TriP Documentation

Release 0.9

Torben Miller

May 06, 2022

CONTENTS

Have you ever worked with a robot with hydraulic actuators? Or ever have to compensate bad motors by having them moving the joint via a complicated linkage?

Then you have worked with a hybrid kinematic chain.

TriP is a python library designed to calculate the forward- and inverse-kinematics of such chains. Since hybrid chains are the most general type of rigid mechanism this includes almost all robots.

FEATURES

- Model any robot (including closed and hybrid chains)

- Generate symbolic representations of forward kinematics

- Compute Jacobian matrices for differential kinematics

- Compute the inverse kinematics of arbitrary rigid mechanisms

- Compute the Inverse Kinematics in position and/or orientation

- Support arbitrary joint types and quaternions

- Includes several ready to use examples (TriPed robot, Excavator Arm)

- TriPs validates the inverse kinematics algorithms with extensive testing using analytic solutions.

1.1 How TriP models Robots

TriP models robots using the `Robot` class. A robot object is made up of `Transformation` and `KinematicGroup` objects. The `KinematicGroup` objects are used to model closed chains.

The following sections will explain the `Transformation`, `KinematicGroup` and `Robot` objects in more detail. It is advised to read these sections before using them to model robots.

The last section also explains how Kinematic Solvers work.

1.1.1 Transformations

A Kinematic model is made up of Coordinate systems. These coordinate systems are connected by transformations.

TriP implements its own *Transformation* class.

One can distinguish between static transformations and dynamic transformations. Dynamic transformations change depending on an internal state thereby modeling the joints of a mechanism.

The *Transformation* class has an attribute that manages the internal state.

Transformation Descriptions

In general, states can influence the transformation in arbitrary ways. Yet robotics uses several standard conventions.

The `Transormation` class currently supports the following conventions:

- translation with Euler angle rotation
- translation with quaternion rotation

Translation with Euler Angles rotation

This convention is perhaps the most natural and intuitive. In this convention, the transformation is specified using 6 parameters *[tx ty tz rx ry rz]*. These parameters have the following interpretation:

parameter	interpretation
tx	moves the coordinate system along the x-axis
ty	moves the coordinate system along the y-axis
tz	moves the coordinate system along the z-axis
rx	rotates the coordinate system around the x-axis
ry	rotates the coordinate system around the y-axis
rz	rotates the coordinate system around the z-axis

Important: In this convention, rotation is always applied before translation.

The Euler angles follow the *XYZ* convention. This means that the transformation first rotates around x, then around y, and lastly around z. This convention is also called Roll, Pitch, and Yaw. Here rx=Roll, ry=Pitch, and rz=Yaw.
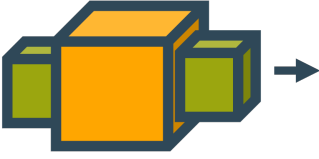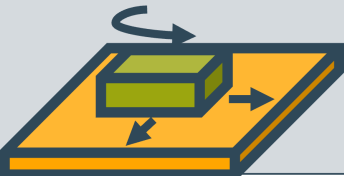
This transformation is captured by the following transformation matrix:

$$\begin{pmatrix} \cos rz \cos ry & \cos rz \sin ry \sin rx - \sin rz \cos rx & \cos rz \sin ry \cos rx + \sin rz \sin rx & t_x \\ \sin rz \cos ry & \sin rz \sin ry \sin rx + \cos rz \cos rx & \sin rz \sin ry \cos rx - \cos rz \sin rx & t_y \\ -\sin ry & \cos ry \sin rx & \cos ry \cos rx & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The definition of joints in this convention is very straightforward, below is a sample list of different joints:

	Transformation(name = 'Revolute Joint',
	               values = { 'rz': 0},
	               state_variables = ['rz'])

	Transformation(name = 'Spherical Joint',
	               values = {'rx': 0, 'ry': 0, 'rz': 0},
	               state_variables = ['rx', 'ry', 'rz'])

	Transformation(name = 'Prismatic Joint',
	               values = { 'tz': 0},
	               state_variables = ['tz'])

	Transformation(name = 'Planar Slider Joint',
	               values = {'tx': 0, 'ty': 0, 'rz': 0},
	               state_variables = ['tx', 'ty', 'rz'])

Note that while all non specified parameters are assumed to be zero, the value of each *state_variable* still has to be supplied.

Translation with Quaternion rotation

Quaternions are an alternative four-dimensional description of rotation. They have many advantages compared to Euler angles, which are explained here . However, they trade these advantages for an intuitive interpretation.

| parameter	interpretation
tx	moves the coordinate system along the x-axis
ty	moves the coordinate system along the y-axis
tz	moves the coordinate system along the z-axis
qw	first quaternion, also called a.
qx	second quaternion, also called b.
qy	third quaternion, also called c.
qz	fourth quaternion, also called d.

The corresponding matrix is:

$$\begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) & t_x \\ 2(q_x q_y + q_z q_w) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_x q_w) & t_y \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & 1 - 2(q_x^2 + q_y^2) & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Important: The matrix only describes a rotation if all quaternions are normalized, meaning $qw^2 + qx^2 + qy^2 + qz^2 = 1$. Since the current inverse kinematics solver does not support constraints this means that quaternions are not supported when calculating inverse kinematics.

Transformation trees

To fully specify the kinematic model of a robot not only the transformations are needed but also how they are connected. This is described by the so-called transformation tree. Conventionally nodes of this tree describe coordinate frames while its edges describe transformations. An example can be seen down below:
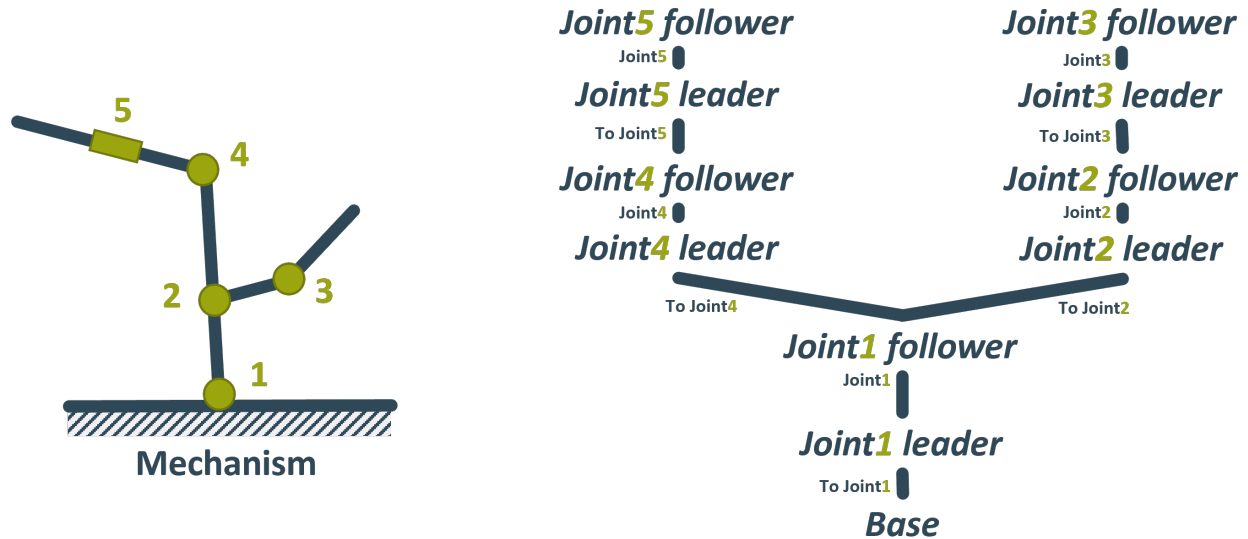
Here the cursive graph nodes are coordinate frames while the edges are the transformations between them. Since TriP only models transformations and not coordinate frames in TriP the name of a coordinate frame is synonymous with the name of the transformation leading to it. This leads to the following simplified transformation tree:
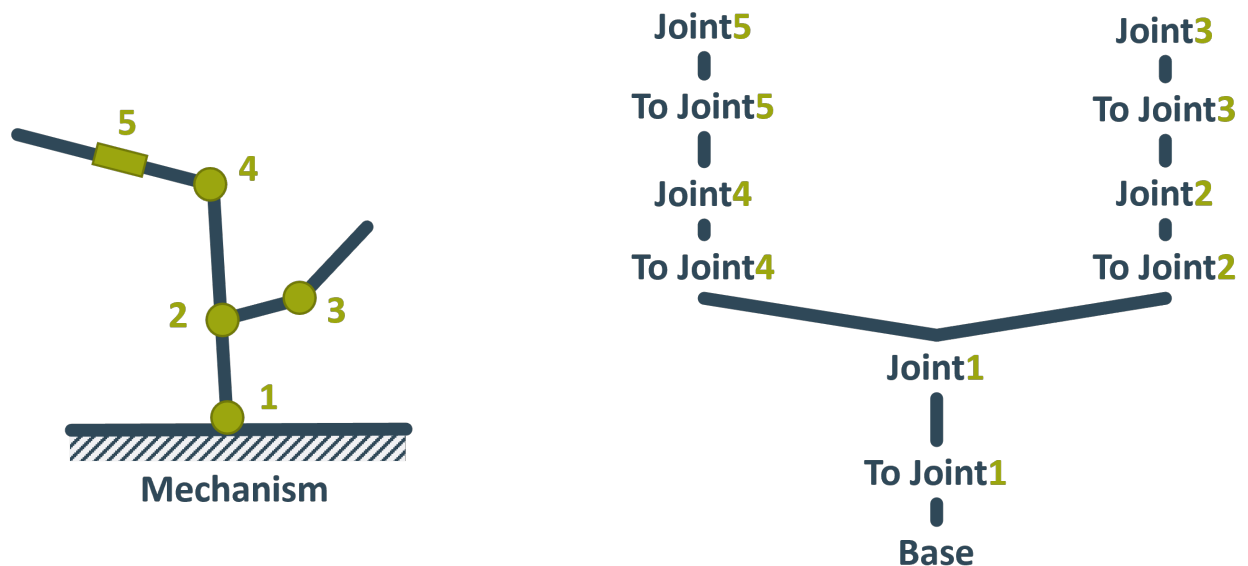
In this tree the edge and the node it leads to refer to the transformation. TriP builds this simplified transformation tree by specifying the parent of each transformation. The parent is in this case the transformation that preceded the current transformation. For the example transformation tree this would look like this:

```
to_joint_1 = Transformation(name="To Join1")
```

```
joint_1     = Transformation(name="Joint1",values={'ry': 0},state_variables=['ry'],
→parent=to_joint_1)

to_joint_2 = Transformation(name="To Joint2",values={'tx':1},parent=joint_1)
joint_2     = Transformation(name="Joint2",values={'ry': 0},state_variables=['ry'],
→parent=to_joint_2)
to_joint_3 = Transformation(name="To Joint3",values={'tx':1},parent=joint_2)
joint_3     = Transformation(name="Joint3",values={'ry': 0},state_variables=['ry'],
→parent=to_joint_3)

to_joint_4 = Transformation(name="To Joint4",values={'tx':1},parent=joint_1)
joint_4     = Transformation(name="Joint4",values={'ry': 0},state_variables=['ry'],
→parent=to_joint_4)
to_joint_5 = Transformation(name="To Joint5",values={'tx':1},parent=joint_4)
joint_5     = Transformation(name="Joint5",values={'ry': 0},state_variables=['ry'],
→parent=to_joint_5)
```

Important: Transformations with no parent are considered connected to the base Frame. Since for most robots, this is where they are connected to the floor this frame is also called Ground. This can be seen in transformation `to_joint_1`. Note that strictly speaking this transformation is necessary since its transformation is an identity matrix. It is only included for clarity.


The transformation tree building concept does not work if more than one transformation leads to the same frame. Here one would have to distinguish between the transformations leading to the frame and the frame itself. Such a situation is referred to as a closed kinematic chain, the next section will explain how they are modeled in TriP.

1.1.2 Kinematic Groups

Most kinematic libraries rely only on transformation objects because they only model open chains. An example for this is IKPY . In an open chain, the position and orientation of a coordinate system depend only on one transformation from its parent.

But, consider the excavator arm below:
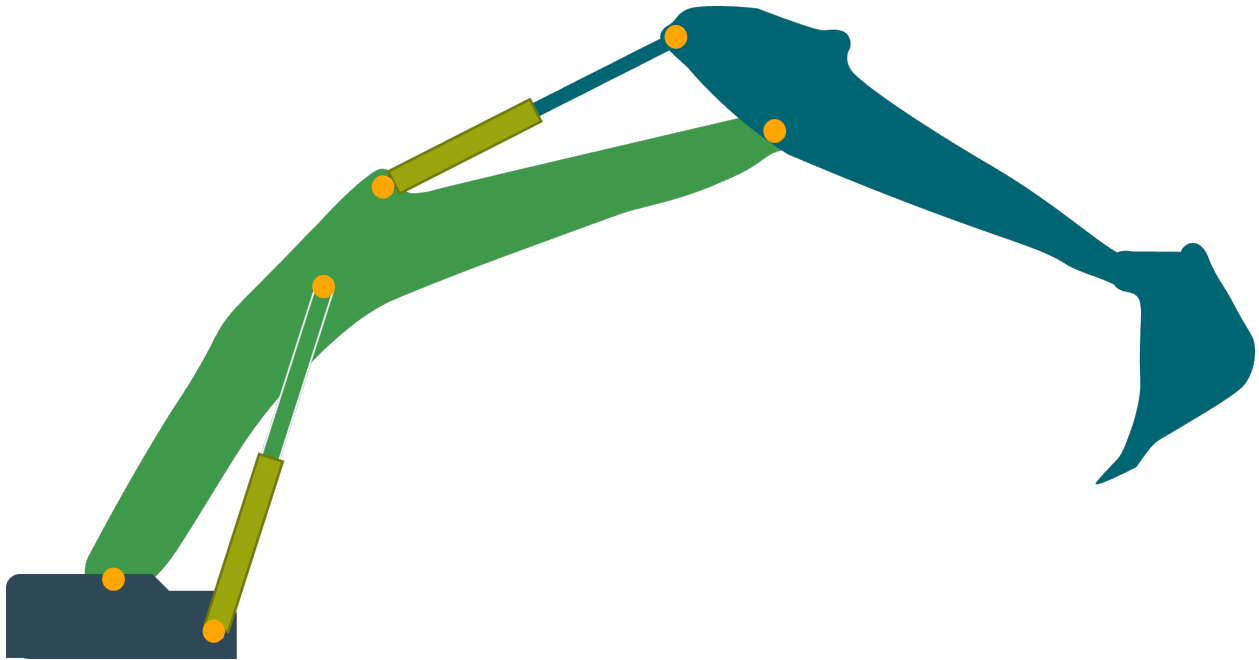
In this example, multiple coordinate systems have more than one parent since the transformations form a loop.

Such a loop is called a closed kinematic chain.

Classically closed chains are modeled using an algebraic closure equation $g(q) = 0$. The closure equation couples all joint states q so that multi transformations leading to the same frame all agree on the state of the frame.

In practice, this is computationally expensive and often entirely unnecessary.

Important: To simplify the system one could treat the system as if the hinges of the excavator's arm were directly actuated.

This simplified virtual chain contains no closed loops and thus standard kinematics algorithms can be used to compute forward or inverse kinematics.

To get the solution of the real excavator, one simply has to convert between the state of the hinges and the state of the hydraulic cylinders.

This can be done using some kind of mapping function based on trigonometry.

TriP embraces this mapping approach and implements it using the `KinematicGroup` class. A `KinematicGroup` is made up of a *virtual_chain*, an *actuated_state*, and two mappings. The mappings convert between the state of the *virtual_chain*, called *virtual_state*, and the state of the actuated joints called *actuated_state*.
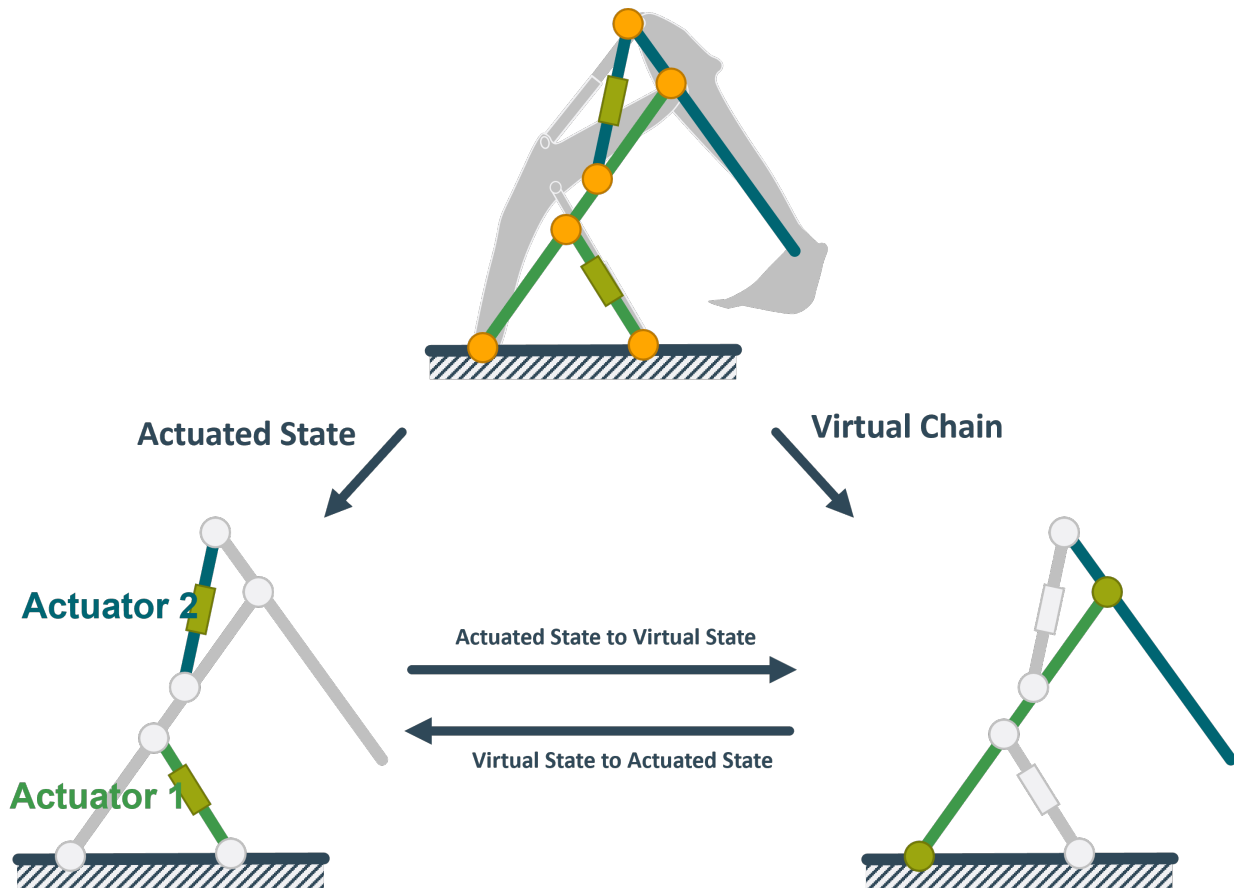
Important: The virtual_chain has to be a single open chain without branches. The reasons for this will be discussed in the next section

divide a robot into groups

In the example above the excavator is modeled as a single group. However, it is also possible to divide the excavator into multiple groups. These groups can then be combined just like transformations. Multiple smaller groups have two advantages over a single large group:

For one it improves modularity, making it easier to reuse assembly parts.

But more importantly, it reduced computational cost. To keep virtual and actuated state consistent mapping has to be called every time part of one state changes. A single group mechanism would mean updating every state. This problem is especially bad for branching mechanisms. Consider a four-legged robot, setting the actuator of one leg would then mean updating all four legs. To prevent this problem outright the virtual chain can not contain branches.

In summary, groups should be defined as small as possible. Small in this case refers to the number of actuators that have to be grouped. The minimum size is defined by the closed chains. Consider the following mechanism

a)   b)   c)
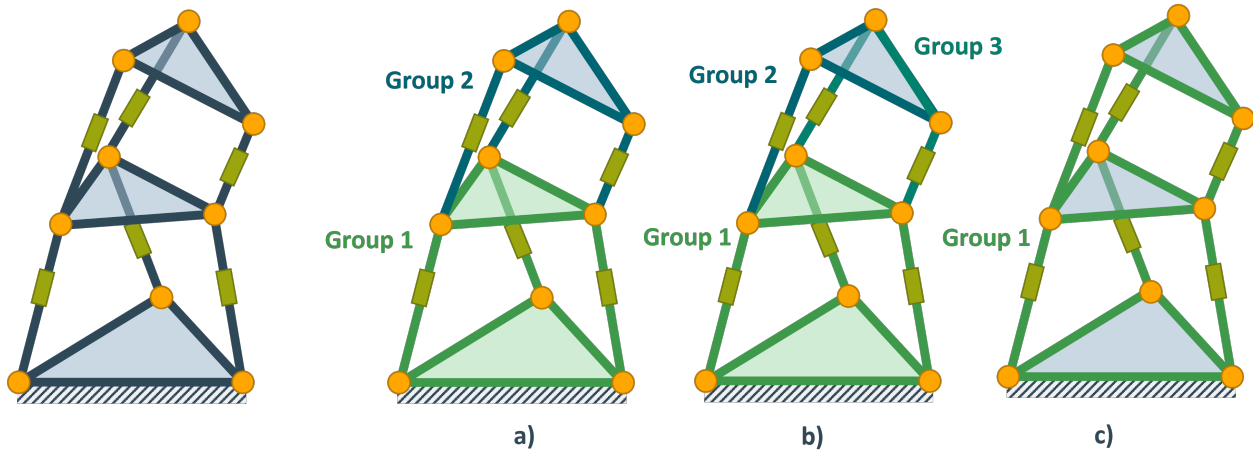
Grouping a) and c) are valid groups, with a) being more performant. However the Grouping in b) is not valid. The reason is that the state of the top platform depends on the state of all three green prismatic joints.

These considerations lead to the following guidelines for building hybrid robots:

Important: Every closed chain should be modeled by a Group. Every open chain should be modeled by Transformations. Se the following robots as an example:

(a)   (b)   (c)
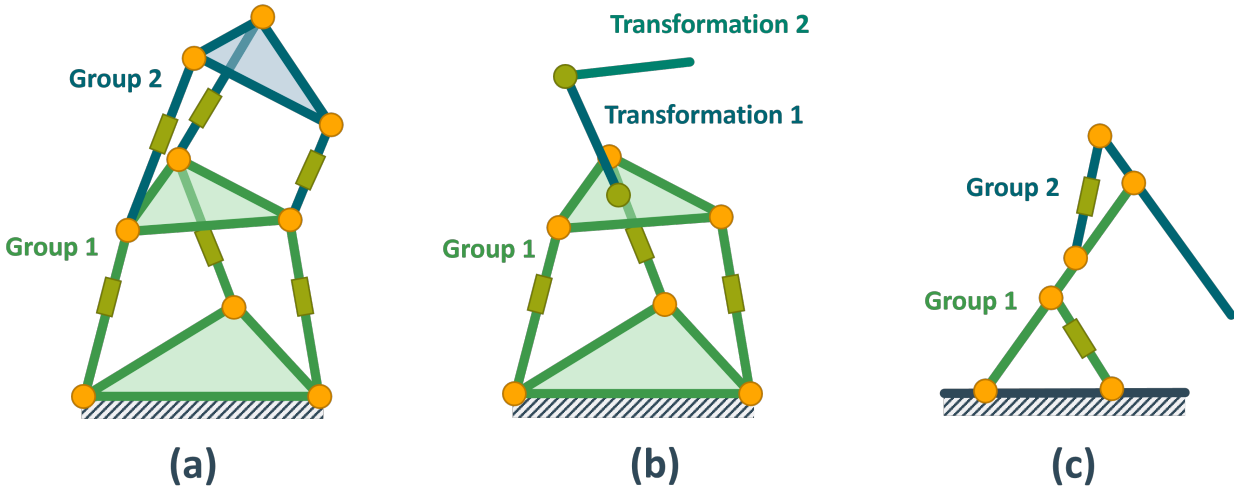
The excavator has two actuated states and two virtual states. These are the lengths of hydraulic cylinders a_1, a_2 and the arm angles q_1, q_2. Since each cylinder length a_i controls one arm angle q_i, the excavator can be divided into two groups. These are visualized by the green and blue parts respectively.

The mappings for each group can be calculated using trigonometry:
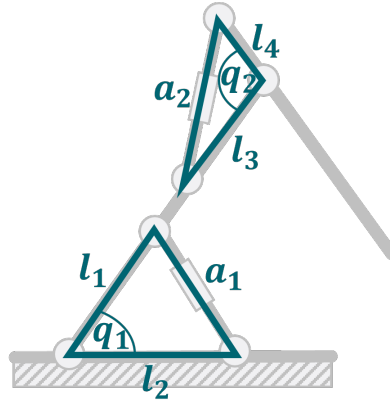
The full code for the excavator looks like this:

```python
from typing import Dict
from math import radians
import casadi
import numpy as np

from trip_kinematics.Utility import hom_rotation, get_translation
from trip_kinematics.Utility import hom_translation_matrix, y_axis_rotation_matrix
from trip_kinematics.KinematicGroup import KinematicGroup
from trip_kinematics.Transformation import Transformation
from trip_kinematics.Robot import Robot


l_1 = 1
l_2 = 0.7
l_3 = 1.2
l_4 = 0.4
l_5 = 1.7

# zero conventions for the actuated joints
a_1_offset = 0
a_2_offset = 0

virtual_joint_1 = Transformation(name="q_1",
                                 values={'ry': 0},
                                 state_variables=['ry'])
link_1 = Transformation(name="link_1",
                        values={'tx': l_1+l_3+0.4},
                        parent=virtual_joint_1)
virtual_joint_2 = Transformation(name="q_2",
                                 values={'ry': radians(-90)},
                                 state_variables=['ry'])
link_2 = Transformation(name="link_2",
                        values={'tx': l_5},
                        parent=virtual_joint_2)


###############################################
# Direct mappings using geometric calculations #
```

(continues on next page)

```python
39    ################################################
40
41    def geometric_q_to_a_group_1(state: Dict[str, float], tips: Dict[str, float] = None):
42        # convert joint angle to triangle angle
43        q_1 = radians(90) - state['q_1']['ry']
44        return {'a_1': np.sqrt(l_1**2+l_2**2-2*l_1*l_2*np.cos(q_1))}
45
46
47    def geometric_a_to_q_group_1(state: Dict[str, float], tips: Dict[str, float] = None):
48        a_1 = state['a_1'] + a_1_offset
49        return {'q_1': {'ry': np.arccos((l_1**2+l_2**2-a_1**2)/(2*l_1*l_2))}}
50
51
52    def geometric_q_to_a_group_2(state: Dict[str, float], tips: Dict[str, float] = None):
53        q_2 = -1 * state['q_2']['ry']  # convert joint angle to triangle angle
54        return {'a_2': np.sqrt(l_3**2+l_4**2-2*l_3*l_4*np.cos(q_2))}
55
56
57    def geometric_a_to_q_group_2(state: Dict[str, float], tips: Dict[str, float] = None):
58        a_2 = state['a_2'] + a_2_offset
59        return {'q_2': {'ry': np.arccos((l_3**2+l_4**2-a_2**2)/(2*l_3*l_4))}}
60
61
62    geometric_group_1 = KinematicGroup(name="geometric group 1",
63                                       virtual_chain=[virtual_joint_1, link_1],
64                                       actuated_state={'a_1': 0},
65                                       actuated_to_virtual=geometric_a_to_q_group_1,
66                                       virtual_to_actuated=geometric_q_to_a_group_1)
67
68    geometric_group_2 = KinematicGroup(name="geometric group 2",
69                                       virtual_chain=[virtual_joint_2, link_2],
70                                       actuated_state={'a_2': 0},
71                                       actuated_to_virtual=geometric_a_to_q_group_2,
72                                       virtual_to_actuated=geometric_q_to_a_group_2,
73                                       parent=geometric_group_1)
74
75    geometric_excavator = Robot([geometric_group_1, geometric_group_2])
```
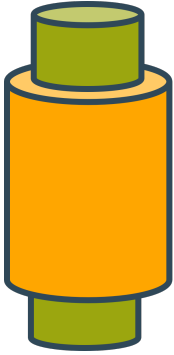
actuated state vs virtual state

If one looks at the code above one can see that the dictionary values of the actuated state in lines 26 and 36 are float values, while the values of the virtual states in lines 32 and 39 are dictionaries.

This difference is because virtual states always specify convention parameters of a `Transformation`. Actuated values on the other hand are not associated with a Transformation and thus don't adhere to transformation conventions.

This is an important difference to keep in mind when dealing with both states. Below are a few examples of joints and how their actuated and virtual states would differ.

actuated_state = {revolute: θ_z}
virtual_state = {revolute: {rz : θ_z}}
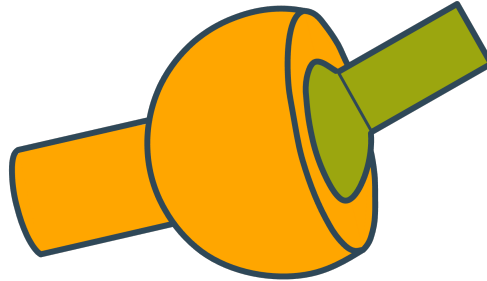
actuated_state = {spherical_1 : θ_x, spherical_2 : θ_y, spherical_3 : θ_z}
virtual_state = {spherical: { rx : θ_x, ry : θ_y, rz : θ_z}}

Using closure equations

While direct mappings are always preferable it is not always possible to find a direct function. In this case, one can always resort to the closure equation. Since TriP is based on mappings the closure equation is used to set up mapping functions that solve the closure equation. For the mapping from actuated state to virtual state, the actuated states are fixed and the virtual states calculated. Likewise, for the reverse mapping, the virtual state is fixed while the actuated states are calculated.

The setup of the closure equation will require extra transformations. This can be done by building a full open chain or for simple chains by directly setting up the transformation matrices using the Utility submodule. In this case of the excavator, the following joints can be defined:
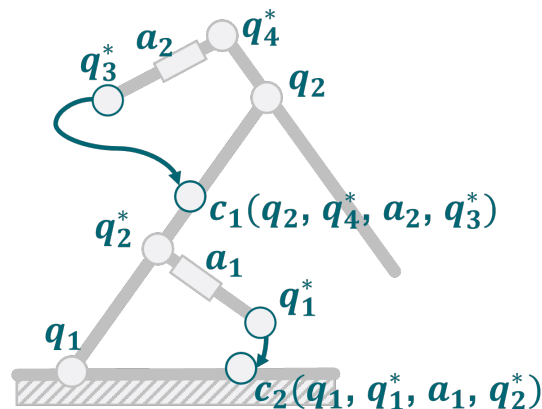
The solving of the closure equation can be performed by casadi, which TriP also uses for inverse kinematics calculations:

```python
from typing import Dict
from math import radians
import casadi
import numpy as np

from trip_kinematics.Utility import hom_rotation, get_translation
from trip_kinematics.Utility import hom_translation_matrix, y_axis_rotation_matrix
from trip_kinematics.KinematicGroup import KinematicGroup
from trip_kinematics.Transformation import Transformation
from trip_kinematics.Robot import Robot


l_1 = 1
l_2 = 0.7
l_3 = 1.2
l_4 = 0.4
l_5 = 1.7

# zero conventions for the actuated joints
a_1_offset = 0
a_2_offset = 0

virtual_joint_1 = Transformation(name="q_1",
                                 values={'ry': 0},
                                 state_variables=['ry'])
link_1 = Transformation(name="link_1",
                        values={'tx': l_1+l_3+0.4},
                        parent=virtual_joint_1)
virtual_joint_2 = Transformation(name="q_2",
                                 values={'ry': radians(-90)},
                                 state_variables=['ry'])
link_2 = Transformation(name="link_2",
                        values={'tx': l_5},
                        parent=virtual_joint_2)


####################################################################
# mappings using colsure equation solution geometric calculations #
####################################################################
opts = {'ipopt.print_level': 0, 'print_time': 0}

closure_1_state = casadi.SX.sym('cls_1_q', 3)

cls_q_1 = hom_rotation(y_axis_rotation_matrix(closure_1_state[0]))
cls_l_1 = hom_translation_matrix(t_x=l_1)
cls_qs_2 = hom_rotation(y_axis_rotation_matrix(closure_1_state[1]))
cls_a_1 = hom_translation_matrix(t_x=closure_1_state[2])
cls_a_1z = hom_translation_matrix(t_x=a_1_offset)
cls_1_trafo = cls_q_1 * cls_l_1 * cls_qs_2 * cls_a_1 * cls_a_1z

cls_1_trafo_pos = get_translation(cls_1_trafo)

c_1 = (cls_1_trafo_pos[0]-l_2)**2 + \
```

```python
54          cls_1_trafo_pos[1]**2 + cls_1_trafo_pos[2]**2
55
56
57   def closure_q_to_a_group_1(state: Dict[str, float]):
58       nlp = {'x': closure_1_state[1:], 'f': c_1, 'p': closure_1_state[0]}
59       nlp_solver = casadi.nlpsol('q_to_a', 'ipopt', nlp, opts)
60       solution = nlp_solver(x0=[0, 0], p=[state['q_1']['ry']])
61       sol_vector = np.array(solution['x'])
62       return {'a_1': sol_vector[1]}
63
64
65   def closure_a_to_q_group_1(state: Dict[str, float]):
66       nlp = {'x': closure_1_state[:1], 'f': c_1, 'p': closure_1_state[2]}
67       nlp_solver = casadi.nlpsol('a_to_q', 'ipopt', nlp, opts)
68       solution = nlp_solver(x0=[0, 0], p=[state['a_1']])
69       sol_vector = np.array(solution['x'])
70       return {'q_1': {'ry': sol_vector[0]}}
71
72
73   closure_2_state = casadi.SX.sym('cls_2_q', 3)
74
75   cls_q_2 = hom_rotation(y_axis_rotation_matrix(closure_2_state[0]))
76   cls_l_4 = hom_translation_matrix(t_x=l_4)
77   cls_qs_4 = hom_rotation(y_axis_rotation_matrix(closure_2_state[1]))
78   cls_a_2 = hom_translation_matrix(t_x=closure_2_state[2])
79   cls_a_2z = hom_translation_matrix(t_x=a_1_offset)
80   cls_2_trafo = cls_q_2 * cls_l_4 * cls_qs_4 * cls_a_2 * cls_a_2z
81
82   cls_2_trafo_pos = get_translation(cls_1_trafo)
83
84   c_2 = (cls_2_trafo_pos[0]+l_3)**2 + \
85          cls_2_trafo_pos[1]**2 + cls_2_trafo_pos[2]**2
86
87
88   def closure_q_to_a_group_1(state: Dict[str, float]):
89       nlp = {'x': closure_2_state[1:], 'f': c_2, 'p': closure_2_state[0]}
90       nlp_solver = casadi.nlpsol('q_to_a', 'ipopt', nlp, opts)
91       solution = nlp_solver(x0=[0, 0], p=[state['q_1']['ry']])
92       sol_vector = np.array(solution['x'])
93       return {'a_2': sol_vector[1]}
94
95
96   def closure_a_to_q_group_1(state: Dict[str, float]):
97       nlp = {'x': closure_2_state[:1], 'f': c_2, 'p': closure_2_state[2]}
98       nlp_solver = casadi.nlpsol('a_to_q', 'ipopt', nlp, opts)
99       solution = nlp_solver(x0=[0, 0], p=[state['a_1']])
100      sol_vector = np.array(solution['x'])
101      return {'q_2': {'ry': sol_vector[0]}}
102
```

Defining virtual chains

In the vast majority of cases, the specification of the virtual chain is straightforward. One simply uses a single chain of transformations that goes from one end of the group to the other. However, in some cases, this can lead to unintended or suboptimal results.

As a simple example of this problem, think of the excavator arm from above. Assuming that it had a spherical joint at the elbow, the system would still not be able to move any differently. However, the virtual open chain which neglects the hydraulic cylinders would suddenly behave much differently.

An inverse kinematics solver might now try to find open chain configurations that are not possible with the full mechanism.

> **Warning:** Since TriP currently does not support Joint limits, it can not detect which open chain configurations are not possible. This can lead to solvers failing outright.

This problem can be avoided by designing a custom virtual open chain. In the case of the excavator this is very simple, just substitute the spherical joint with a revolute joint. For more complicated robots this might be more complex, a general rule of thumb is:

Important: The virtual open chain should offer the same degrees of freedom as the full mechanism. Ideally, the correspondence between virtual joints and actuated joints should be as simple as possible.

1.1.3 Robots

The Robot class is the centerpiece of TriP, they encapsulate `Transormation` and `KinematicGroup` objects of a Robot. This causes some problems. Groups distinguish between actuated_states and virtual_states while for a transformation these are the same.´

Important: To solve this problem Transformations are internally converted into groups. The actuated state of a transformation follows the naming convention *NAME_KEY* where *NAME* is the name of the `Transformation` and *KEY* is a key for the state of the transformation. The full actuated state of a robot can be returned using the *get_actuated_state* method.

For robots without closed chains, both the virtual_state and the actuated_state can be used interchangeably. We advise nevertheless to use the actuated_state whenever possible as a general convention.

End Effectors

The `Robot` class offers the same functionality as the Group object at a more abstract level. Additionally, it supports end effectors. In robotics, an end effector is conventionally a device at the end of a robotics arm which interacts with the environment. Kinematically and more, generally speaking, it is a coordinate frame whose position and orientation is of special interest. This might be because it holds a tool, or because it specifies a foot position or maybe it just holds a sensor.

In any case, end effectors are coordinate frames for which we might want to compute forward or inverse kinematics. Since for open chains frames and the transformations leading to them are synonymous an effectors can be any frame resulting from a transformation of a robots virtual_chain. Remember since the robot is comprised of groups it's virtual open chain is a concatenation of the virtual_chains of each group.

The *Robot* class offers the *get_endeffectors* method for ease of use which returns the names of all possible end effector frames.

Symbolic Representations

The robot class is capable of generating symbolic representations of end effector kinematics. This means it can describe the state of and end effector frame as a mathematical function $\tilde{p}(\tilde{q})$ whose input is the virtual state $\tilde{q}$.

This is handy for several reasons:

- It allows for quick calculations of the end effector position without needing matrix multiplication at every step

- It allows automatic mathematical derivation to calculate the Jacobi matrix and generally analyze the virtual chain

- It allows the setup of mathematical solvers which can compute the inverse kinematics for a given end effector.

Generally, the first point is not needed as the forward kinematics is quite fast on its own and can be called using the *forward_kinematic* function given an end effector and a robot object. However, if speed is the issue, the casadi library which is used for the symbolic representation is capable of generating C code from such a function object. This can be used to further speed up code executions or calculate the kinematics on an embedded device.

The main advantages, however, are the second and the third points which are related seeing that most numerical optimization requires the computation of gradients. Casadi features fast algorithmic differentiation (a powerful hybrid between numerical differentiation and analytic differentiation used in many machine learning solutions). This allows the fast calculations of gradients and Jacobi matrices.

As such TriP can also be used to do kinematic analysis for open chains. Note that it does not support closed chains because the mapping functions don't have to be casadi objects. However if one does use casadi like functions TriP can also be used to analyze hybrid chains.

In general, the symbolic representation returns a casadi object on which the full`casadi feature pallet can be used. This includes the setup of numerical solvers which are used to calculate the inverse kinematics.

Forward kinematics

The calculation of forward kinematics is done using the *forward_kinematic* function. The general procedure can be seen in the image below:
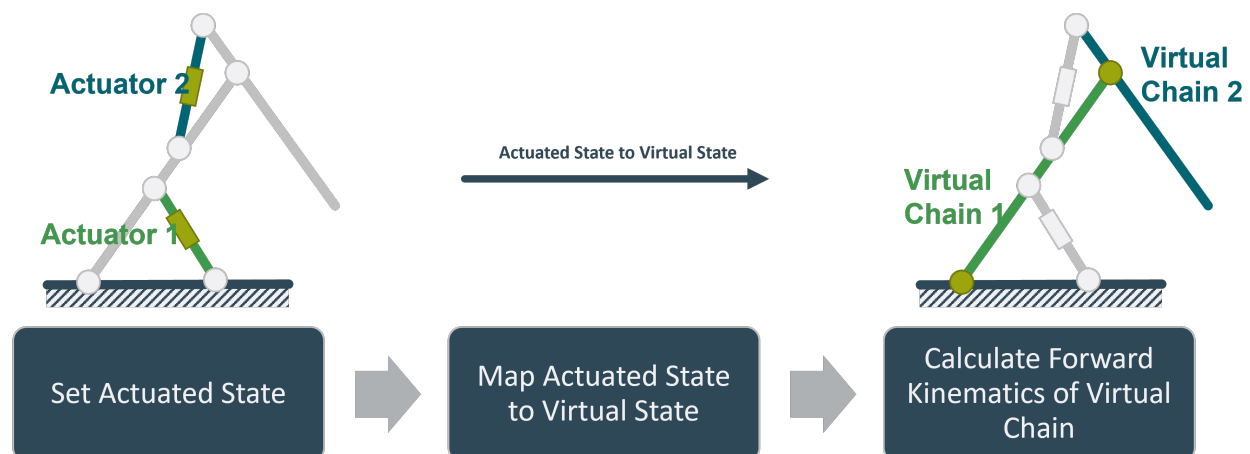
The forward kinematics of the virtual_chain are in this case calculated by multiypling the transformation matrices of the virtual_chain together. This results in a 4x4 transformation matrix describing the state of the end effector.

1.1.4 Solvers

Solvers are objects which calculate the inverse kinematics for a given *Robot* and end effector. The reason they are not functions like the forward_kinematic functions is that the generation and setup of a casadi solver object is the biggest performance bottleneck. To circumvent this, solver classes generate and store casadi solver objects for a given robot and end effector. Using casadis framework it is then easily possible to implement different types of solving algorithms.

TriP currently only has one type of solver object called `SimpleInvKinSolver` which will be further explained shortly. However it is also possible to write your own solver classes, all one has to do is implement the *solve_virtual* and *solve_actuated* functions which return the virtual and actuated state respectively given a desired end effector state and an optional initial solver tip.

SimpleInvKinSolver

The `SimpleInvKinSolver` is as the name implies a rudimentary solver whose general procedure can be seen in the image below:
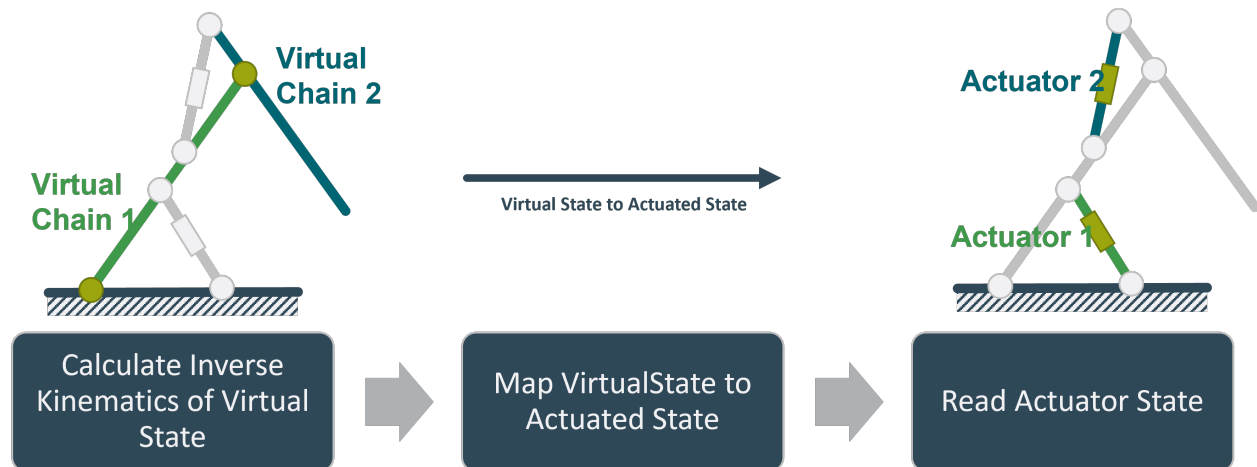
First, the system tries to find a virtual state which results in the end effector being as close as possible to the target. This is done using a casadis NLP solver using auto-generated jacobian and hessian matrices.

In the next step, the system calls the internal mapping functions of the robot to convert the virtual_state into an actuated_state.

> **Warning:** This sequential approach is not capable of handling virtual_states for which no actuate_state can be found. In this case, the solver simply fails. To prevent this, suitable starting values for the solver can be supplied and the virtual_chain can be set up to minimize the chance of this happening. See section *Defining virtual chains* for reference.

1.2 Getting Started

The current release of the project can always be found on the python package index PyPi . However, since another python package is already called trip, the package is called trip_kinematics. The current stable release can be installed using:

```
pip install trip_kinematics
```

Alternatively, the current development version can be downloaded using Github:

```
git clone https://github.com/TriPed-Robot/TriP
cd TriP
pip install src/
```

For more information on how to use the library once installed visit the Tutorials or read 'How TriP models Robots'.

1.3 Tutorials

1.3.1 Building a Robot Model

Before TriPs functionality can be used on a robot, it first has to be build within TriP. This Tutorial will show how to build the TriPed robot shown below:

Here each leg was highlighted in a different color. Since all legs are identicall, this tutorial will start with a single leg. More information about the triped legs can be found here .

The first step in setting up a robot is identifying the groups and transformations. TriP uses Groups to model closed kinematic chains. These are structures where multiple moving parts converge in a single location forming one or more loops. Some examples can be seen down below:

These closed chains will either be connected directly to each other or using a series of other moving parts. Such a series is called a open kinematic chain. Open Kinematic chains are handled using a series of transformations.

In the case of the TriPed the following chains can be identified:
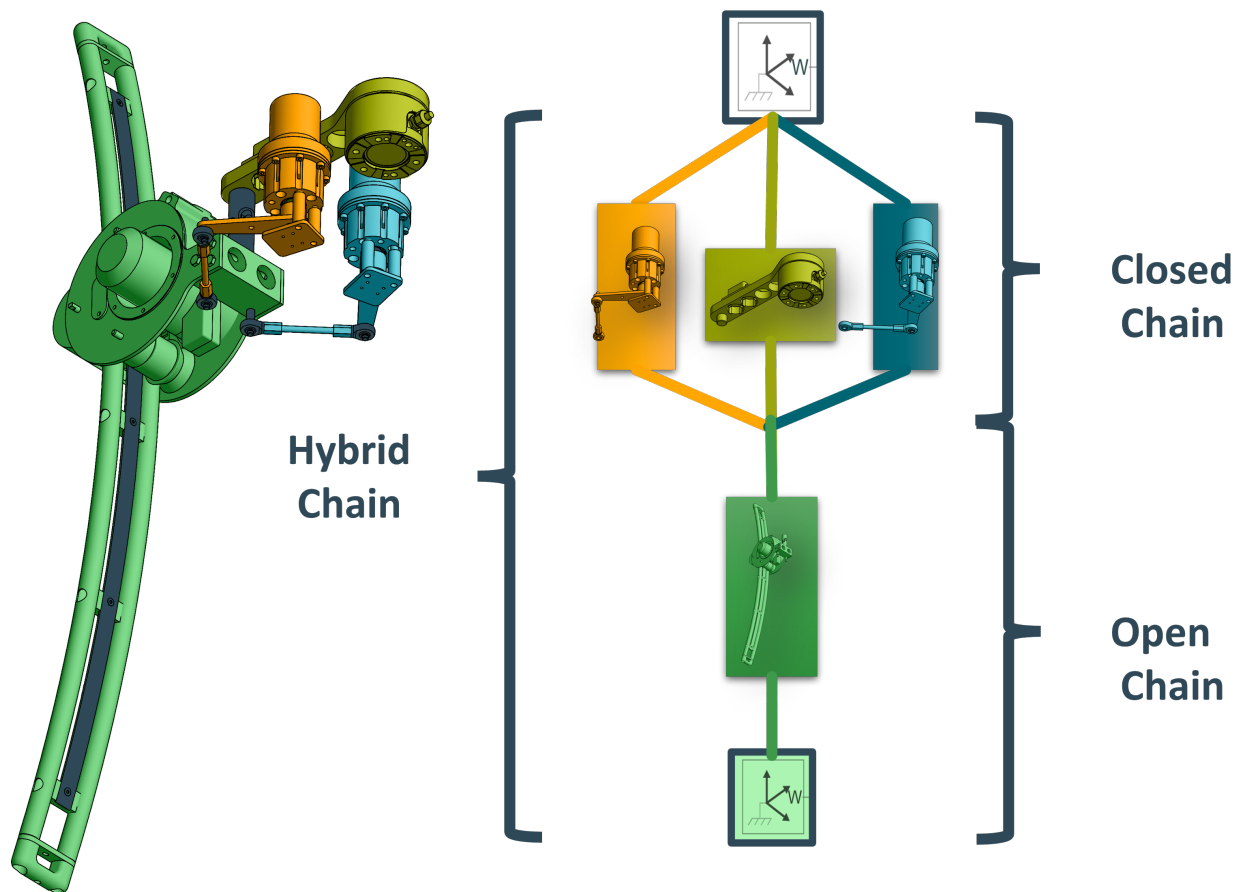
This in turn means that the leg of the TriPed contains one kinematic group and a number of transformations representing the open chain.

Once each group has been identified the group construction worklow goes like this:
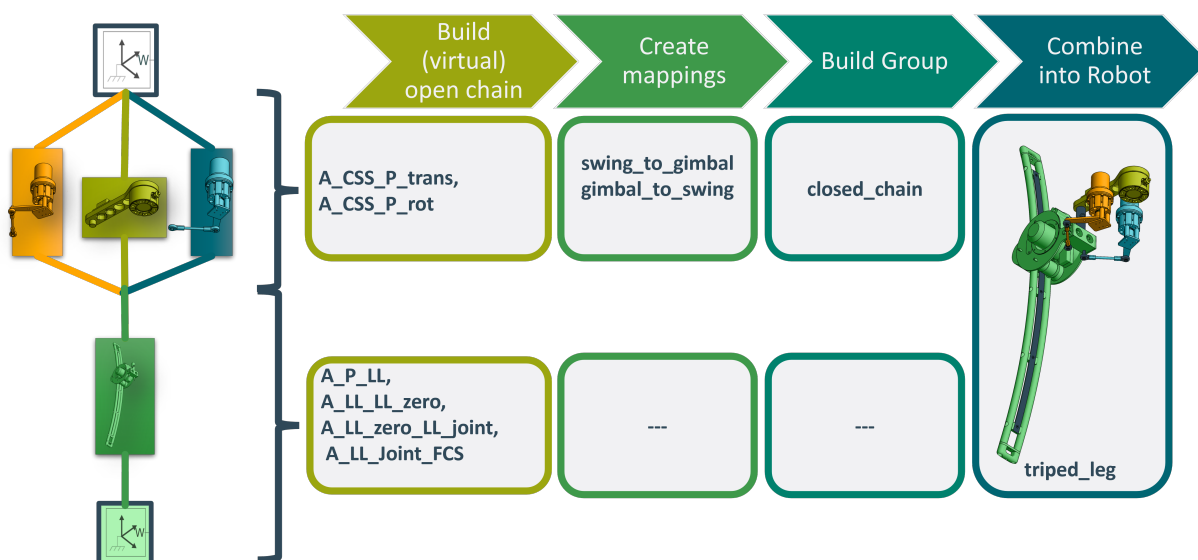
Building open chains

Groups handle closed chains by abstracting them into a virtual open chains that models how group moves combined with two mapping functions from these virtual joints to the actual actuated joints and back. This means that for both the open and the closed chain, a chain has to be build. The precise transformations depend on the type of robot and its conventions. The kinematic transformations of the TriPed are described here: here . This leads to the following virtual open chain:

```python
from math import radians
from typing import Dict
from casadi import SX, nlpsol, vertcat
import numpy as np


from trip_kinematics.KinematicGroup import KinematicGroup
from trip_kinematics.Transformation import Transformation
from trip_kinematics.Robot import Robot
from trip_kinematics.Utility import hom_translation_matrix, x_axis_rotation_matrix
from trip_kinematics.Utility import y_axis_rotation_matrix, z_axis_rotation_matrix
from trip_kinematics.Utility import hom_rotation, get_translation
a_ccs_p_trans = Transformation(name='A_ccs_P_trans',
                               values={'tx': 0.265, 'tz': 0.014})
a_ccs_p_rot = Transformation(name='gimbal_joint',
                             values={'rx': 0, 'ry': 0, 'rz': 0},
                             state_variables=['rx', 'ry', 'rz'],
                             parent=a_ccs_p_trans)
```

And the corresponding transformations for the open chain:

```python
a_p_ll = Transformation(name='A_P_LL',
                        values={'tx': 1.640, 'tz': -0.037, },
                        parent=closed_chain)
a_ll_zero = Transformation(name='zero_angle_convention',
                           values={'ry': radians(-3)},
                           parent=a_p_ll)
a_ll_zero_ll_joint = Transformation(name='extend_joint',
                                    values={'ry': 0},
                                    state_variables=['ry'],
                                    parent=a_ll_zero)
a_ll_joint_fcs = Transformation(name='A_LL_Joint_FCS',
                                values={'tx': -1.5},
                                parent=a_ll_zero_ll_joint)
```

> **Warning:** Note that the second code block references the closed chain as its parent. Since this group is not yet build the code above will encounter errors. In Practice the close chain group first has to be built. The correct order of this code can be seen [here](https://github.com/TriPed-Robot/trip_kinematics/blob/main/src/trip_robots/triped_leg.py).

Note that both chains are made up of transformations without *state_variables*. Such transformations are 'static' and dont represent a joint. It is possible to construct open chains without such transformations (see the denavit hartenberg convention for example). In practice however they are handy to specify the position of a joint at a specified angle. This allows the joint angles to be interpretable. This can be seen with the *a_ll_zero* transformation. It ensures that a *extend_joint* angle corresponds to the foot of the leg being completely retracted.

Create Mappings

For the kinematic group a mapping from the actuated swing_joints to the virtual gimbal joint have to be provided. These joints can be seen down below:
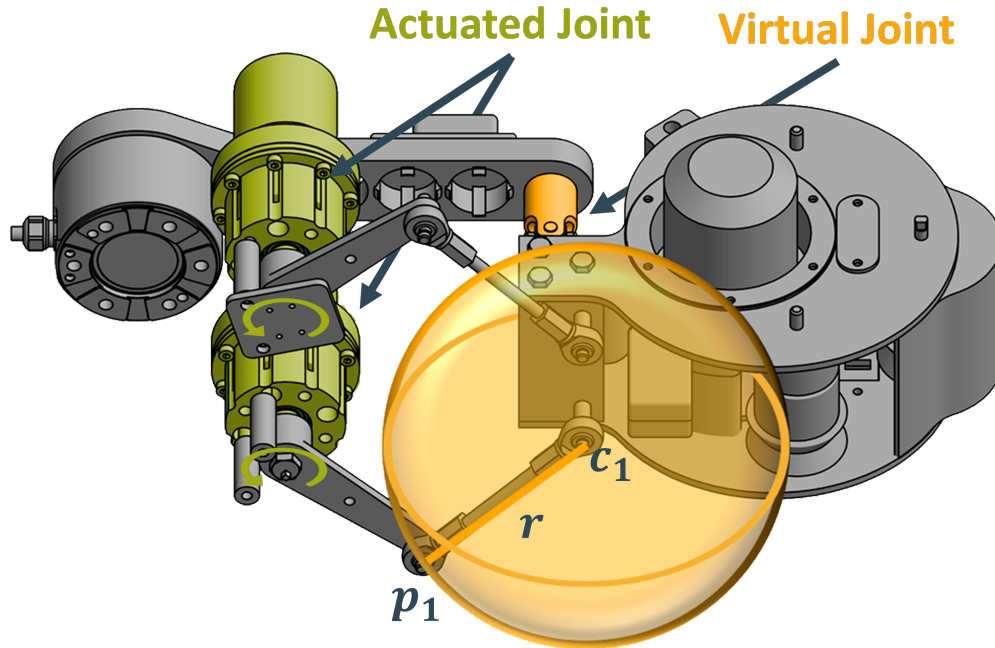
The mapping between these joints can be computet by solving a geometric closing equation. As pictured above, the tip x_i the output lever connected to the actuated joints i alwas intersects the sphere at position c_i where i is either 1 or 2.

Mathematically this can be described using:

$$\sum_{i=1}^{2} ||(c_i(rx, ry, rz^v) - p_i(rz_i^a))^T (c_i(rx, ry, rz^v) - p_i(rz_i^a)) - r^2||^2 = 0$$

Where rx, ry, rz are the rotation around the x, y and z axis and the subscript :math:`v,~a`denotes the virtual and actuated state respectively.

The virtual_to_actuated and actuated_to_virtual mappings can now be defined as the virtual or actuated state that solves the closure equation assuming the other state as a fixed value. This can be done using casadis nonlinear programming solver. The final code can be seen down below:

```
def sphere_centers(r_x, r_y, r_z):
    """Generates the centers of the spheres used for the geometric closure equation

    Args:
        r_x (float): gimbal joint state rx
        r_y (float): gimbal joint state ry
        r_z (float): gimbal joint state rz

    Returns:
        numpy array, numpy array: Two 3D arrays containing the sphere centers
    """
    a_ccs_p_trans_m = hom_translation_matrix(
```

```python
13              t_x=0.265, t_y=0, t_z=0.014)
14      a_ccs_p_rot_m = hom_rotation(x_axis_rotation_matrix(r_x) @
15                                   y_axis_rotation_matrix(r_y) @
16                                   z_axis_rotation_matrix(r_z))
17      a_p_sph_1_2 = hom_translation_matrix(
18          t_x=0.015, t_y=0.029, t_z=-0.0965)
19      a_p_sph_2_2 = hom_translation_matrix(
20          t_x=0.015, t_y=-0.029, t_z=-0.0965)
21
22      a_ccs_ = a_ccs_p_trans_m @ a_ccs_p_rot_m
23      a_c1 = a_ccs_ @ a_p_sph_1_2
24      a_c2 = a_ccs_ @ a_p_sph_2_2
25
26      return get_translation(a_c1), get_translation(a_c2)
27
28
29  def intersection_left(theta):
30      """calculates the desired sphere intersection point based on the left swing joint
31
32      Args:
33          theta (float): angle of the left swing joint
34
35      Returns:
36          numpy array: the sphere intersection
37      """
38      a_ccs_lsm_trans = hom_translation_matrix(
39          t_x=0.139807669447128, t_y=0.0549998406976098, t_z=-0.051)
40      a_ccs_lsm_rot = hom_rotation(z_axis_rotation_matrix(radians(-345.0)))
41      a_mcs_1_joint = hom_rotation(z_axis_rotation_matrix(theta))
42      a_mcs_1_sp_1_1 = hom_translation_matrix(
43          t_x=0.085, t_y=0, t_z=-0.0245)
44
45      a_ccs_sp_1_1 = a_ccs_lsm_trans @ a_ccs_lsm_rot @ a_mcs_1_joint @ a_mcs_1_sp_1_1
46      return get_translation(a_ccs_sp_1_1)
47
48
49  def intersection_right(theta):
50      """calculates the desired sphere intersection point based on the right swing joint
51
52      Args:
53          theta (float): angle of the right swing joint
54
55      Returns:
56          numpy array: the sphere intersection
57      """
58      a_ccs_rsm_tran = hom_translation_matrix(
59          t_x=0.139807669447128, t_y=-0.0549998406976098, t_z=-0.051)
60      a_ccs_rsm_rot = hom_rotation(z_axis_rotation_matrix(radians(-15.0)))
61      a_mcs_2_joint = hom_rotation(z_axis_rotation_matrix(theta))
62      a_mcs_2_sp_2_1 = hom_translation_matrix(
63          t_x=0.085, t_y=0, t_z=-0.0245)
64
```

```python
65        a_ccs_sp_2_1 = a_ccs_rsm_tran @ a_ccs_rsm_rot @ a_mcs_2_joint @ a_mcs_2_sp_2_1
66        return get_translation(a_ccs_sp_2_1)
67
68
69    def swing_to_gimbal(state: Dict[str, float], tips: Dict[str, float] = None):
70        """Actuated to virtual state mapping for the TriPed legss closed subchain
71
72        Args:
73            state (Dict[str, float]): actuated state of the TriPed leg closed subchain
74            tips (Dict[str, float], optional): Initial state for the closure equation solver.
75                                               Defaults to None in which case [0,0,0] is␣
    ↪used.
76
77        Returns:
78            Dict[str, Dict[str, float]]: the correspdonding state of the virtual chain
79        """
80        x_0 = [0, 0, 0]
81        if tips:
82            x_0[2] = tips['rx']
83            x_0[3] = tips['ry']
84            x_0[4] = tips['rz']
85
86        nlp = {'x': virtual_state, 'f': closing_equation, 'p': actuated_state}
87        nlp_solver = nlpsol('swing_to_gimbal', 'ipopt', nlp, opts)
88        solution = nlp_solver(x0=x_0,
89                              p=[state['swing_left'], state['swing_right']])
90        sol_vector = np.array(solution['x'])
91        return {'gimbal_joint': {'rx': sol_vector[0][0],
92                                 'ry': sol_vector[1][0],
93                                 'rz': sol_vector[2][0]}}
94
95
96    def gimbal_to_swing(state: Dict[str, Dict[str, float]], tips: Dict[str, float] = None):
97        """Virtual to actuated state mapping for the TriPed legss closed subchain
98
99        Args:
100            state (Dict[str, Dict[str, float]]): virtual state of the TriPed leg closed␣
    ↪subchain
101            tips (Dict[str, float], optional): Initial state for the closure equation solver.
102                                               Defaults to None in which case [0,0] is used.
103
104        Returns:
105            Dict[str, float]: the correspdonding actuated state
106        """
107        x_0 = [0, 0]
108        continuity = 0
109        if tips:
110            x_0[0] = tips['swing_left']
111            x_0[1] = tips['swing_right']
112            continuity = (x_0-actuated_state).T @ (x_0-actuated_state)
113
114        nlp = {'x': actuated_state, 'f': closing_equation+continuity, 'p': virtual_state,
```

```python
115              'g': actuated_state[0]*actuated_state[1]}
116     nlp_solver = nlpsol('gimbal_to_swing', 'ipopt', nlp, opts)
117     solution = nlp_solver(x0=x_0,
118                           p=[state['gimbal_joint']['rx'],
119                             state['gimbal_joint']['ry'],
120                             state['gimbal_joint']['rz']],
121                           ubg=0)
122     sol_vector = np.array(solution['x'])
123     return {'swing_left': sol_vector[0][0], 'swing_right': sol_vector[1][0]}
124
125
126 theta_left = SX.sym('theta_left')
127 theta_right = SX.sym('theta_right')
128 gimbal_x = SX.sym('gimbal_x')
129 gimbal_y = SX.sym('gimbal_y')
130 gimbal_z = SX.sym('gimbal_z')
131
132 virtual_state = vertcat(gimbal_x, gimbal_y, gimbal_z)
133 actuated_state = vertcat(theta_left, theta_right)
134
135 opts = {'ipopt.print_level': 0, 'print_time': 0}
136 RADIUS = 0.11
137 c1, c2 = sphere_centers(r_x=gimbal_x, r_y=gimbal_y, r_z=gimbal_z)
138 closing_equation = (((c1-intersection_left(theta_right)).T @ (c1-intersection_left(theta_
    →right)) -
139                     RADIUS**2)**2 +
140                     ((c2-intersection_right(theta_left)).T @ (c2-intersection_
    →right(theta_left)) -
141                     RADIUS**2)**2)
```

Building the group

Using both the mappings and the virtual open chain, the group can be build:

```python
1 closed_chain = KinematicGroup(name='closed_chain',
2                               virtual_chain=[a_ccs_p_trans, a_ccs_p_rot],
3                               actuated_state={
4                                   'swing_left': 0, 'swing_right': 0},
5                               actuated_to_virtual=swing_to_gimbal,
6                               virtual_to_actuated=gimbal_to_swing)
```

Note that the closed chain specifies no parent since it is located directly at the robots base and the open chain specifies no mappings since these are autogenerated.

Combining groups to a robot

The last step is to combine the group and transformations into a robot object:

```
triped_leg      = Robot([closed_chain,a_p_ll,a_ll_zero,a_ll_zero_ll_joint,a_ll_joint_fcs])
```

Building the complete robot

Since the TriPed has three legs, the above process has to be repeated three times. Each time the joints and actuated state need their own unique names.

Since this would be tedious to do by hand, a function can be written that returns the full transformation of a single leg. This functions then follows the above steps, only appending the initial kinematic group with a transformation to the start position of each leg.

This function can be seen here:

```python
def leg_model(leg_number: str):
    """Helper function that constructs each TriPed leg as a list of Transformations.

    Args:
        leg_number (str): The leg number which determins the orientation.
                          Acceptable values [0,1,2]
    """
    def rename_swing_to_gimbal(swing: Dict[str, float], tips: Dict[str, float] = None):
        swing = deepcopy(swing)
        swing['swing_left'] = swing[leg_name+'swing_left']
        swing['swing_right'] = swing[leg_name+'swing_right']
        del swing[leg_name+'swing_left']
        del swing[leg_name+'swing_right']

        if tips is not None:
            tips = deepcopy(tips)
            tips['gimbal_joint'] = tips[leg_name+'gimbal_joint']
            del tips[leg_name+'gimbal_joint']

        gimbal = swing_to_gimbal(swing, tips)

        gimbal[leg_name+'gimbal_joint'] = gimbal['gimbal_joint']
        del gimbal['gimbal_joint']

        return gimbal

    def rename_gimbal_to_swing(gimbal: Dict[str, float], tips: Dict[str, float] = None):
        gimbal = deepcopy(gimbal)
        gimbal['gimbal_joint'] = gimbal[leg_name+'gimbal_joint']
        del gimbal[leg_name+'gimbal_joint']

        if tips is not None:
            tips = deepcopy(tips)
            tips['swing_left'] = tips[leg_name+'swing_left']
            tips['swing_right'] = tips[leg_name+'swing_right']
            del tips[leg_name+'swing_left']
```

(continues on next page)

```
37              del tips[leg_name+'swing_right']
38
39          swing = gimbal_to_swing(gimbal, tips)
40
41          swing[leg_name+'swing_left'] = swing['swing_left']
42          swing[leg_name+'swing_right'] = swing['swing_right']
43          del swing['swing_left']
44          del swing['swing_right']
45
46          return swing
47
48      leg_name = 'leg_'+str(leg_number)+'_'
49
50      leg_rotation = Transformation(name=leg_name+'leg_rotation',
51                                    values={'rz': -1*radians(120)*leg_number})
52      a_ccs_p_trans = Transformation(name=leg_name+'A_ccs_P_trans',
53                                     values={'tx': 0.265, 'tz': 0.014},
54                                     parent=leg_rotation)
55      a_ccs_p_rot = Transformation(name=leg_name+'gimbal_joint',
56                                   values={'rx': 0, 'ry': 0, 'rz': 0},
57                                   state_variables=['rx', 'ry', 'rz'],
58                                   parent=a_ccs_p_trans)
59
60      closed_chain = KinematicGroup(name=leg_name+'closed_chain',
61                                    virtual_chain=[leg_rotation,
62                                                   a_ccs_p_trans, a_ccs_p_rot],
63                                    actuated_state={
64                                        leg_name+'swing_left': 0, leg_name+'swing_right':␣
    ↪0},
65                                    actuated_to_virtual=rename_swing_to_gimbal,
66                                    virtual_to_actuated=rename_gimbal_to_swing)
67
68      a_p_ll = Transformation(name=leg_name+'A_P_LL',
69                              values={'tx': 1.640, 'tz': -0.037, },
70                              parent=closed_chain)
71      a_ll_zero = Transformation(name=leg_name+'zero_angle_convention',
72                                 values={'ry': radians(-3)},
73                                 parent=a_p_ll)
74      a_ll_zero_ll_joint = Transformation(name=leg_name+'extend_joint',
75                                          values={'ry': 0},
76                                          state_variables=['ry'],
77                                          parent=a_ll_zero)
78      a_ll_joint_fcs = Transformation(name=leg_name+'A_LL_Joint_FCS',
79                                      values={'tx': -1.5},
80                                      parent=a_ll_zero_ll_joint)
81
82      return [closed_chain, a_ll_zero_ll_joint, a_ll_joint_fcs, a_p_ll, a_ll_zero]
```

The final TriPed robot can then be build using:

```
1  triped = Robot(leg_model(0)+leg_model(1)+leg_model(2))
```

1.3.2 Importing URDF Files

The Universal Robot Description Format (URDF) is used to describe a variety of serial robot mechanisms. Trip includes a URDF parser that allows the importation of alist of transfomrations from a URDF file

It can be used directly after importing the TriP library by calling the function trip_kinematics.from_urdf with the path to the URDF file as the argument. Note that the function returns a list of Transformations, which you probably want to create a Robot from in most cases:

```
transformations_list = trip_kinematics.urdf_parser.from_urdf(urdf_path)
robot = Robot(transformations_list)
```

This means you can also add other Transformations manually on top of those specified in the URDF file, if required.

Also note that the parser includes defaults for certain values if the corresponding URDF tag is missing, specifically:

- <origin> defaults to <origin xyz="0 0 0" rpy="0 0 0" />.
  - (The same also applies if only one of xyz and rpy is specified, with the omitted value defaulting to "0 0 0")
- <axis> defaults to <axis xyz="0 0 1" />

1.4 Code Documentation

class `trip_kinematics.Transformation.`**`Transformation`**(*name: str*, *values: Dict[str, float]*, *state_variables: Optional[List[str]] = None*, *parent=None*)

> Initializes the *Transformation* class.
>
> > **Parameters**
> >
> > > - **name** (`str`) – The unique name identifying the Transformation. No two *Transformation* objects of a :py:class`Robot` should have the same name
> > > - **values** (`Dict[str, float]`) – A parametric description of the transformation.
> > > - **state_variables** (`List[str], optional`) – This list describes which state variables are dynamically changable. This is the case if the *Transformation* represents a joint. Defaults to [].
> >
> > **Raises** `ValueError` – A dynamic state was declared that does not correspond to a parameter declared in `values`.
>
> **add_children**(*child: str*)
>
> > Adds the name of a `KinematicGroup` or *Transformation* as a child.
> >
> > > **Parameters** **child** (`str`) – the name of a `KinematicGroup` or *Transformation*
>
> **static get_convention**(*state: Dict[str, float]*)
>
> > Returns the connvention which describes how the matrix of a *Transformation* is build from its state.
> >
> > > **Parameters** **state** (`Dict[str, float]`) – :py:attr:'state'
> > >
> > > **Raises**
> > >
> > > > - `ValueError` – "Invalid key." If the dictionary kontains keys that dont correspond to a parameter of the transformation.

> - **ValueError** – "State can't have euler angles and quaternions!" If the dictionary contains keys correspondig to multiple mutually exclusive conventions.
>
> **Returns** A string describing the convention
>
> **Return type** [type]

get_name()

> Returns the _name of the [Transformation](#)
>
> > **Returns** a copy the _name attribute
> >
> > **Return type** str

get_state()

> **Returns a copy of the _state** attribute of the [Transformation](#) object.
>
> > **Returns** a copy of the _state
> >
> > **Return type** Dict[str,float]

get_transformation_matrix()

> **Returns a homogeneous transformation matrix build** from the `state` and `constants`
>
> > **Raises** `RuntimeError` – If the convention used in `state` is not supported. Should normally be catched during initialization.
> >
> > **Returns**
> >
> > > **A transformation matrix build using the parameters** of the [Transformation](#) state
> >
> > **Return type** [type]

set_state(*state: Dict[str, float]*)

> Sets the state of the [Transformation](#) object.
>
> > **Parameters** `state` (*Dict[str, float]*) – Dictionary with states that should be set. Does not have to be the full state.
> >
> > **Raises** `KeyError` – If a key in the argument is not valid state parameter name.

`trip_kinematics.Transformation.`**array_find**(*arr*, *obj*) → int

> **A helpher function which finds the index of an object in an array.** Instead of throwing an error when no index can be found it returns -1.
>
> > **Parameters**
> >
> > - **arr** – the array to be searched
> > - **obj** – the object whose index is to be found.
> >
> > **Returns** The index of obj in the array. -1 if the object is not in the array
> >
> > **Return type** int

class trip_kinematics.KinematicGroup.**KinematicGroup**(*name: str*, *virtual_chain:*
*List[*trip_kinematics.Transformation.Transformation*]*,
actuated_state: Dict[str, float],
actuated_to_virtual: Callable,
virtual_to_actuated: Callable,
act_to_virt_args=None, *virt_to_act_args=None*,
parent=None)

Initializes a *KinematicGroup* object.

Parameters

- **name** (*str*) – The unique name identifying the group. No two *KinematicGroup* objects of a :py:class`Robot` should have the same name

- **virtual_chain** (*List[Transformation]*) – A list of `Transformation` objects forming a serial Kinematic chain.

- **actuated_state** (*List[Dict[str, float]]*) – The State of the Groups actuated joints.

- **actuated_to_virtual** (*Callable*) – Maps the `actuated_state` to the `virtual_state` of the `virtual_chain`.

- **virtual_to_actuated** (*Callable*) – Maps the `virtual_state` of the `virtual_chain` to the `actuated_state`.

- **act_to_virt_args** (*[type]*, *optional*) – Arguments that can be passed to `actuated_to_virtual` during the initial testing of the function. Defaults to None.

- **virt_to_act_args** (*[type]*, *optional*) – Arguments that can be passed to `virtual_to_actuated` during the initial testing of the function. Defaults to None.

- **parent** (*Union(Transformation,KinematicGroup), optional*) – The transformation or group preceding the *KinematicGroup*. Defaults to None.

Raises

- **ValueError** – 'Error: Actuated state is missing. You provided a mapping to actuate the group but no state to be actuated.' if there is no `actuated_state` despite a mapping being passed

- **ValueError** – 'Error: Only one mapping provided. You need mappings for both ways. Consider to pass a trivial mapping.' if either `actuated_to_virtual` or `virtual_to_actuated` was not set despite providing a `actuated_state`.

- **ValueError** – 'Error: Mappings missing. You provided an actuated state but no mappings. If you want a trivial mapping you don't need to pass an actuated state. Trip will generate one for you.' if both `actuated_to_virtual` and `virtual_to_actuated` were not set despite providing a `actuated_state`.

- **RuntimeError** – "actuated_to_virtual does not fit virtual state" if the `actuated_to_virtual` function does not return a valid `virtual_state` dictionary.

- **RuntimeError** – "virtual_to_actuated does not fit actuated state" if the `virtual_to_actuated` function does not return a valid `actuated_state` dictionary.

add_children(*child: str*)

Adds the name of a *KinematicGroup* or `Transformation` as a child.

Parameters **child** (*str*) – the name of a *KinematicGroup* or `Transformation`

get_actuated_state()

> **Returns a copy of the `actuated_state` attribute of the [KinematicGroup] object.**
>
> > **Returns** a copy of the `actuated_state`
> >
> > **Return type** Dict[str,float]

get_name()

> Returns the _name of the [KinematicGroup]
>
> > **Returns** the _name attribute
> >
> > **Return type** str

get_transformation_matrix()

> **Calculates the full transformationmatrix from** the start of the virtual chain to its endeffector.
>
> > **Returns**
> >
> > > **The homogenous transformation matrix from** the start of the virtual chain to its endeffector.
> >
> > **Return type** array

get_virtual_chain()

> **Returns a copy of the `_virtual_chain` attribute of a [KinematicGroup] object.**
>
> > **Returns** a copy of the `_virtual_chain`
> >
> > **Return type** Dict[str,*Transformation*]

get_virtual_state()

> **Returns a copy of the `virtual_state` attribute of the [KinematicGroup] object.**
>
> > **Returns** a copy of the `virtual_state`
> >
> > **Return type** Dict[str,Dict[str,float]]

static object_list_to_key_lists(*object_lst*)

> Helper function which transforms dictionary into list of keys.
>
> > **Parameters** `object_lst` (`Dict`) – The dictionary to be transformed
> >
> > **Returns** A list of keys
> >
> > **Return type** list(str)

pass_arg_a_to_v(*argv*)

> Allows arguments to be passed the `actuated_to_virtual` mapping.
>
> > **Parameters** `argv` (`[type]`) – arguments to be passed.

pass_arg_v_to_a(*argv*)

> Allows arguments to be passed the `virtual_to_actuated` mapping.
>
> > **Parameters** `argv` (`[type]`) – arguments to be passed.

set_actuated_state(*state: Dict[str, float]*)

> **Sets the** `actuated_state` **of the Group and** automatically updates the corresponding `virtual_state`.
>
> > **Parameters state** (`Dict[str, float]`) – A dictionary containing the members of `actuated_state` that should be set.
> >
> > **Raises**
> >
> > - **RuntimeError** – if all `Transformation` objects of `_virtual_chain` are static.
> > - **ValueError** – if the state to set is not part of keys of `actuated_state`

set_virtual_state(*state: Dict[str, Dict[str, float]]*)

> **Sets the** `virtual_state` **of the Group and** automatically updates the corresponding `actuated_state`.
>
> > **Parameters state** (`Dict[str,Dict[str, float]]`) – A dictionary containing the members of `virtual_state` that should be set. The new values need to be valid state for the state of the joint.
> >
> > **Raises**
> >
> > - **RuntimeError** – if all `Transformation` objects of `_virtual_chain` are static.
> > - **ValueError** – if the state to set is not part of keys of `virtual_state`

class trip_kinematics.KinematicGroup.**OpenKinematicGroup**(*name: str, virtual_chain: List[*trip_kinematics.Transformation.Transformation*], parent=None*)

> **A subclass of the** [*KinematicGroup*] **that assumes that all states** of the virtual_chain are actuated and automatically generates mappings. Typically only used internally by the :py:class`Robot` class to convert :py:class`Transformation` objects to :py:class`KinematicGroup`s.
>
> > **Parameters**
> >
> > - **name** (`str`) – The unique name identifying the group. No two [*KinematicGroup*] objects of a :py:class`Robot` should have the same name
> > - **virtual_chain** (`List[`[`Transformation`]`]`) – A list of `Transformation` objects forming a serial Kinematic chain.
> > - **parent** (`Union(`[`Transformation`]`,`[`KinematicGroup`]`), optional`) – The transformation or group preceding the [*KinematicGroup*]. Defaults to None.

class trip_kinematics.Robot.**Robot**(*kinematic_chain: List[*trip_kinematics.KinematicGroup.KinematicGroup*]*)

A class representing the kinematic model of a robot.

> **Parameters kinematic_chain** (`List[`[`KinematicGroup`]`]`) – A list of Kinematic Groups and Transformations which make up the robot. Transformations are automatically converted to groups
>
> **Raises**
>
> - **KeyError** – "More than one robot actuator has the same name! Please give each actuator a unique name" if there are actuated states with the same names between the :py:class`KinematicGroup` objects of the :py:class`Robot`

- **KeyError** – if there are joints with the same names between the :py:class`KinematicGroup` objects of the :py:class`Robot`

get_actuated_state()

> **Returns the actuated state of the :py:class`Robot` comprised** of the actuated states of the individual :py:class`KinematicGroup`.
>
> > **Returns** combined actuated state of all :py:class`KinematicGroup` objects.
> >
> > **Return type** Dict[str, float]

get_endeffectors()

> **Returns a list of possible endeffectors.** These are the names of all `KinematicGroup` objects. Since Transformations are internally converted to Groups, this includes the names of all Transformations.
>
> > **Returns** list of possible endeffectors.
> >
> > **Return type** list(str)

get_groups()

> **Returns a dictionary of the py:class`KinematicGroup` managed by the :py:class`Robot`_** Since Transformations are internally converted to Groups, this also returns all Transformations.
>
> > **Returns** The dictionary of py:class`KinematicGroup` objects.
> >
> > **Return type** Dict[str, *KinematicGroup*]

get_symbolic_rep(*endeffector: str*)

> This Function returnes a symbolic representation of the virtual chain.
>
> > **Parameters endeffector** (`str`) – The name of the group whose virtual chain models the desired endeffector
> >
> > **Raises KeyError** – If the endeffector argument is not the name of a transformation or group
> >
> > **Returns** A 4x4 symbolic casadi matrix containing the transformation from base to endeffector
> >
> > **Return type** SX

get_virtual_state()

> **Returns the virtual state of the :py:class`Robot` comprised** of the virtual states of the individual :py:class`KinematicGroup`.
>
> > **Returns**
> >
> > > **combined virtual state of all** :py:class`KinematicGroup` objects.
> >
> > **Return type** Dict[str,Dict[str, float]]

pass_group_arg_a_to_v(*argv_dict*)

> **Passes optional actuated_to_virtual mapping arguments** to :py:class`KinematicGroup` objects of the robot.
>
> > **Parameters argv_dict** (`Dict`) – A dictionary containing the mapping arguments keyed with the :py:class`KinematicGroup` names.

Raises `KeyError` – If no group with the name given in the argument is part of the robot.

pass_group_arg_v_to_a(*argv_dict: Dict*)

> **Passes optional virtual_to_actuated mapping arguments** to :py:class`KinematicGroup` objects of the robot.
>
> > **Parameters** `argv_dict` (`Dict`) – A dictionary containing the mapping arguments keyed with the :py:class`KinematicGroup` names.
> >
> > **Raises** `KeyError` – If no group with the name given in the argument is part of the robot.

set_actuated_state(*state: Dict[str, float]*)

> Sets the virtual state of multiple actuated joints of the robot.
>
> > **Parameters** `state` (`Dict[str, float]`) – A dictionary containing the members of `__actuated_state` that should be set.

set_virtual_state(*state: Dict[str, Dict[str, float]]*)

> Sets the virtual state of multiple virtual joints of the robot.
>
> > **Parameters** `state` (`Dict[str,Dict[str, float]]`) –
> >
> > **A dictionary containing the members of** `__virtual_state` that should be set.
> >
> > The new values need to be valid state for the state of the joint.

trip_kinematics.Robot.**forward_kinematics**(*robot:* trip_kinematics.Robot.Robot, *endeffector*)

> Calculates a robots transformation from base to endeffector using its current state
>
> > **Parameters** `robot` (Robot) – The robot for which the forward kinematics should be computed
> >
> > **Returns** The Transformation from base to endeffector
> >
> > **Return type** numpy.array

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t