Robotics Prototyping Package Math and Algorithms

Ben Kolligs

Contents

1 Core

1.1 Transform

The main class of the core of the package is the Transform class. This class performs the functionality of a homogeneous rigid transform, which shows up in many places within robotics and robotics adjacent fields, like computer vision and graphics.

The homogeneous rigid transform is an extremely useful tool used to represent various things in robotics. First, the rigid transform is defined as a transformation T that when acting on any vector v, produces a transformed vector $T(v)$ of the form:

$$T(v) = Rv + t \tag{1}$$

where $R^T R^{-1}$, and t is a vector giving the translation of the origin. This concept can be represented in a form called a *homogeneous transformation matrix.* A homogeneous transformation matrix $T \in \mathbb{R}^{4\times 4}$ is a member of special euclidean group $SE(3)$, and can be written in the form:

$$T = \begin{bmatrix} R & t \\ \mathbf{0} & 1 \end{bmatrix} \tag{2}$$

where $R \in \mathbb{R}^{3\times 3}$ is a rotation matrix, and $t \in \mathbb{R}^3$ is a translation vector. The inverse of a homogeneous transformation matrix is:

$$T^{-1} = \begin{bmatrix} R^T & -R^T t \\ \mathbf{0} & 1 \end{bmatrix} \tag{3}$$

$$TT^{-1} = \mathbf{I} \tag{4}$$

When using super and subscripts to specify the frames we are referring to, then $T_b^a = (T_a^b)^{-1}$. We can use a homogeneous transformation to operate on homogeneous points $p, q \in \mathbb{R}^4$,

$$q = Tp \tag{5}$$

$$\begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}. \tag{6}$$

In general, there are three interpretations for homogeneous transformations:

1. A description of relative orientation and translation between frames.

2. A coordinate transform between frames. Specifically, T^i_j is a transform from frame j to frame i:
$$P^i = T^i_j P^j \tag{7}$$

3. A motion of a point (or a collection of points) within a single frame. For example, as we saw above point p can be moved to point q:
$$q = Tp \tag{8}$$
$$P^i_2 = T^i_j P^i_1 \tag{9}$$

   Similarly, a homogeneous transform can represent a motion from one frame to another. Specifically, T^i_j moves frame i to frame j.

It is important to specify which of these interpretations you are using within your program, as it can get confusing what these transformations represent if people aren't on the same page.

Additionally, one must pay attention to the frame that the operand is in, as this determines the function of the transform to some degree. $P^i_2 = T^i_j P^i_1$ is a motion that operates on point P^i_1 within frame i whereas $P^i = T^i_j P^j$ is a coordinate transformation from j to i, and therefore if we feed a point in frame j to this transformation, the point will not move, but the coordinates will be transformed into frame i, which is often desired.

1.2 Transforming between frames

We can chain together homogeneous transforms to represent a traversal through frames.

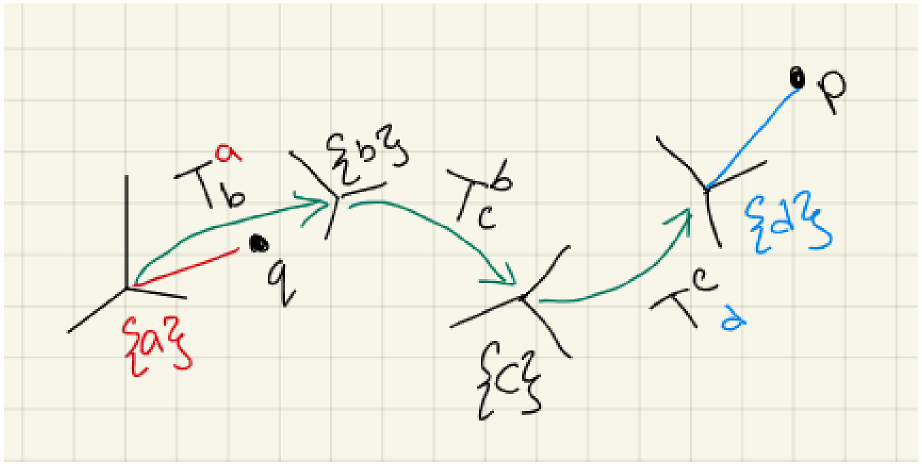


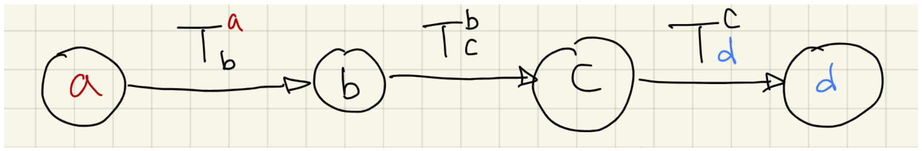
Figure 1: Traversing through different coordinate frames represented by rigid transformation matrices.

Say we want the transform from a to d:

$$T_d^a = T_b^a T_c^b T_d^c. \tag{10}$$

Notice that the bottom subscript and the following top subscript "cancel" out when multiplying frames together. Additionally when composing transforms like this, the "source" frame (in this case d) is included in the right most transform, and the "destination" frame a, is included in the left most.

Using this knowledge we can represent frames that are related in this way by a pose graph, or pose tree consisting of parent and child relationships between each frame.



$$\begin{tikzcd} a \arrow[r, "T_b^a"] & b \arrow[r, "T_c^b"] & c \arrow[r, "T_d^c"] & d \end{tikzcd}$$

Figure 2: Frames and their transforms can be represented in graph form.

2 Pointcloud

The pointcloud class contains a list of 3D points in space. When transforming the entire pointcloud, the class simply creates homogeneous points out of the point list, and then multiplies a given homogeneous transform by this array. So if we want to transform the coordinates of pointcloud C^i from frame i into frame j:

$$C^i = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \\ y_1 & y_2 & y_3 & \dots & y_n \\ z_1 & z_2 & z_3 & \dots & z_n \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix} \tag{11}$$

Then we need to multiply by the transform T_i^j:

$$C^j = T_i^j C^i. \tag{12}$$

3 Rigid Collection

The rigid collection class $\mathcal{T}$ is simply a set of transforms, $\mathcal{T} = \{T_1, T_2, T_3, \dots T_n\}$. It is required that this list of transforms be expressed in the same datum frame, that is:

$$\mathcal{T}^i = \{T_1^i, T_2^i, T_3^i, \dots, T_n^i\} \tag{13}$$

The rigid collection is able to be graphed in its ”parent” frame i, which allows all contained transforms to be visualized. The collection is able to add transforms to itself by simply appending new transforms to the set.

4 Kinematic Tree

The kinematic tree is an abstract representation of a collection of transforms. Unlike the rigid collection (section 3), the kinematic tree is able to represent transforms in multiple datum frames. The tree can then use the network of frames to ”lookup” a transform between any two connected nodes, as hinted at in figure 2. This concept is widely used in robotics and therefore has a place in the prototyping package. There are several algorithms this class implements:

1. Representation: Construct a tree representation given only the edges
2. Lookup: apply breadth first graph search to find a path between any two nodes on the tree
3. Root: use depth first search to express all frames in the base link frame or another specified frame on the tree

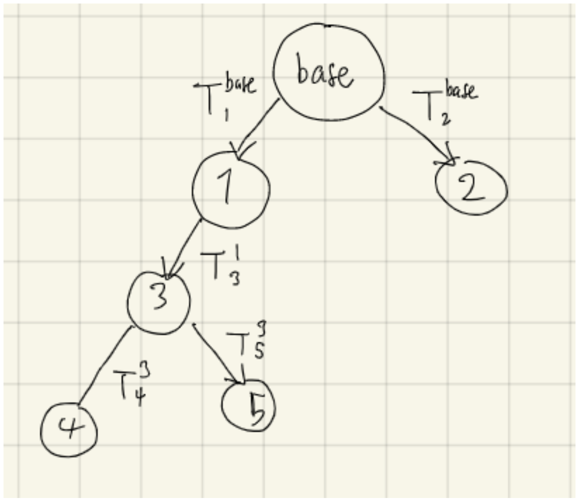


Figure 3: Example tree

4.1 Representation

Frames in a kinematic tree must have a child frame, a parent frame, or both. Frame T is described as:

$$T_i^{p(i)} = \begin{bmatrix} R_i^{p(i)} & t_i^{p(i)} \\ \mathbf{0} & 1 \end{bmatrix} \tag{14}$$

where i is the child frame, and $p(i)$ is the parent frame. The kinematic tree has a root, which we can also call the ”base link”. The challenge with the kinematic tree is that we are only given the transforms, which are the edges in the graph. So we need to create a tree representation from just the transforms. Transforms contain the parent and child of the edge, and are therefore directed. Thus we want a representation that is easily able to express the directed nature of the graph.

The importance of the edges in the kinematic tree suggests that an incidence list is the best representation for this situation. For example, say we have a tree as shown in figure 3.

This takes the form:

$$base : \left[T_1^{base}, T_2^{base}\right] \tag{15}$$

$$1 : \left[T_3^{1}, T_{base}^{1}\right] \tag{16}$$

$$2 : \left[T_{base}^{2}\right] \tag{17}$$

$$3 : \left[T_4^{3}, T_5^{3}\right] \tag{18}$$

$$4 : \left[T_3^{4}\right] \tag{19}$$

$$5 : \left[T_3^{5}\right] \tag{20}$$

$$\tag{21}$$

We can improve memory storage by using a dictionary (hash map) in order to only save the relevant edges. The construction algorithm is shown in algorithm 1.

Once the incidence list is constructed, then we can use it to query the tree to find paths between frames. The incidence list can keep track of the direction by assigning a flag to each edge specifying whether it is backwards or forwards. This will be useful for the rooting algorithm, shown later.

An important thing to note is that in the context of the kinematic tree, an edge transform T_j^i contains a child j and a parent i similar to a node. However an edge can only have one parent and one child, whereas a node can have only one parent, but many children.

4.2 Lookup path between frames

Now say we wanted to know the transform between two frames i, j on the tree that aren't connected by an existing edge. We can find this edge, T_i^j by finding the path between the two frames, and then multiplying the transforms we pass through, as shown in figure 4.

This can be done by executing a breadth first search on the tree in an attempt to find a path between two frames. The algorithm is given in alorithm 2. The algorithm shows how the algorithm is modified to deal with the need to store edges. Once we have a path as a sequence of edges, we can multiply these transforms together to obtain the transform between frames i, j.

Algorithm 1: Incidence list representation for the Kinematic Tree

```
Input: Unordered edge set $T$
Output: Hash map where frames are the keys and the transforms they are
        incident with are values
tree ← {}
// hash.insert(key :  value)
tree.insert(root : $\varnothing$)
for $t \in T$ do
    $p \leftarrow t.parent$
    $c \leftarrow t.child$
    // process parents first
    if $p \neq NULL$ then
        if $p \in tree$ then
            // we store list of transforms touching $p$
            tree[$p$].append($t$)
        else
            tree.insert($p : \varnothing$)
        end
    end
    // process children
    if $c \neq NULL$ then
        if $c \in tree$ then
            // store whether the edge is forwards (used later)
            $(t^{-1}).forwards \leftarrow false$
            tree[$c$].append($t^{-1}$)
        else
            tree.insert($c : \varnothing$)
        end
    end
end
return $tree$
```

Algorithm 2: Lookup between frames using breadth first search. The backpointers can then be used to extract the path of n transforms between the frames: $\prod_i^n T_i^{p(i)} T_{c(i)}^i$

```
Input: start frame, end frame
Output: backpointers dictionary
$O \leftarrow \varnothing$
// the main search function
def search_for_path($x_{start}, x_{goal}$):
    $C \leftarrow \varnothing$
    // the backpointers dictionary stores edges we've passed through
    backpointers = {}
    $O$.insert(start)
    backpointers.insert(start:$NULL$)
    while $O \notin \varnothing$ do
        $x \leftarrow O$.pop()
        $C$.insert($x$)
        if expand($x$) then
            return Success, backpointers
        end
    end
    return Failure, backpointers
// The node expansion function
def expand($x$):
    // get the ordered set of transforms linked to $x$
    $T \leftarrow x.edges$
    for $t \in T$ do
        if $t.child = end$ then
            backpointers.insert($t.child : t$)
            return Success
        else if $t.child \notin O$ and $t.child \notin C$ then
            backpointers.insert($t.child : t$)
            O.insert($t.child$)
        end
        return Failure
    end
```

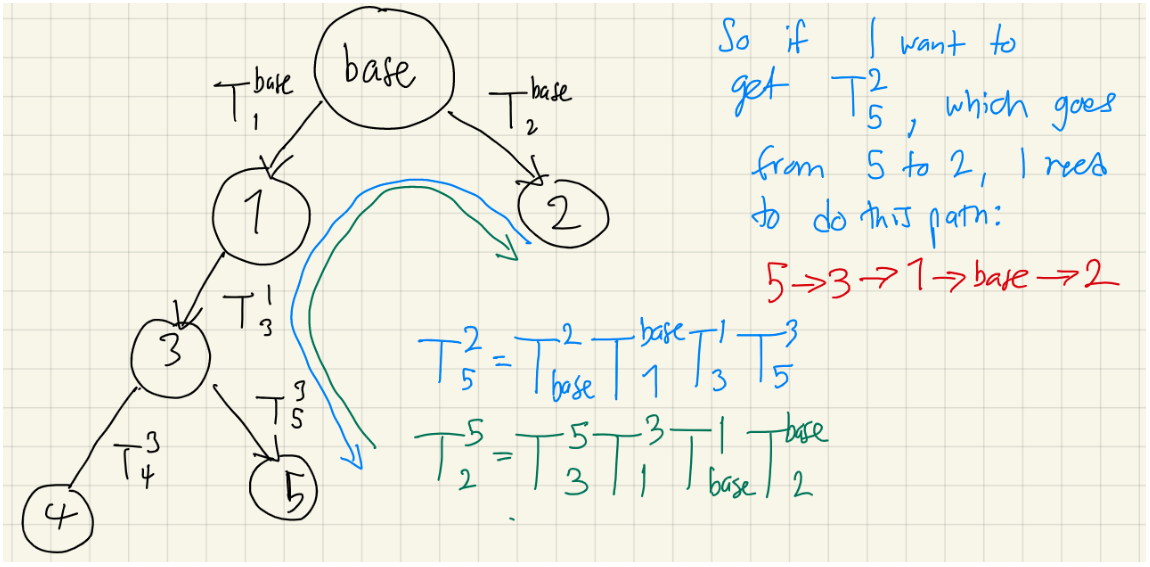

base
T_1^{base}
T_2^{base}
1
2
T_3^1
3
T_5^3
T_4^3
4
5
So if I want to get T_5^2, which goes from 5 to 2, I need to do this path:
$5 \to 3 \to 1 \to base \to 2$
$T_5^2 = T_{base}^2 T_1^{base} T_3^1 T_5^3$
$T_2^5 = T_3^5 T_1^3 T_{base}^1 T_2^{base}$

Figure 4: Lookup path between frames

4.3 Root function

The other main algorithm used by the kinematic tree is the root function. The purpose of this is simply to put every frame in the tree into the base frame. From here, it is trivial to place the entire tree in coordinates of any frame in the tree. If we use figure 4 for reference, then we want the frames $T_1^{base}, T_2^{base}, T_3^{base}, T_4^{base}, T_5^{base}$. We can obtain this through a depth first search that visits nodes in order and extracts the transform during the traversal. Using the tree in figure 4, we would visit the nodes from left to right: 4, 3, 5, 1, 2.

This is shown in algorithm 3. The root function can then be used to graph the entire tree in the base frame or any other frame in the tree.

Algorithm 3: The depth first traversal used to derive the set of transforms $\mathcal{T} = \{T_i^{base} \forall i : i \in K\}$ with frames i in kinematic tree K.

```
1  $O \leftarrow \varnothing$
   **Input:** Root of the tree
   **Output:** Dictionary of transforms $T_i^{base}$ for each frame $i$ in the tree
2  frames = {}
3  $O \leftarrow \varnothing$
4  $C \leftarrow \varnothing$
5  $O$.insert(root)
6  **while** $O \neq \varnothing$ **do**
7      $x \leftarrow O$.pop
8      $C$.insert($x$)
9      expand($x$)
10 **end**
11 **return** *frames*
   // depth first search expansion
12 **def** expand($x$):
       // traverse the ordered set $T$ of neighboring edges in reverse order
13     $R \leftarrow reverse(T)$
14     **forall** $t \in R$ **do**
15         **if** $t.forward()$ **then**
16             $x \leftarrow t.child$
17             $p \leftarrow t.parent$
18             **if** $x \notin O and x \notin C$ **then**
19                 $O$.insert($x$)
20                 $t_x^p \leftarrow$ frames[$p$]
21                 frames.insert($x : t_x^p * t$)
22             **end**
23         **end**
24     **end**
```