Coordinate Transforms and the TfListener

Wyatt Newman
February, 2015

Defining coordinate frames: A "frame" is defined by a point in 3-D space, **p**, that is the frame's origin, and three vectors: **n**, **t** and **b** (which define the local x, y and z axes, respectively). The axis vectors are normalized (have unit length), and they form a right-hand triad, such that **n** crossed into **t** equals **b:**

$\boldsymbol{b}=\boldsymbol{n}\times\boldsymbol{t}$ These three directional axes can be stacked side-by-side as column vectors, comprising a 3x3 matrix, **R**.

$$\boldsymbol{R}=[\boldsymbol{n}\,\boldsymbol{t}\,\boldsymbol{b}]=\begin{bmatrix} n_x & t_x & b_x \\ n_y & t_y & b_y \\ n_z & t_z & b_z \end{bmatrix}$$

We can include the origin vector as well to define a 3x4 matrix as:

$$\boldsymbol{A}=[\boldsymbol{n}\,\boldsymbol{t}\,\boldsymbol{b}\,\boldsymbol{p}]=\begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \end{bmatrix}$$

A useful trick to simplify mathematical operations is to define an augmented matrix, converting the above 3x4 matrix in to a square 4x4 matrix by adding a fourth row consisting of [0 0 0 1]. This augmented matrix is a 4x4, which we will refer to as a "**T**" matrix:

$$\boldsymbol{T}=\begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Conveniently, matrices constructed as above (consistent with valid frame specifications) are always invertible. Further, computation of the inverse of a **T** matrix is efficient.

Abstractly, one can refer to an origin and a set of orientation axes (vectors) without having to specify numerical values. However, to perform computations, numerical values are required—and this requires further definitions. Specifically, when numerical values are given, one must define the coordinate system in which the values are measured. For example, to specify the origin of frame "B" (i.e. point "**p**") with respect to frame "A", we can measure p_x along the x axis of frame "A", measure p_y along the y axis of frame "A", and p_z along the z axis of frame A. These can be referred to explicitly as $p_{x/A}$, $p_{y/A}$ and $p_{z/A}$, respectively. If we had provided coordinates for the origin of frame B from any other viewpoint, the numerical values of the components of **p** would be different.

Similarly, we can describe the components of frame-B's **n** axis (similarly, **t** and **b** axes) with respect to frame A by measuring the x, y and z components along the respective axes of frame A. We can then state the position and orientation of frame B with respect to frame A as:

$$ {}^{A}\boldsymbol{T}_{B} = \boldsymbol{T}_{B/A} = \begin{bmatrix} n_{x/A} & t_{x/A} & b_{x/A} & p_{x/A} \\ n_{y/A} & t_{y/A} & b_{y/A} & p_{y/A} \\ n_{z/A} & t_{z/A} & b_{z/A} & p_{z/A} \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

We can refer to the above matrix as “frame B with respect to frame A.”

Having labeled frames “A” and “B”, providing values for the elements of $ {}^{A}\boldsymbol{T}_{B} $ fully specifies the position and orientation of frame B with respect to frame A.

Coordinate-frame transformations: In addition to providing a means to explicitly declare the position and orientation of a frame (with respect to some named frame, e.g. frame B with respect to frame A), **T** matrices can also be interpreted as operators. For example, if we know the position and orientation of frame C with respect to frame B, i.e. $ {}^{B}\boldsymbol{T}_{C} $, and if we also know the position and orientation of frame B with respect to frame A, $ {}^{A}\boldsymbol{T}_{B} $, then we can compute the position and orientation of frame C with respect to frame A as follows:

$$ {}^{A}\boldsymbol{T}_{c} = {}^{A}\boldsymbol{T}_{B}\,{}^{B}\boldsymbol{T}_{C} $$

That is, a simple matrix multiplication yields the desired transform. This process can be extended, e.g.:

$$ {}^{A}\boldsymbol{T}_{F} = {}^{A}\boldsymbol{T}_{B}\,{}^{B}\boldsymbol{T}_{C}\,{}^{C}\boldsymbol{T}_{D}\,{}^{D}\boldsymbol{T}_{E}\,{}^{E}\boldsymbol{T}_{F} $$

In the above, the 4x4 on the left-hand side can be interpreted column by column. For example, the fourth column (rows 1 through 3), are the coordinates of the origin of frame “F” as measured with respect to frame “A.”

By using the notation of prefix superscripts and post subscripts, there is a visual mnemonic to aid logical compatibility. The super and sub-scripts act like “Lego” blocks, such that a subscript of a leading **T** matrix must match the pre-superscript of the trailing **T** matrix. Following this convention helps to keep transform operations consistent.

Transforms in ROS: Coordinate-frame transformations are ubiquitous in robotics. For articulated robot arms, full 6-DOF transforms are required to compute gripper poses as a function of joint angles. Multiplication of sequential transforms, link by link, performs such computations. Sensor data, e.g. from cameras, or LIDARs, is acquired in terms of the sensor's own frame, and this data must be interpreted in terms of alternative frames (e.g. world frame or robot frame).

Since coordinate transforms are so common in robotics, ROS has provided a powerful package—the “tf” package (see http://wiki.ros.org/tf)-- for handling transforms. Use of this package can be confusing, because “tf” performs considerable work.

Within ROS, the topic “/tf” carries messages of type tf2_msgs/TFMessage. Running:

```
rosmsg show tf2_msgs/TFMessage
```

reveals that the tf2_msgs/TFMessage is organized as follows:

```
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
```

```
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

That is, this message contains a vector (variable-length array) of messages of type geometry_msgs/TransformStamped. Below, we examine these transforms for the minimal robot.

There can be (and typically are) many publishers to the "tf" topic. Each publisher expresses a transform relationship, describing a named "child" frame with respect to a named "parent" frame. The ROS transform datatype is not identical to a 4x4 homogeneous transformation matrix, but it carries equivalent information (and more). The ROS transform datatype contains a 3-D vector (equivalent to the 4th column of a 4x4 transform) and a quaternion (an alternative representation of orientation). In addition, transform messages have time stamps, and they explicitly name the child frame and the parent frame (as text strings). A history of prior transforms (with respective time stamps) is also retained (with a limited history).

As an example, one can launch the minimal robot with:

```
roslaunch minimal_robot_description minimal_robot.launch
```

Running: "rostopic list" reveals that there is a topic called /tf, and "rostopic info tf" shows that this topic has three nodes that publish to it: robot_state_publisher, kinect_calib and kinect_calib2. Running "rostopic echo tf" results in a rapid stream of output to the terminal. The display includes the following:

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 27
        nsecs: 177000000
      frame_id: world
    child_frame_id: link1
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
```

```
      y: 0.0
      z: 0.0
      w: 1.0
```
which declares how link1 is related to the world frame. Here we see that the origins are coincident, and the respective axes are also aligned (since the quaternion is [0,0,0,1], corresponding to zero relative rotation). This follows from our robot-description URDF from the lines:

```
 <link name="world"/>
 <joint name="glue_robot_to_world" type="fixed">
  <parent link="world"/>
  <child link="link1"/>
 </joint>
```

In the above, a fixed relationship between link1 and the world is declared. Since no values of xyz nor rpy are declared, these are implicitly set to zero, which declares that the coordinate frames of world and link1 are precisely coincident (have the same origin and the same axes).

Another message on tf shows the relationship between the “kinect_link” (the body of our Kinect sensor) and link2:

```
    seq: 0
    stamp:
      secs: 27
      nsecs: 138000000
    frame_id: link2
  child_frame_id: kinect_link
  transform:
    translation:
      x: 0.2
      y: 0.0
      z: 0.95
    rotation:
      x: 0.0
      y: 0.564642473395
      z: 0.0
      w: 0.82533561491
```

This information is identical to the (fixed) spatial relationship declared in our URDF file, minimal_robot_w_sensor.urdf, per the following lines:

```
 <joint name="kinect_joint" type="fixed">
    <axis xyz="0 1 0" />
    <origin xyz="0.2 0.0 0.95" rpy="0.0 1.2 0"/>
    <parent link="link2"/>
    <child link="kinect_link"/>
 </joint>
```

What is not obvious is that the roll-pitch-yaw values and the specified axis are equivalent to the relative rotation expressed by the quaternion (0,0.565, 0, 0.8253). Nonetheless, the transform being published is entirely equivalent to the transform expressed in the URDF.

A third transform message expresses the relationship of link2 to link1. This transform is variable, depending on the angle of joint1. On start-up, tf shows the following relationship:

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 26
        nsecs: 658000000
      frame_id: link1
    child_frame_id: link2
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 1.0
      rotation:
        x: 0.0
        y: 0.023434555814
        z: 0.0
        w: 0.999725373087
```

This shows that link2's frame origin is located at (0,0,1) with respect to link1's coordinate frame, i.e. the origin of link2 is located 1m up (in the z direction), directly above the origin of link1. This is consistent with the robot-description URDF, per the lines:

```
  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="0 0 1" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
```

In the above, we see that the origin of link2 is at xyz = “0 0 1”. The axis of rotation of link2 is parallel to the link1 y-axis, and the orientation of link2 relative to link1 will change as the joint1 angle changes.

In the tf message relating link2 to link1, the quaternion is (0,0.02343, 0, 0.9997), which corresponds to a slight rotation about the y axis. This is due to gravity acting on the Kinect object, which is offset from link2, thus producing a torque about joint1, which results in some “gravity droop” about this joint.

The transform between link2 and link1 is continuously updated by the node “robot_state_publisher.” This node consults the robot URDF (via the parameter server) and subscribes to the topic “joint_state”. The “joint_state” topic is published by the minimal_joint_controller node, which updates the angle of joint2 at every iteration of the joint controller. The robot_state_publisher combines the URDF information and the joint_state information to continuously recompute and republish the link2/link1 transform.

A fourth transform published on the “tf” topic relates the sensor frame, kinect_pc_frame, to the sensor body, kinect_link:

```
transforms:
```

```
-
  header:
    seq: 0
    stamp:
      secs: 26
      nsecs: 666000000
    frame_id: kinect_link
  child_frame_id: kinect_pc_frame
  transform:
    translation:
      x: 0.0
      y: 0.0
      z: 0.0
    rotation:
      x: 0.5
      y: -0.5
      z: 0.5
      w: -0.5
```

This shows that the two frames have their origins coincident, since the x,y,z offsets are all zero. The quaternion rotation is (0.5,-0.5,0.5,-0.5), which describes the orientation of the sensor frame relative to the sensor body. This transform is published as described in the minimal_robot launch file by the line:

<node pkg="tf" type="static_transform_publisher" name="kinect_calib2" args="0 0 0 -0.500 0.500 -0.500 0.500 kinect_link kinect_pc_frame 10"/>

This launch-file line starts a node “static_transform_publisher”, which is part of the package “tf.” The node, when launched, is known to the ROS system by the name “kinect_calib2”, and it publishes a transform describing the pose of the sensor relative to the sensor body. The transform is static, and it consists of an origin offset of [0,0,0] and a quaternion of [-0.5, 0,5, -0.5, 0.5]. These describe the offset and rotation parameters of the “kinect_pc_frame” relative to the parent frame “kinect_link”. Although this is a static relationship (the sensor frame should never move relative to the sensor housing), the ROS convention is to continue to publish (refresh) this transform on the topic “tf”.

A node that listens to the topic “tf” would be able to deduce the relationship of any frame to any other frame in our system. Since our transforms include: kinect_pc_frame relative to kinect_link; kinect_link relative to link2; link2 relative to link1; and link1 relative to world, we can combine this information to compute the position and orientation of the kinect_pc_frame relative to the world. This can be done by cascading transforms. This process is so common that a ROS tool already exists to perform this task: the “tf_listener.”

The tf_listener: A user node can instantiate a “transformListener” object (from the tf library), which is very useful for computing coordinate transforms. The transformListener subscribes to the tf topic, receives messages relating child and parent frames (with random arrival times) and assembles this information into a connected chain (using the most recently available relative transforms). By maintaining this chain, the transformListener is capable of responding to inquiries regarding coordinate-frame transforms (e.g., where is my right index fingertip with respect to my nose).

The package “example_robot_tf_listener” illustrates use of the tfListener. This package depends on the “tf” library (listed in the package.xml file). The source code includes the header file:

```
#include <tf/transform_listener.h>
```

In the main program, a tfListener is instantiated with the line:

```
tf::TransformListener tfListener; //create a TransformListener to listen for tf's and assemble them
```

Results of using the “lookup” method of this object are stored in the object:

```
tf::StampedTransform tf_sensor_frame_to_world_frame; //need objects of this type to hold tf's
```

As the tfListener runs, it continues to monitor the various tf messages and it is able to assemble them into a logical chain. However, when it first starts up, it does not have a full set of transforms, and thus an attempt to do a “lookup” can result in an error. The “try/catch” construct allows re-trying lookups until a complete chain is successfully found.

Once the tfListener is “warmed up”, it can be used to find the transform from any frame to any other frame. One such transform of particular interest is the transform from the sensor frame to the world frame, which may be used to transform all of the sensor data to world coordinates.

This transform is found with the line:

```
tfListener.lookupTransform("world", "kinect_pc_frame", ros::Time(0),
        tf_sensor_frame_to_world_frame);
```

The above function requests the transform that expresses the kinect_pc_frame with respect to the world frame, specifically the most recent transform available (specified by ros::Time(0)). The result is put in the object “tf_sensor_frame_to_world_frame.”

We can examine the data components of this object with member accessor functions, e.g. as follows:

```
origin= tf_sensor_frame_to_world_frame.getOrigin();
```

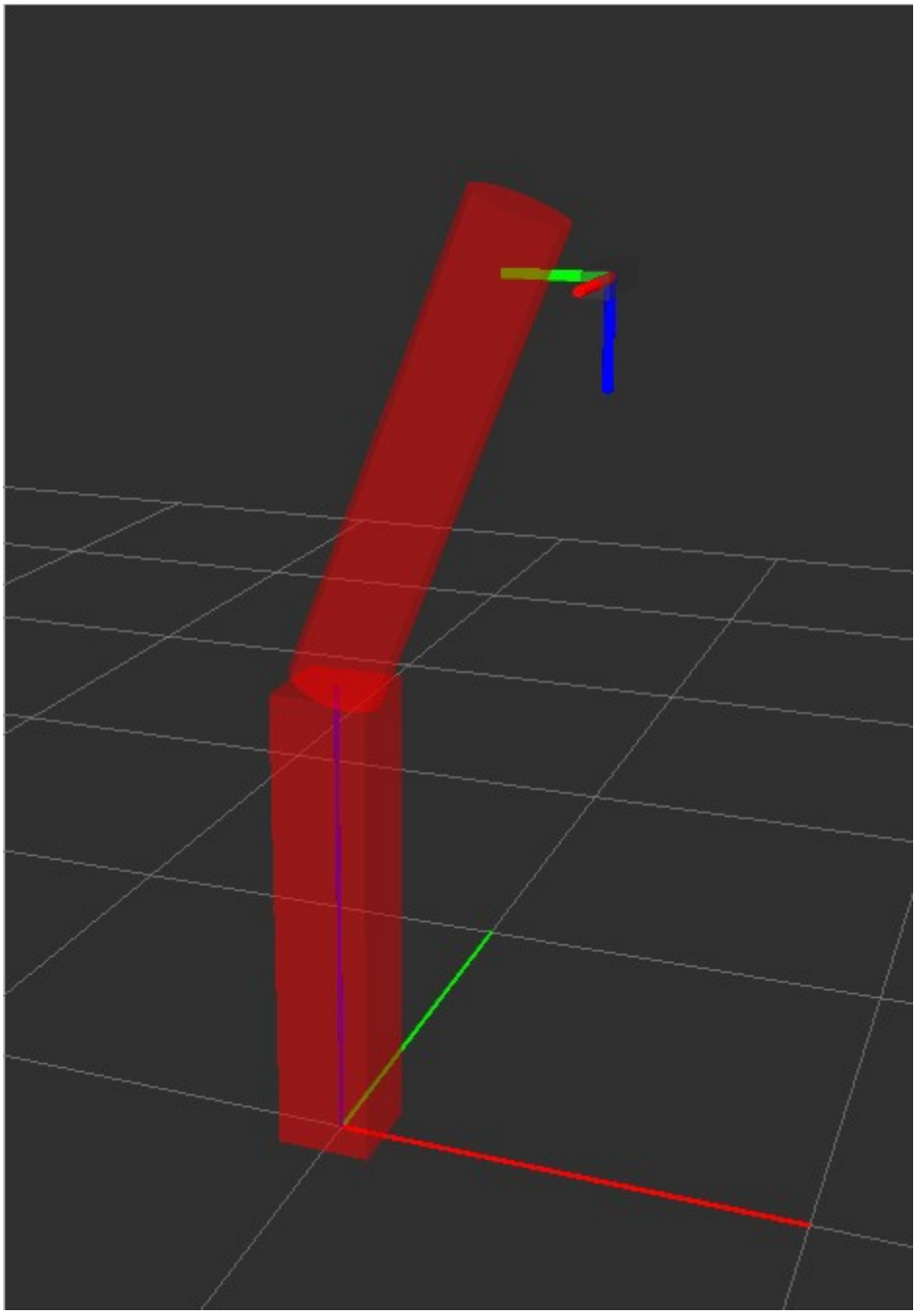
where “origin” is a compatible variable of type: tf::Vector3.

The orientation of the frame kinect_pc_frame with respect to the world frame can be extracted as:

```
R_orientation = tf_sensor_frame_to_world_frame.getBasis();
```

where R_orientation is a compatible variable of type tf::Matrix3x3.

The example program can be run with the minimal robot running. Commanding a slight tilt (0.14 rad) of joint 1 results in the robot posed as shown below. The world and sensor coordinate frames are shown.



An excerpt from the output of the example_robot_tf_listener node is:

[INFO] [1444188623.238188880, 1046.820000000]: origin: 0.531143, 0.000000, 1.812642
[INFO] [1444188623.238221973, 1046.820000000]: orientation matrix:
[INFO] [1444188623.238237849, 1046.820000000]: 0.000000 -1.000000 -0.000609
[INFO] [1444188623.238253102, 1046.820000000]: -1.000000 0.000000 0.000000
[INFO] [1444188623.238269887, 1046.820000000]: 0.000000 0.000609 -1.000000

This data can be interpreted from the figure shown. The origin of the sensor frame is nearly 2 meters high (1.813m), which appears feasible, given that each link is 1m in length. The origin is also

approximately 0.5m forward along the world x-axis (0.5311m). This also appears feasible, since the ground-plane grid is 1m squares. The y value of the sensor origin, with respect to the world, is 0, and this is reasonable, since the sensor is not shifted to the right or the left.

The commanded angle of the robot was deliberate, in order to get an easily-interpretable orientation matrix. The first column of the R matrix is [0; -1; 0], which means that the sensor x-axis (red axis) points along the world -y axis (green axis), and this appears true. The second column of the R matrix is approximately [-1; 0; 0]. This can be interpreted as: the sensor y (green) axis points in the negative x (red) world axis. This, too, appears true. Finally, the third column is approximately [0; 0; -1], which indicates that the sensor z-axis (blue axis) points in the negative direction of the world z-axis (blue axis), which is also true.

From this simple example, we can see that the tfListener did correctly compute the sensor frame, expressed in the world frame.

An object of type tf::StampedTransform has a tf::Transform as a member. If the tf::Transform is extracted, it can be used with a defined operation of "*". That is, one can multiply this object times 3-D data, and it will transform that data from the original frame to a target frame. This transform can be used to re-express sensor data in the world frame, which is important for computing robot motions for hand/eye coordination.

Although the transform class has many useful capabilities, its use in computing linear algebra operations is cumbersome. A more versatile linear-algebra library is "Eigen". By converting tf information to Eigen objects, one can perform many useful kinematic and model-fitting computations. Use of the Eigen library is introduced separately.