\left(\frac{\pi}{6}\right) & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sin\left(\frac{\pi}{6}\right) + \cos\left(\frac{\pi}{6}\right) - 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.366 \\ 1 \end{bmatrix}$$

With our operation, we find that `arm_end_link` is in position `(1.366, 0,366)` of the `base_link` frame!

Since `arm_end_link` is a mobile frame, it's important to note that this result is only valid at the robotic arm's current position and angle. As the transform between `arm_base_link` and `arm_end_link` inevitably changes, so will `arm_end_link`'s position in space.

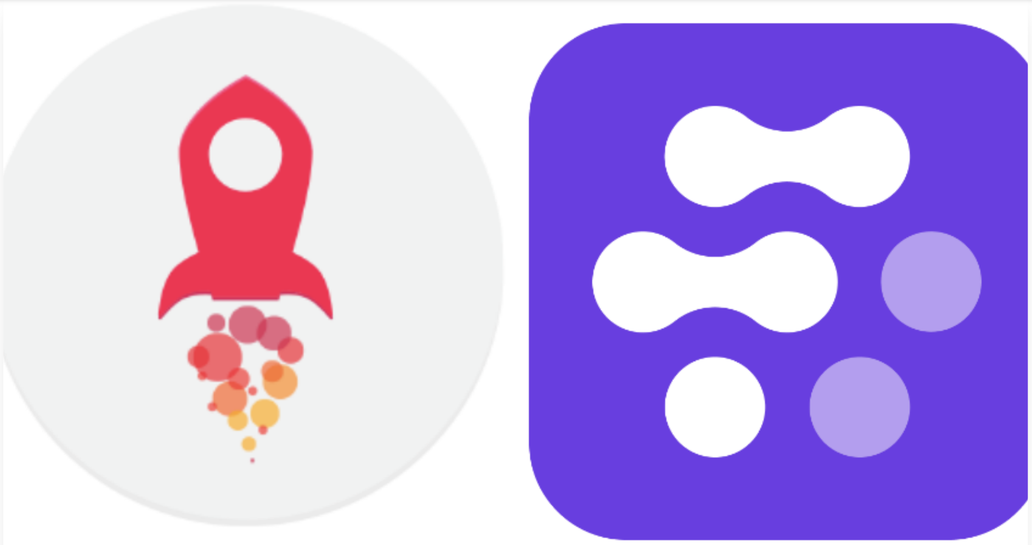
Learn more

Actual robots in the real world are often much more complex than our single-sensor example, and must navigate much trickier terrain than a flat maze. Calculating an object's position in its environment will often require multiplying more than two matrices to travel from a source frame to a target frame.

Fortunately for us, ROS provides the `tf2` library to do the necessary transformation matrix math for us, so we can avoid doing these manual calculations ourselves!

Stay tuned for our next tutorial on how to use these libraries to publish and view transforms in Foxglove Studio. As always, feel free to ask questions, share feedback, and request a topic for the next blog post in our Slack community!

Share on Twitter Share on LinkedIn



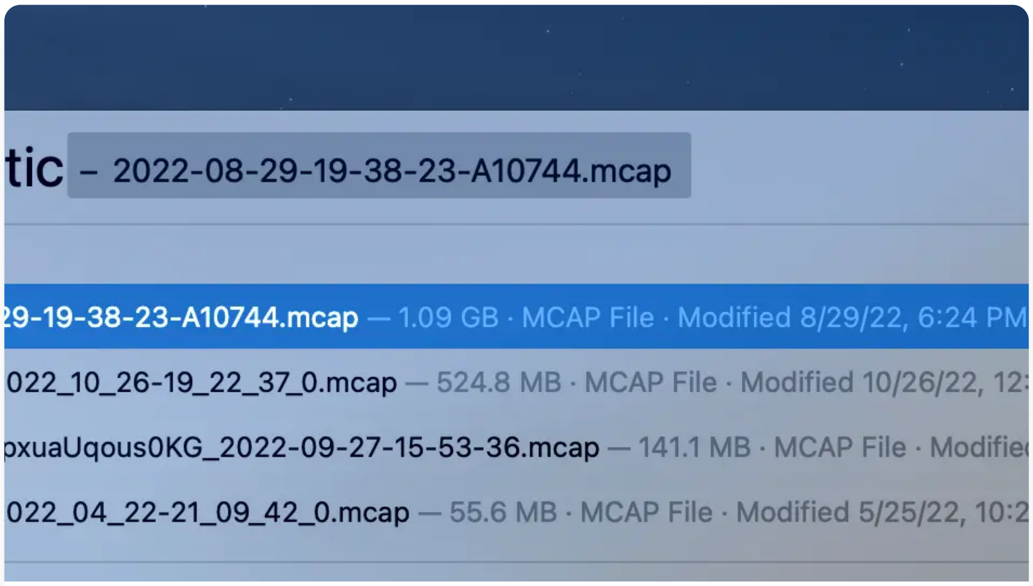
tutorial studio data platform MCAP

Announcing FlatBuffers Support in Foxglove

Analyze your FlatBuffers data with Foxglove Studio and Data Platform.

Sam Nosenzo

December 12, 2022 · 6 min read



tic – 2022-08-29-19-38-23-A10744.mcap

29-19-38-23-A10744.mcap — 1.09 GB · MCAP File · Modified 8/29/22, 6:24 PM

022_10_26-19_22_37_0.mcap — 524.8 MB · MCAP File · Modified 10/26/22, 12:

pxuaUqous0KG_2022-09-27-15-53-36.mcap — 141.1 MB · MCAP File · Modifie

022_04_22-21_09_42_0.mcap — 55.6 MB · MCAP File · Modified 5/25/22, 10:2

article studio MCAP

Robotics Data

How we built a Spotlight Importer for MCAP files using Swift.

Jacob Bandes-Storch

December 7, 2022 · 18 min read

All blog posts

Get blog posts sent directly to your inbox.

Email

Subscribe

Ready to get started?

Download today on Linux, Windows, or macOS.

Try it out

Download app

PRODUCTS	GET IN TOUCH	COMPANY
Foxglove Studio	Join our community	About
Foxglove Data Platform	Schedule a demo	Blog