# NOTICE OF CONFIDENTIALITY AND USE RESTRICTIONS

This PowerPoint presentation, including all of its contents, is the intellectual property of Pi Thanacha Choopojcharoen. It has been exclusively designed and created for the "RobotDevOps" class.

Authorized Users:

If you are currently enrolled in or have previously taken the "RobotDevOps" class and are in possession of this presentation, you acknowledge and agree to the following:

a) This presentation is provided to you solely for your personal reference and educational use related to the "RobotDevOps" class.

b) You do not have the right to reproduce, distribute, share, or disseminate this presentation, in whole or in part, to any third parties, especially those who have not registered for the "RobotDevOps" class, unless explicitly authorized by Pi Thanacha Choopojcharoen.

c) Any unauthorized use or distribution of this presentation will result in legal action and potential academic sanctions.

Unauthorized Possession:

a) If you have come into possession of this presentation but have not registered for the "RobotDevOps" class: You must delete this presentation immediately.

b) You do not have any rights to view, use, copy, or distribute this presentation.

c) Any unauthorized retention or use will result in legal action.

Your respect for intellectual property rights ensures a fair academic and professional environment for all. Violation of these terms is a breach of trust and may have serious legal and academic consequences.

# RBE502: RobotDevOps

## Robot Modelling

by

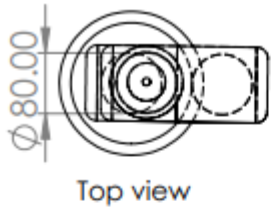
Pi Thanacha Choopojcharoen

# Agenda

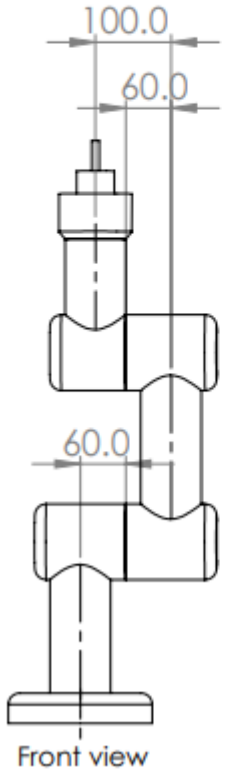
- purpose
- interface
- coordinate frame
- visualization
- URDF
- XACRO
- Frame prefix

# Exercise

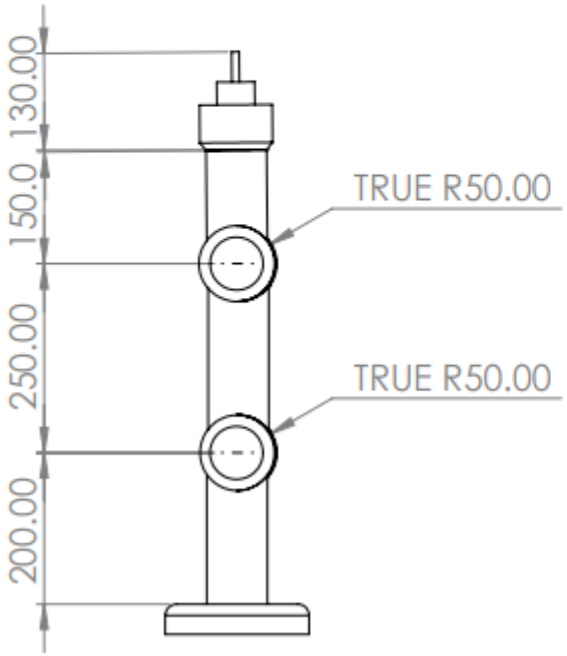
# Exercise



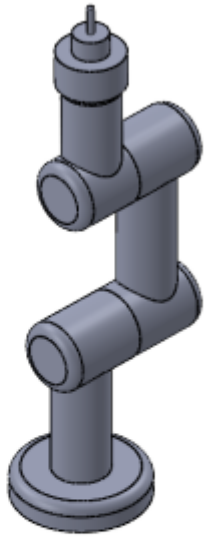

Top view



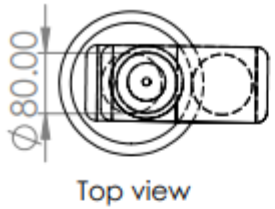

Front view



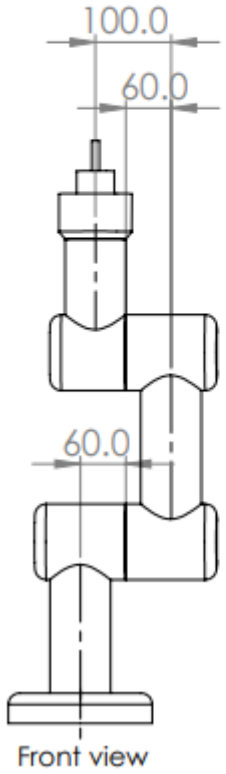

Side view



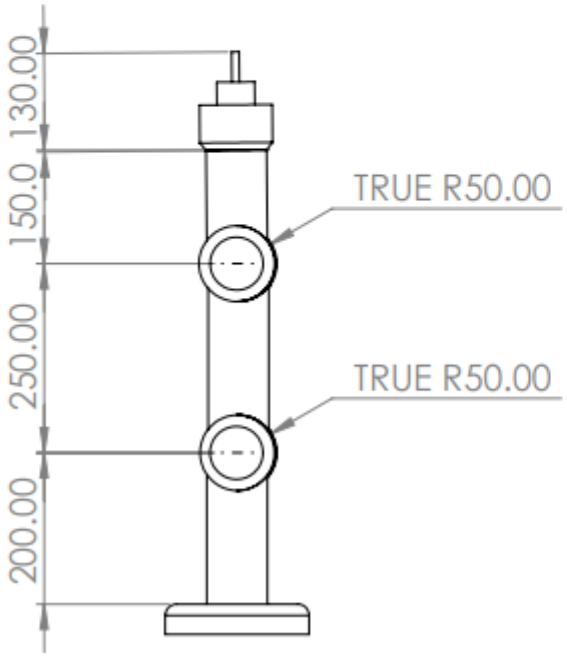
# Exercise



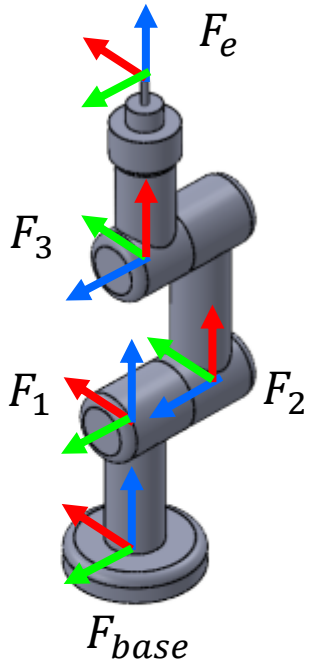

Top view



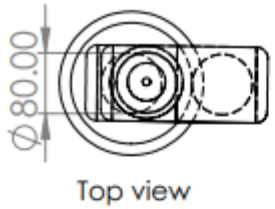

Front view

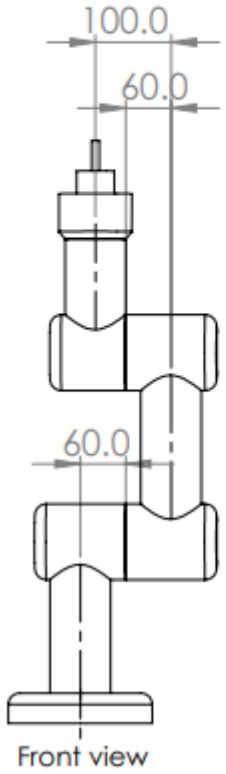


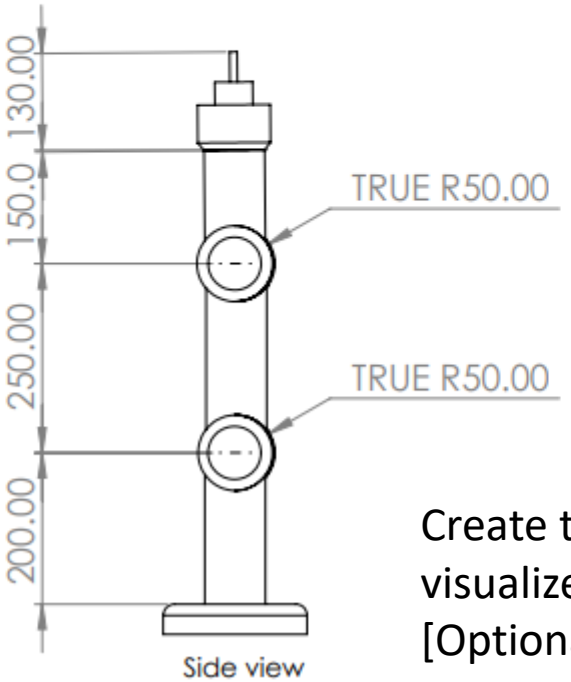

Side view

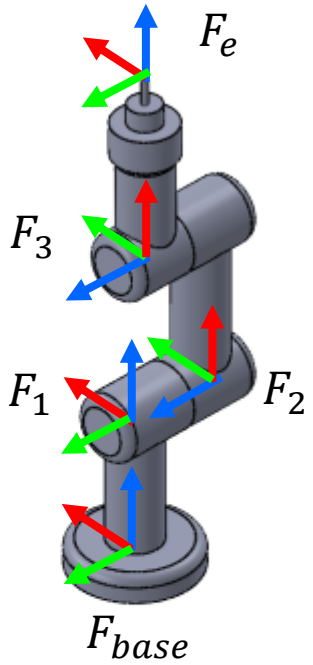


# Exercise




Create the robot “kinematics” model and visualize it in rviz.
[Optional] It should take namespace as well.

# Example Repo- Branch: robot-modelling-exercise

https://github.com/kittinook/FRA501/tree/robot-modelling-exercise

Download & add “example_description” to the “src” directory of your workspace. Then build these packages.

Follow the instruction on README.md

# Existing Packages

# Type of ROS2 Packages

Binary Package

Source Package

# Type of ROS2 Packages

Binary Package

Source Package

Installing aruco_msgs package:

>> sudo apt install ros-humble-aruco-msgs



All underscore must be changed to hyphen.

# Type of ROS2 Packages

## Binary Package

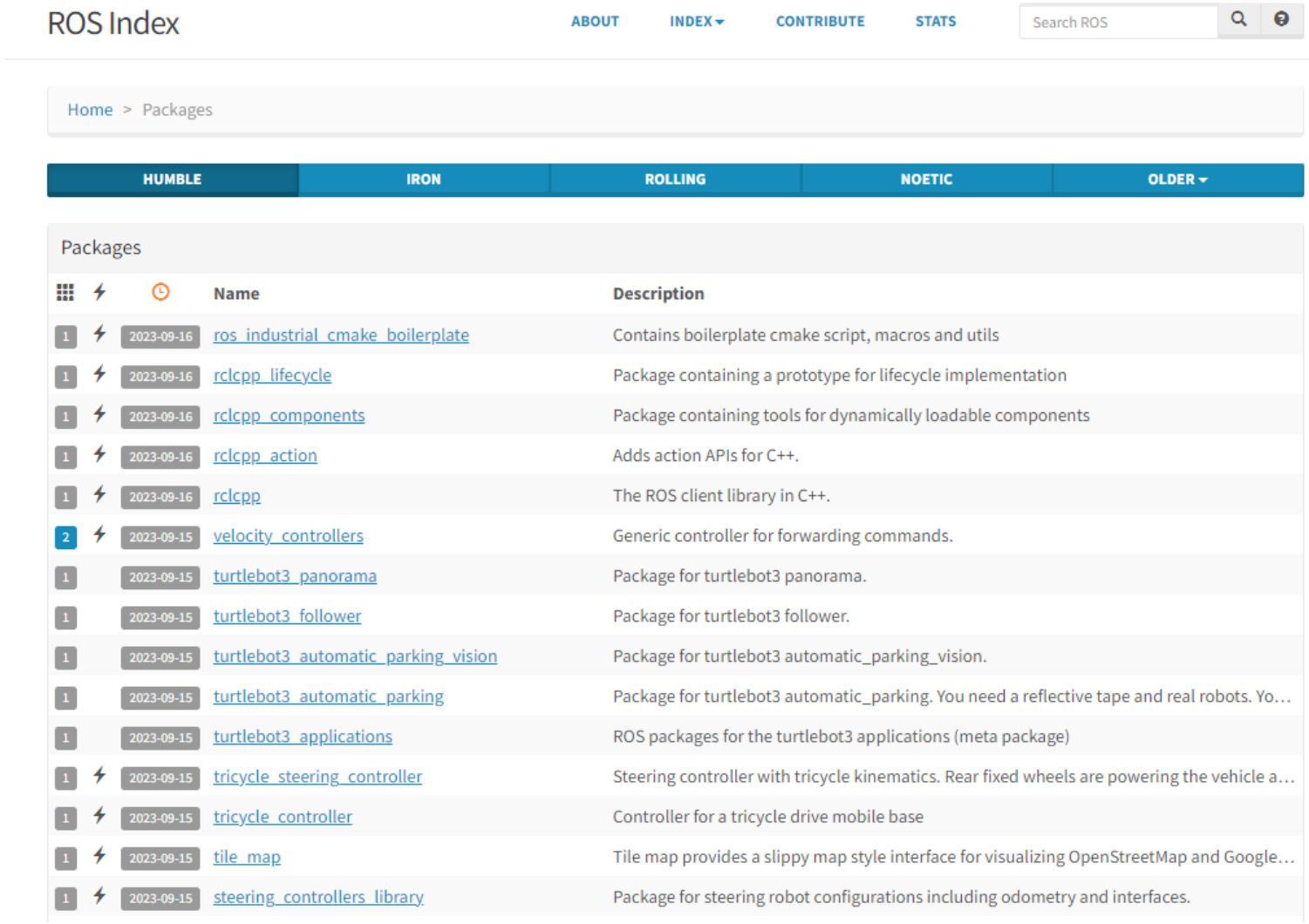
Installing aruco_msgs package:

>> sudo apt install ros-humble-aruco-msgs

## Source Package

Clone the package to “src” folder of the workspace and treat it as any other custom package.

# What binary packages are available?




https://index.ros.org/packages

>> apt search ros-humble-

Existing Packages

# Available & Installed Packages

Check package's existence in the binary package index

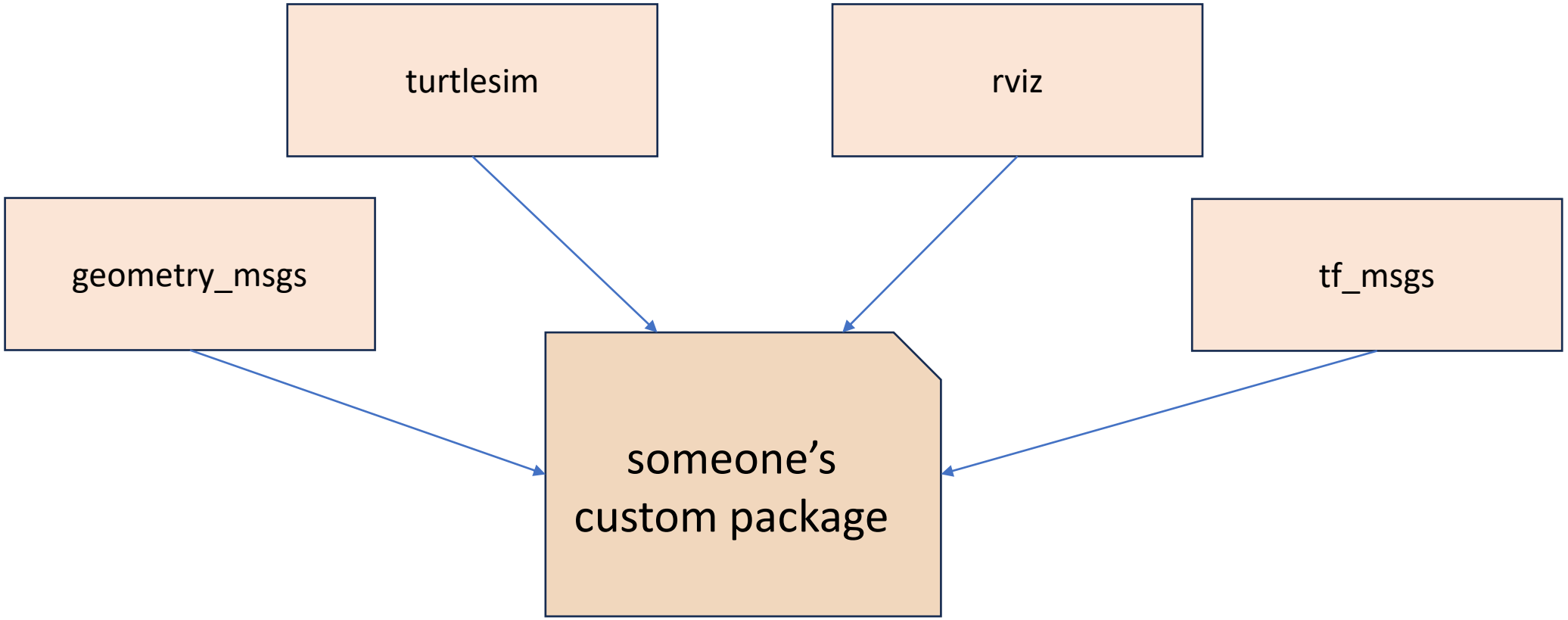
>> apt list ros-humble-aruco*

"aruco*" means any package with the name starts with "aruco" and ends with anything.

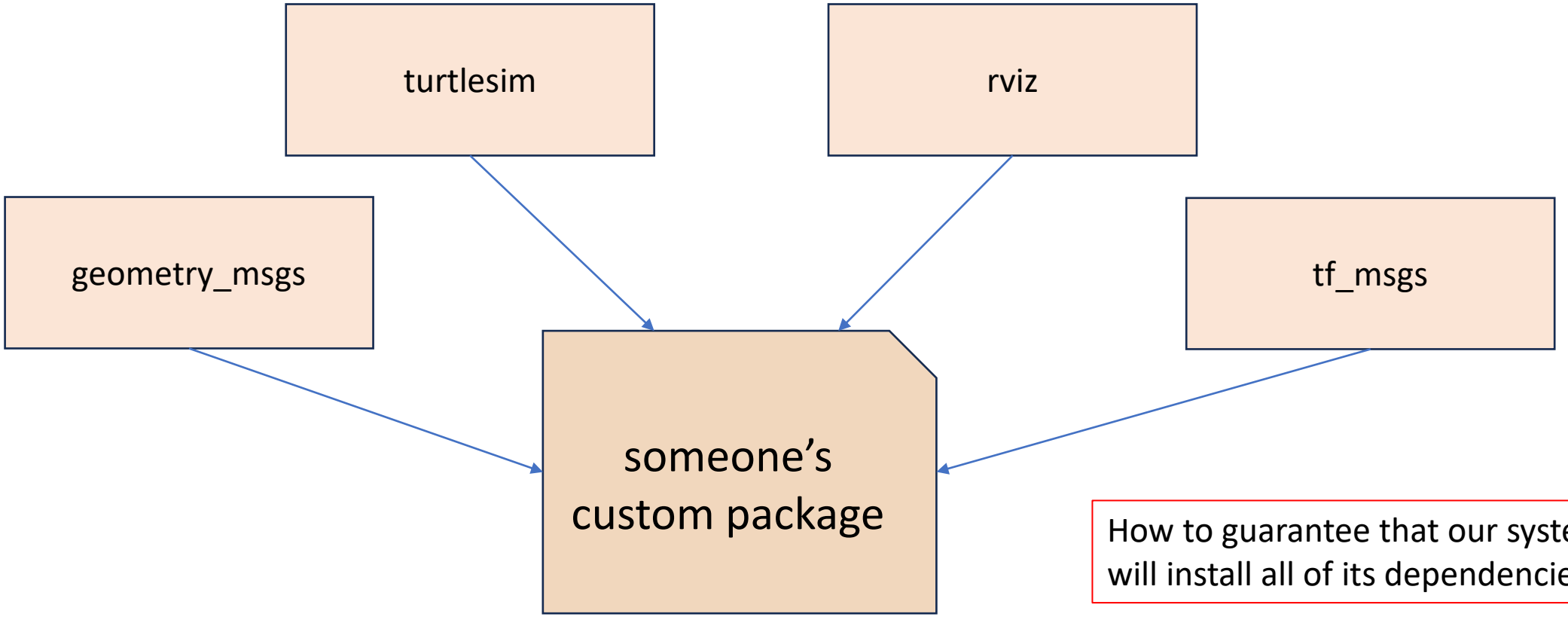
Check installed packages on the computer

>> apt list --installed ros-humble-aruco*

# Package Dependencies



# Package Dependencies




How to guarantee that our system will install all of its dependencies?

# Package Dependencies

TC

I've created a ROS2 custom package which depends on several binary packages. How do I make sure that those who downloaded my package will install all of its dependencies without having to lookup its dependencies? Give me a step-by-step instruction for me and for the downloader.

To ensure that users who download your custom ROS 2 package can easily install its dependencies without manually looking them up, you can follow these steps to specify the dependencies in your package's `package.xml` and `CMakeLists.txt` files. This will allow users to use ROS 2's package management tools to automatically install the required dependencies. Here's a step-by-step guide for both the package creator (you) and the package downloader:

### For the Package Creator (You):

1. **Identify Dependencies**:
   - Determine which binary packages your custom ROS 2 package depends on. You should know the names of these packages.
2. **Update package.xml**:
   - Open the `package.xml` file of your custom package. This file is located in the root directory of your package.
   - Add `<depend>` tags for each binary package your package depends on. Replace `<depend>` with the name of the binary package. For example:

```xml
<build_depend>ros-<distro>-package1</build_depend>
<build_depend>ros-<distro>-package2</build_depend>
<exec_depend>ros-<distro>-package3</exec_depend>
```

   Replace `<distro>` with the ROS 2 distribution you are using (e.g., `foxy`, `galactic`, etc.), and replace `package1`, `package2`, etc., with the actual names of the binary packages your package depends on.
3. **Update CMakeLists.txt**:
   - Open the `CMakeLists.txt` file of your custom package.
   - If you haven't already, include the `ament_cmake_ros` package by adding the following lines at the top of the file:

```cmake
find_package(ament_cmake REQUIRED)
find_package(<dependency_name> REQUIRED)
```

   Replace `<dependency_name>` with the name of your first binary dependency.

### For the Package Downloader:

1. **Clone or Download Package**:
   - Clone or download the custom ROS 2 package from its repository to your ROS 2 workspace's `src` directory.
2. **Build and Install**:
   - Navigate to your ROS 2 workspace's root directory and build the package along with its dependencies using `colcon`:

```csharp
colcon build --packages-select <your_package_name>
```

3. **Install Missing Dependencies**:
   - If `colcon` reports any missing dependencies during the build, you can use ROS 2's package manager (`rosdep`) to automatically install them. Run the following command:

```css
rosdep install --from-paths src --ignore-src --rosdistro <distro> -y
```

   Replace `<distro>` with the ROS 2 distribution you are using (e.g., `foxy`, `galactic`, etc.).

Regenerate

80 % Accurate

Existing Packages

# Setting Package Dependencies

In "**package.xml**" of your custom package, add <depends> tags for each binary package.

<depend> geometry_msgs </depend>
<depend> turtlesim </depend>

https://docs.ros.org/en/humble/Tutorials/Intermediate/Rosdep.html

# Installing Package Dependencies

Run the following command to install the dependency of everything in the "src" folder of your workspace.

>> rosdep install --from-paths [path to your src]/src -y --ignore-src

https://docs.ros.org/en/humble/Tutorials/Intermediate/Rosdep.html

# Today's packages

Additional packages:

ros-humble-tf2-tools
ros-humble-xacro
ros-humble-joint-state-publisher-gui

>> sudo apt install ros-humble-[package name]

https://docs.ros.org/en/humble/Tutorials/Intermediate/Rosdep.html

# Robot Modelling

# Questions regarding “Robot Modelling”

***What*** is “Robot Modelling”?

***Why*** do we need “Robot Modelling”?

***When*** do we need “Robot Modelling”?

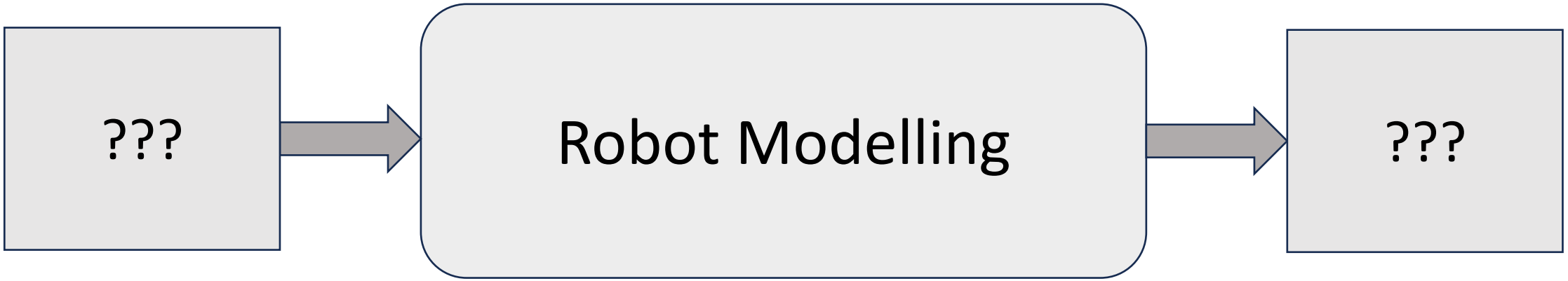
***Who*** models robots? ***Who*** need “Robot Modelling”?

***Where*** do we do “Robot Modelling”?

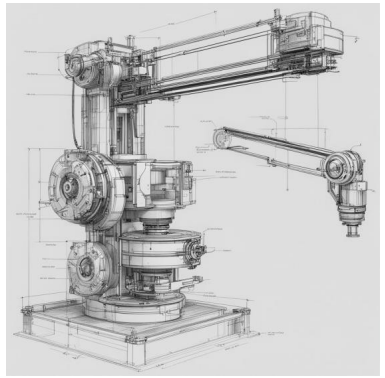
***How*** do we model robots?

# What is Robot Modelling?

# What is Robot Modelling?



# What is Robot Modelling?



Concepts

Vague, Uncertain,
Some ideas

Robot Modelling

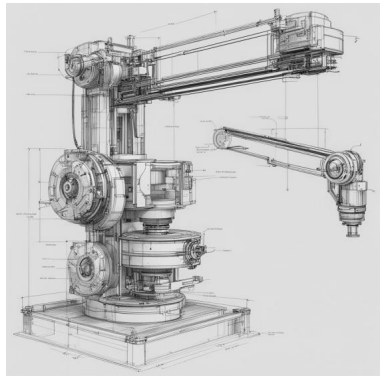
???

Prototype

Experimental,
ideas

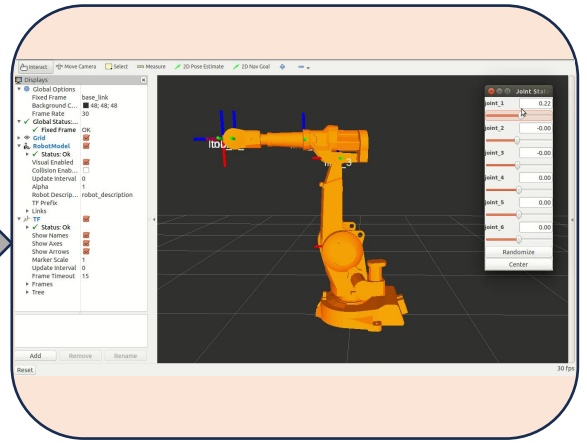
Robot Modelling

# What is Robot Modelling?



Concepts

Robot Model



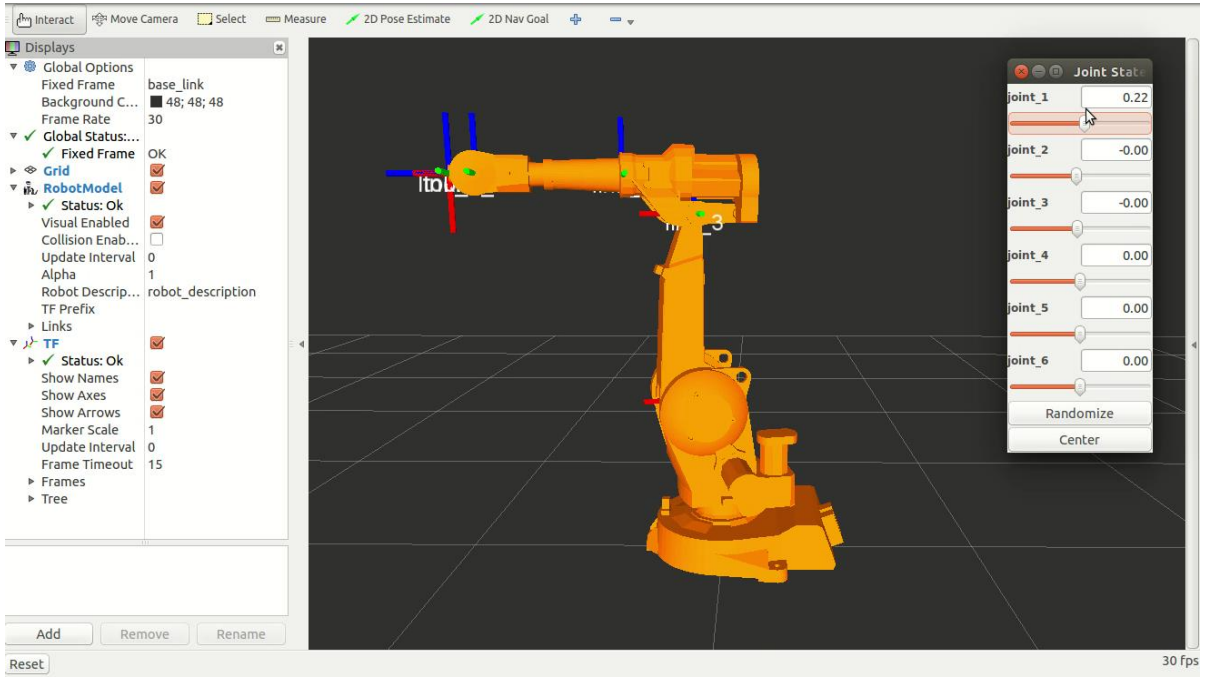
Robot Modelling

Prototype

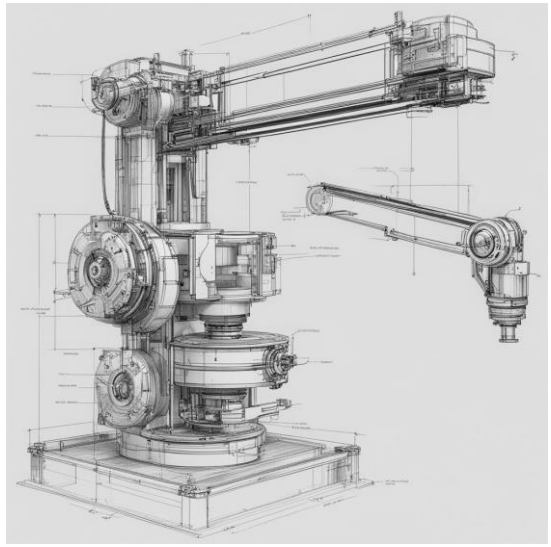
Rigorous, Based-on mathematics,
Often implemented in code

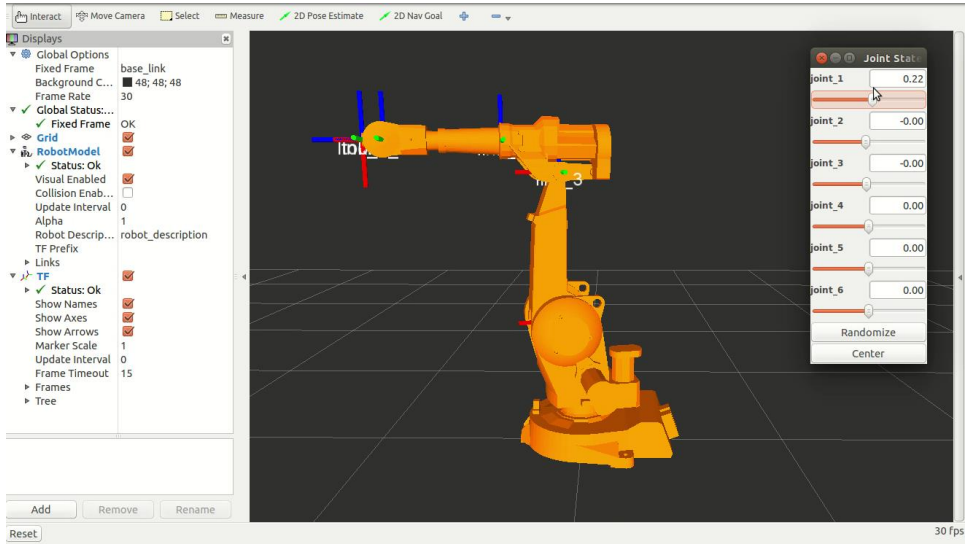
Robot Modelling

# What are robot models? And why do we need them ?

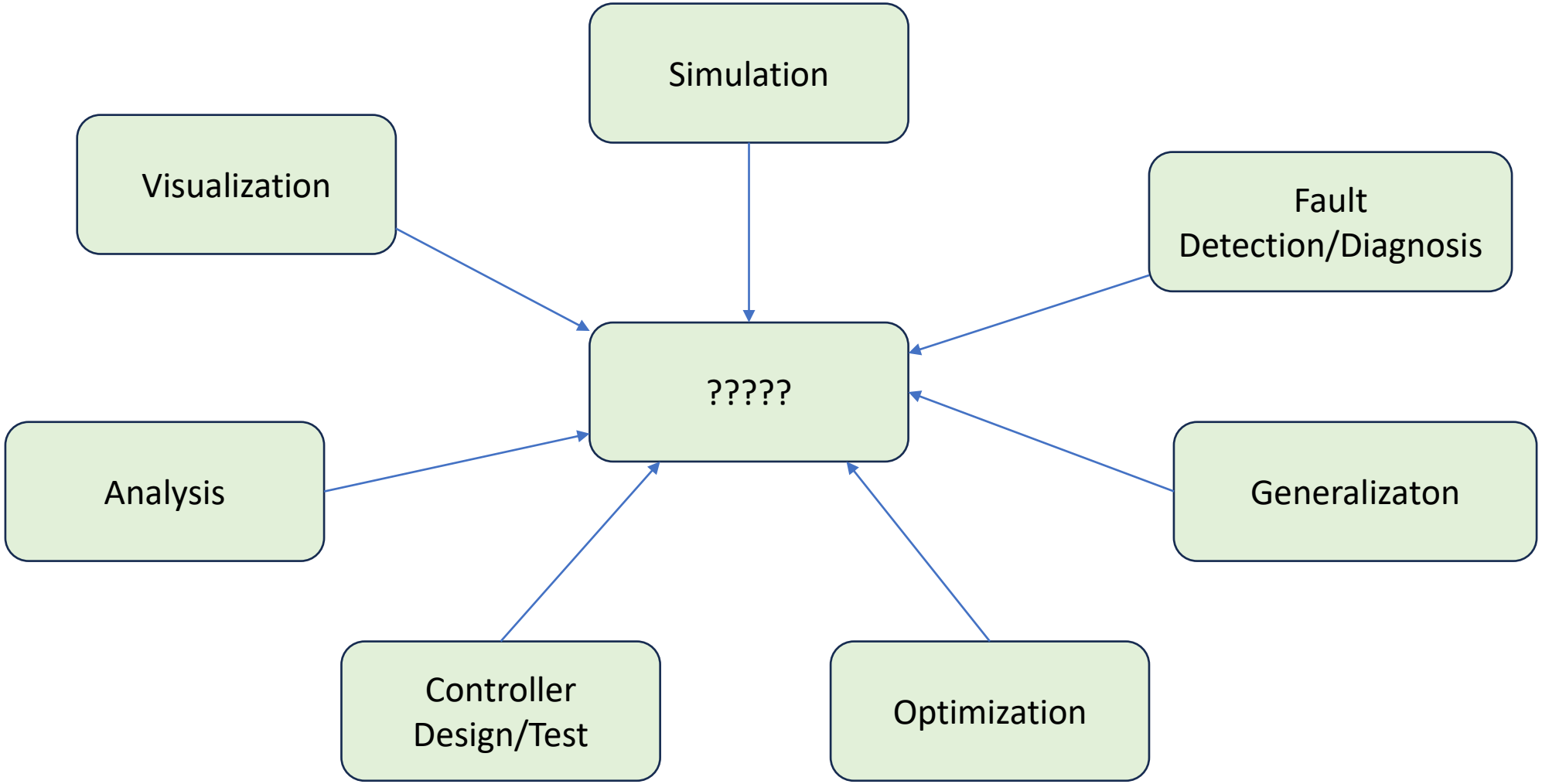


# What is it that the model can do, but the concept or a prototype cannot?

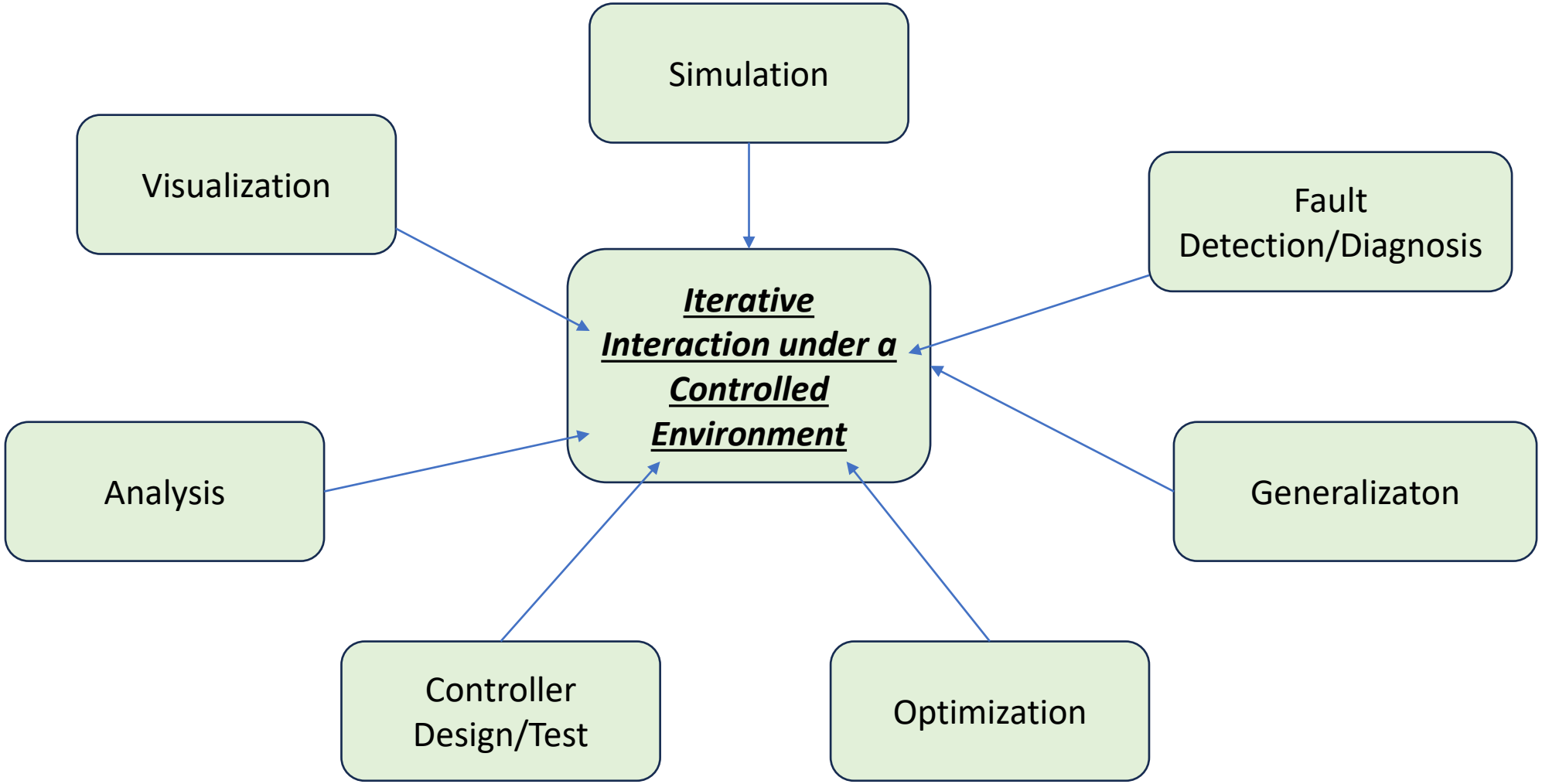




# Why?

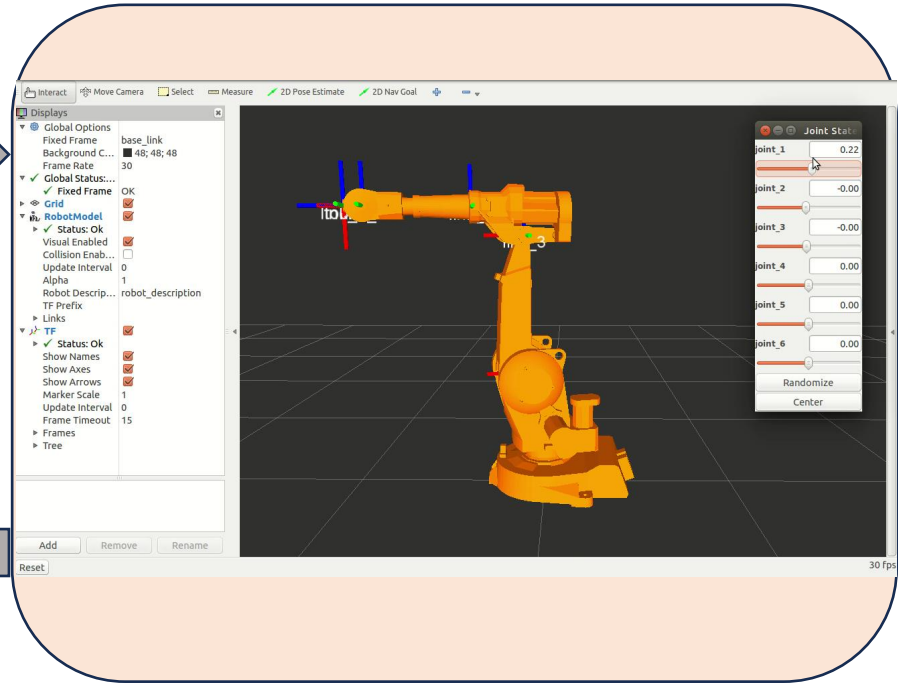


# Why?



# When do we need to “Robot Modelling”?

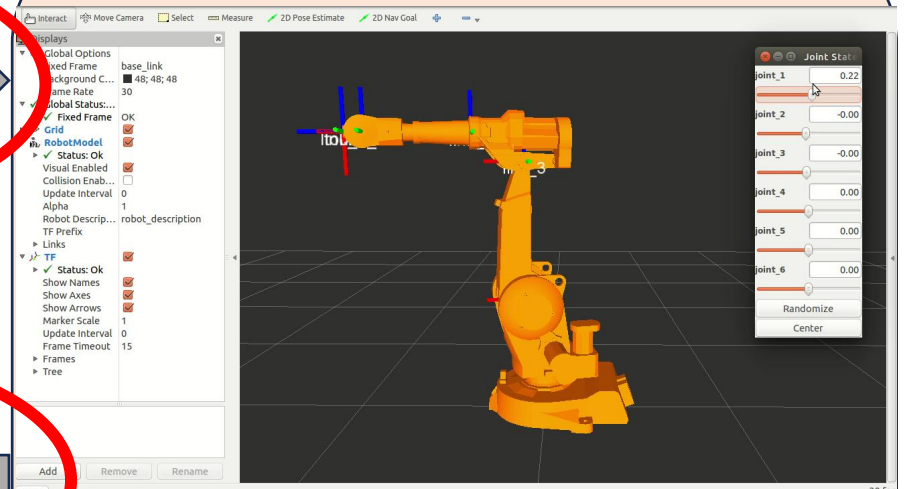
User:
Developers
Engineers




When the user requires “Iterative Interaction under a Controlled Environment”.

# What is a “robot model”?

Users
Developers
Engineers




We must identify these interfaces before “shaping” our robot model.

# Interface for a Robot Model

# What do you want from the model?

# What do you want ***from*** the “interaction”?

- Observe the position of the end-effector
- Observe how fast something is going
- Measure the distance travelled
- Measure the actuated torque at certain revolute joints
- Measure reaction force between the contact point and the ground
- Obtain an ideal sensor measurement from the model
- Observe any collision between the robot and the obstacle

# What do you want from the “interaction”?

- Observe the position of the end-effector
- Observe how fast something is going
- Measure the distance travelled
- Measure the actuated torque at certain revolute joints
- Measure reaction force between the contact point and the ground
- Obtain an ideal sensor measurement from the model
- Observe any collision between the robot and the obstacle

# Physical Quantities/Variables

## Motion

- Pose
  - Joint
  - Orientation
  - Position
- Velocity
  - Joint
  - Angular
  - Linear
- Acceleration
  - Joint
  - Angular
  - Linear

## Wrench

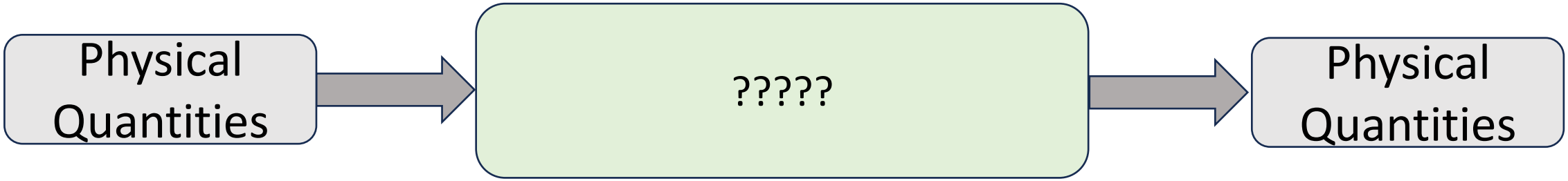
- Actuator Effort
- Moment
- Force

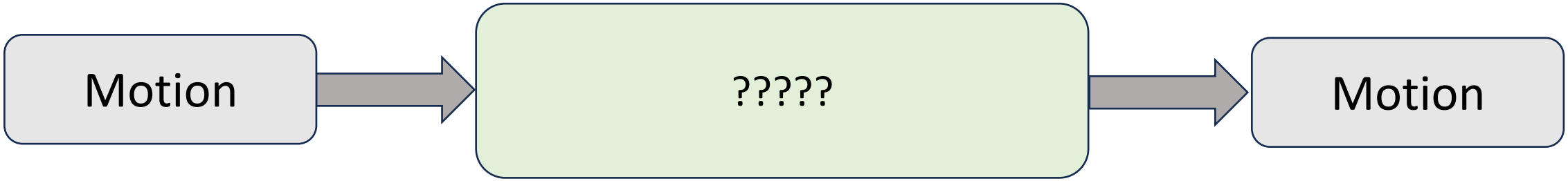
# What can you do ***to*** the “interaction”?

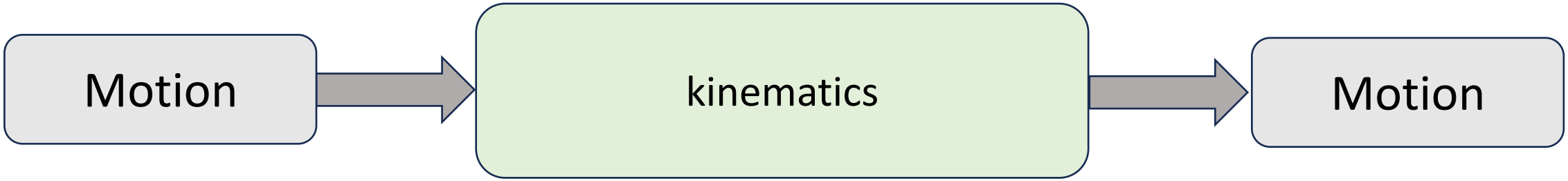
# What can you do ***to*** the “interaction”?

- Jog each joint separately
- Move the end-effector in cartesian space
- Actuate torque at the joint’s level
- Give a motion trajectory for the robot to follow
- Given an external force to the chassis of a robot

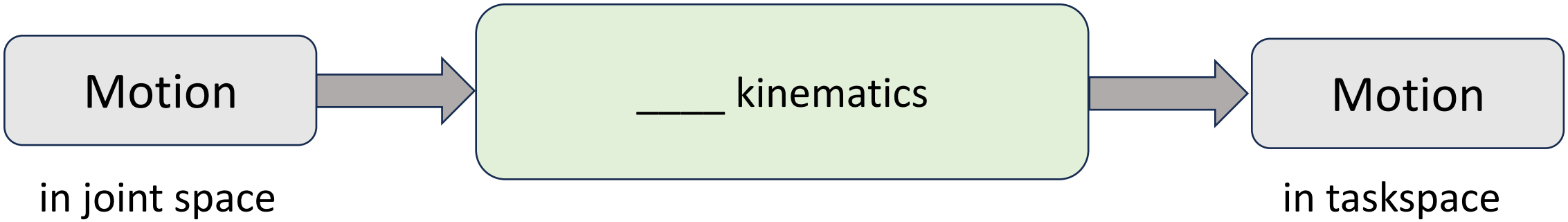
# Giving & Getting Physical Quantities

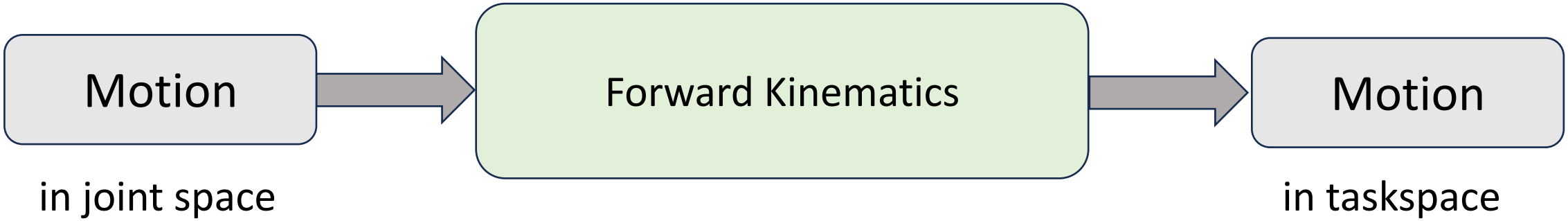


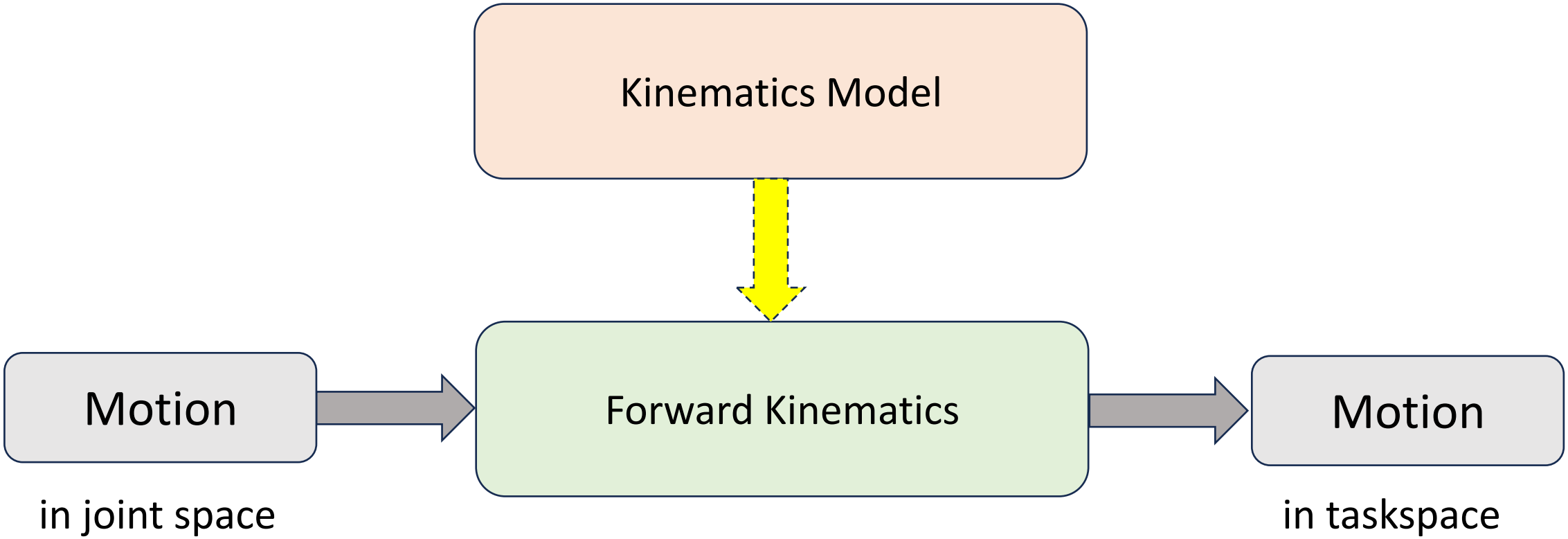


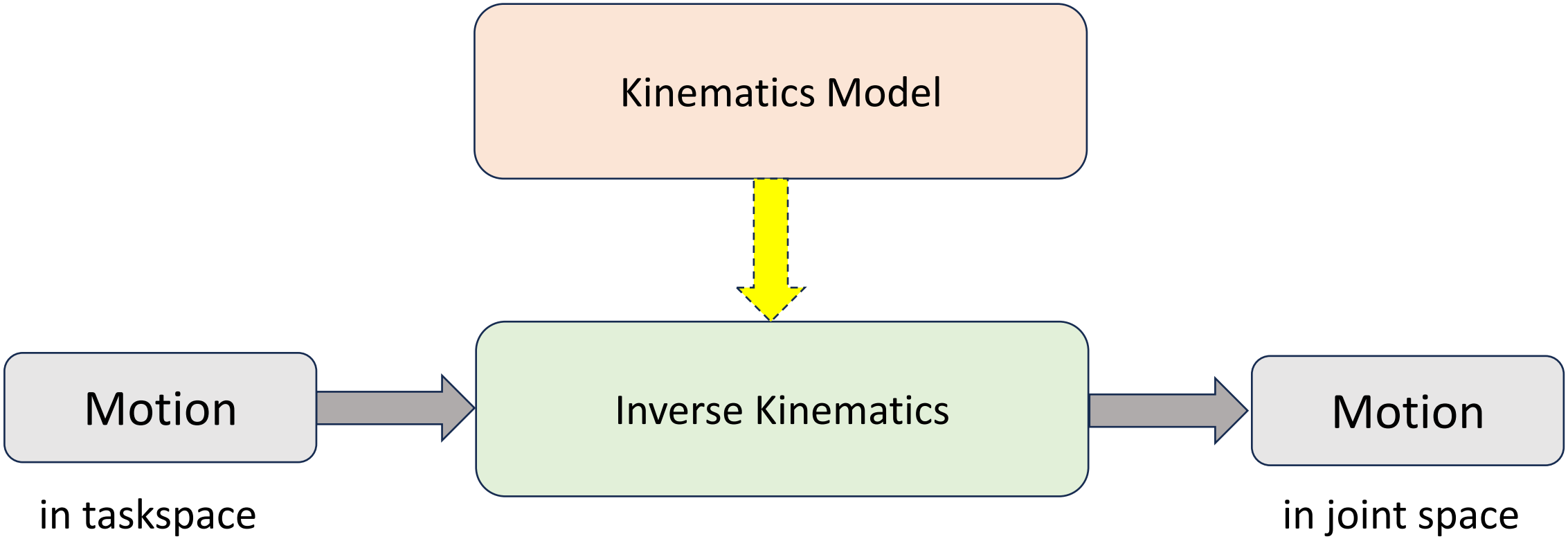


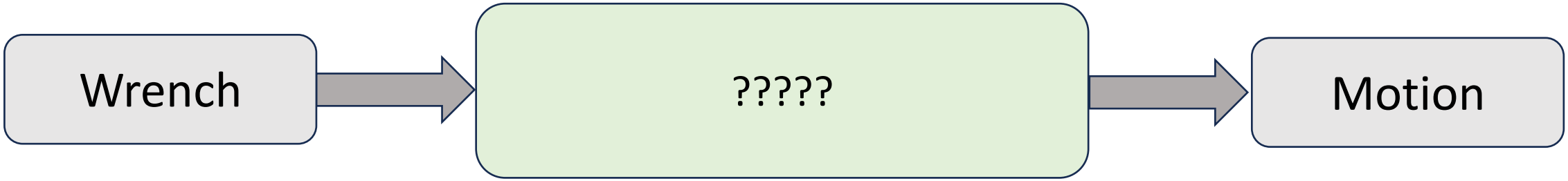

Kinematics: study of motion regardless of force

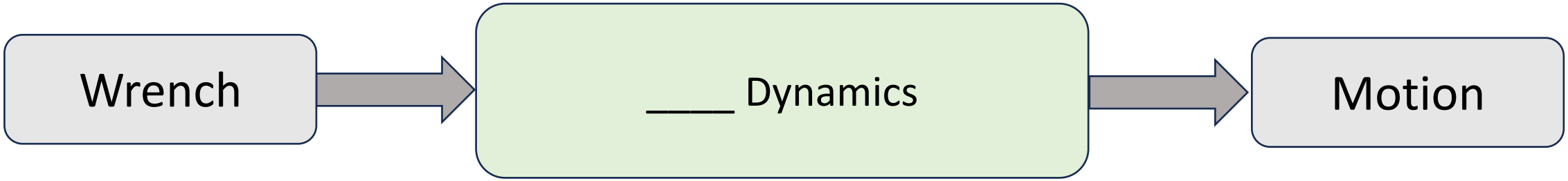


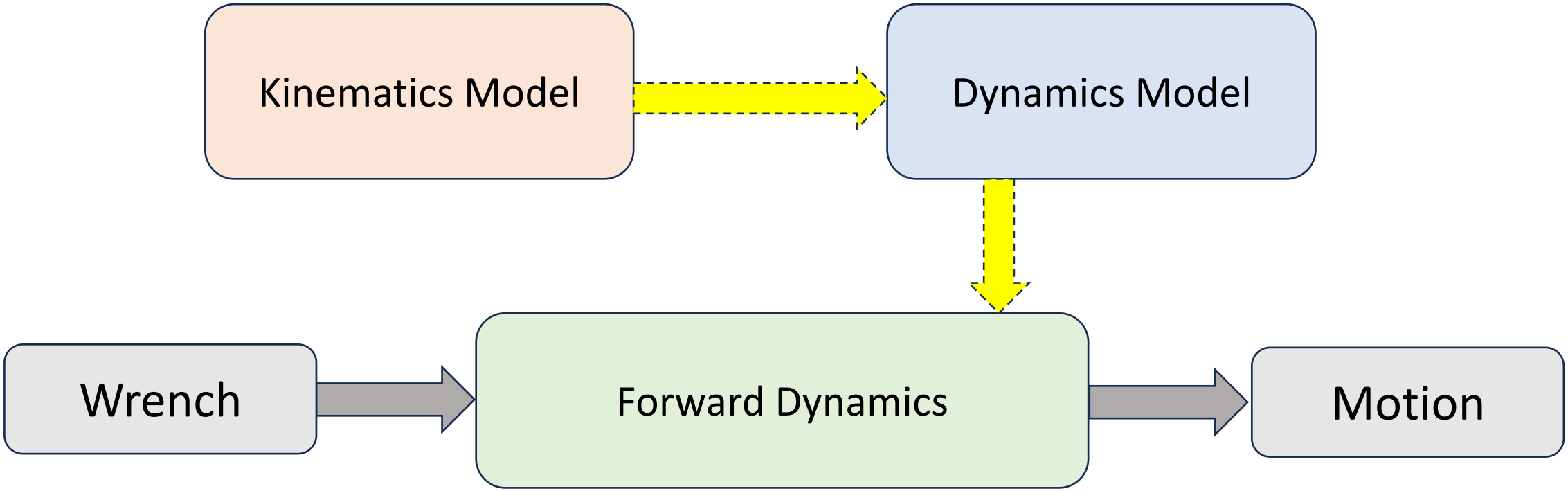


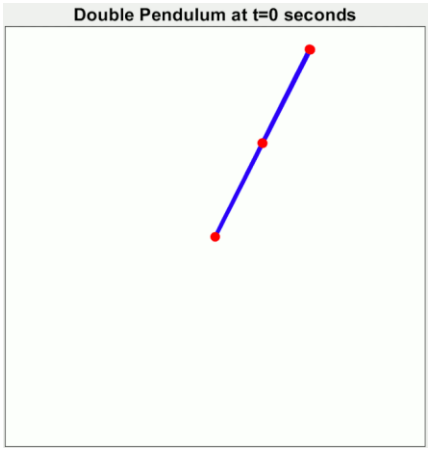


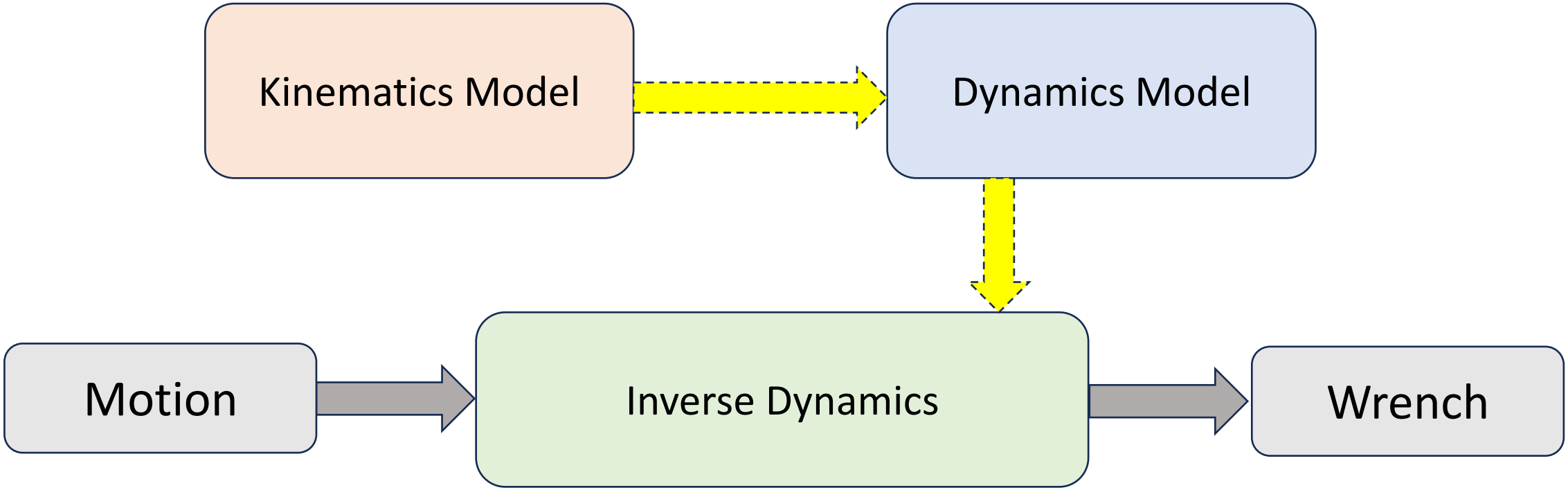








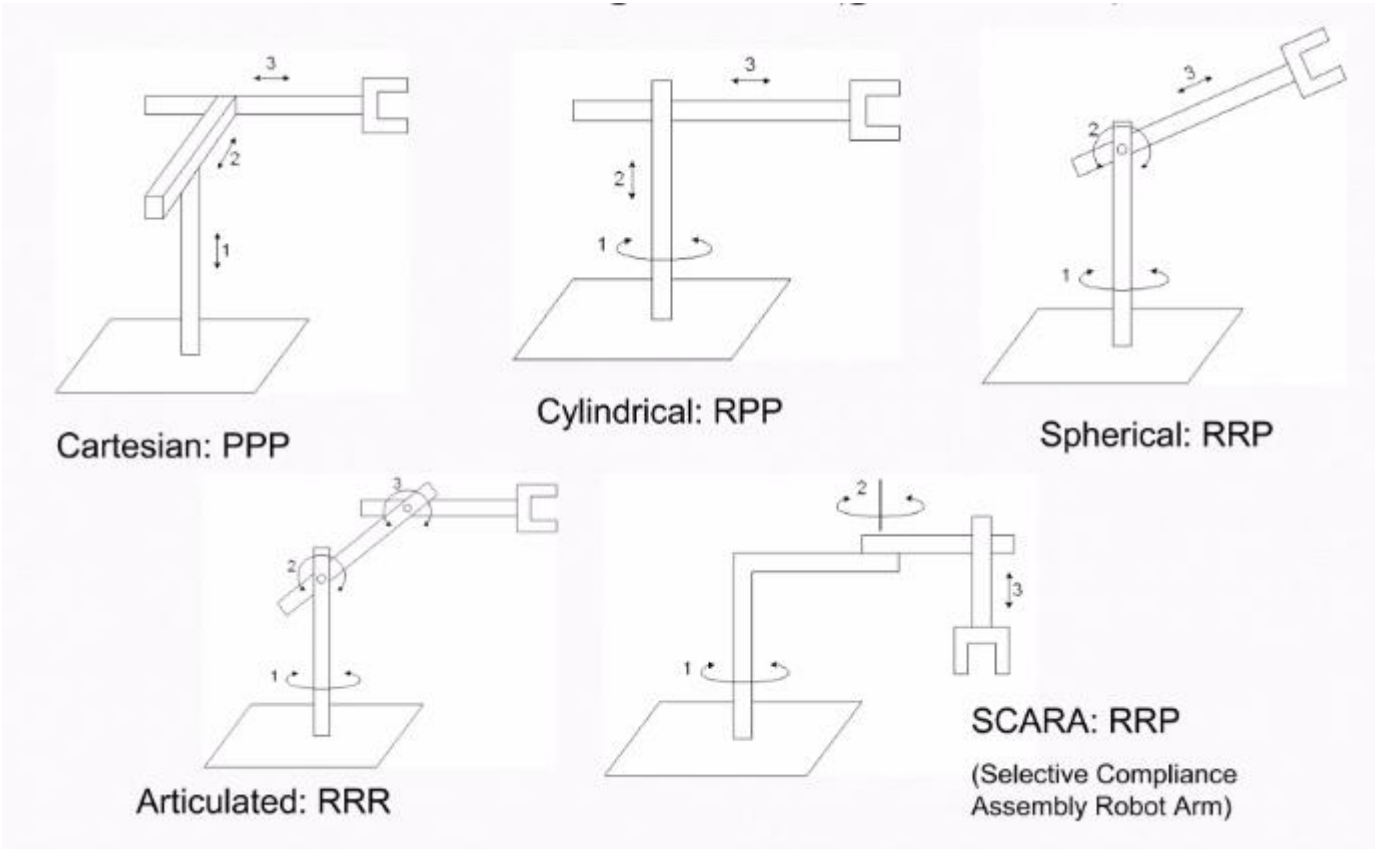





# Robot Model: Kinematics & Dynamics

Kinematics Model

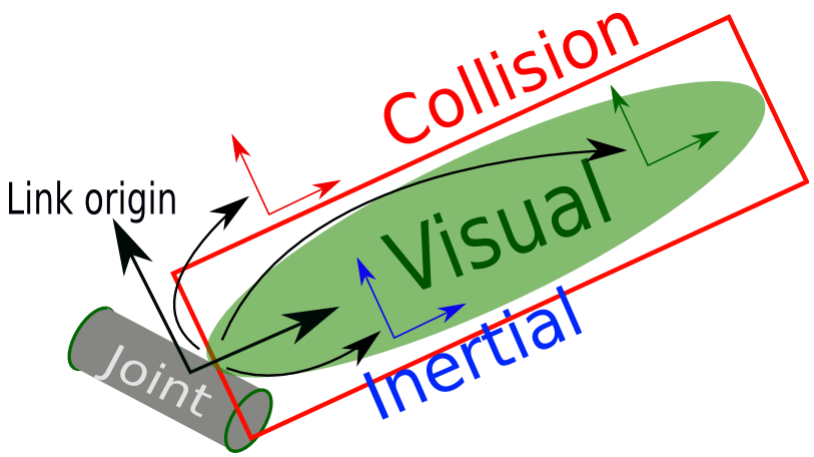
Dynamics Model



# Rigid Body & Coordinate Frame

- To describe the physical properties of a rigid body, one needs at least one reference “coordinate frame”
- This frame is known as the “body-attached” frame or “body” frame.

Q.) Does a coordinate frame need to have a body?



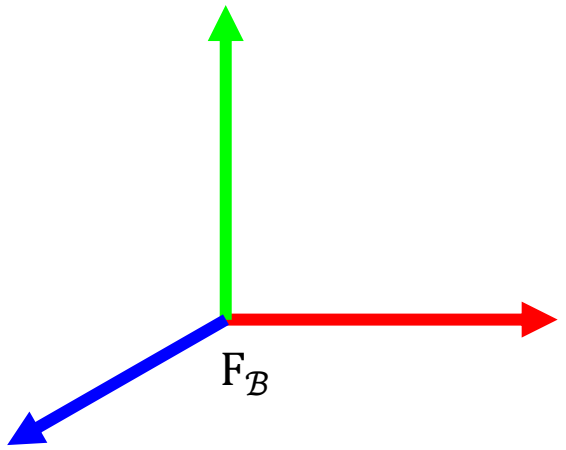
# Tracking a pose

We can track a pose of an entire body by only tracking a pose of its **body frame**.

# Coordinate Frame

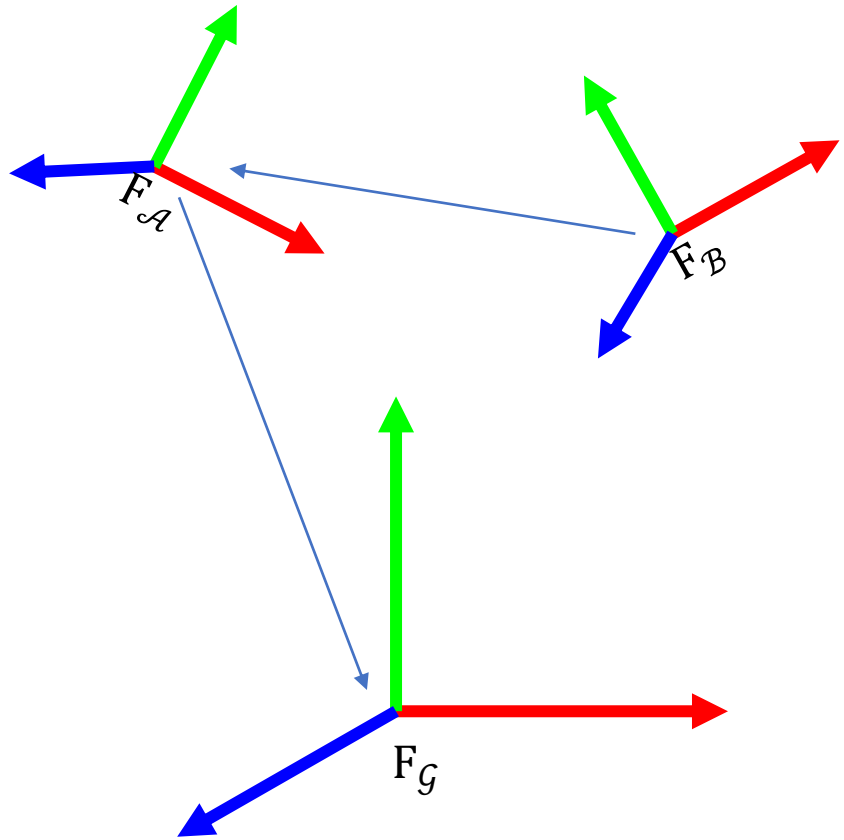
Coordinate Frame consists of

- an origin that describes its position
- 3 orthogonal axes that describes its orientation



# Global Simulation Frame

Visualizer (rviz2) or Simulator (gazebo) has its main coordinate frame when displaying its content. This frame is known as "Global (Simulation) Frame"




- Global Frame is mainly used by the GUI
- Not every frames has to refer to this frame.

# Transform

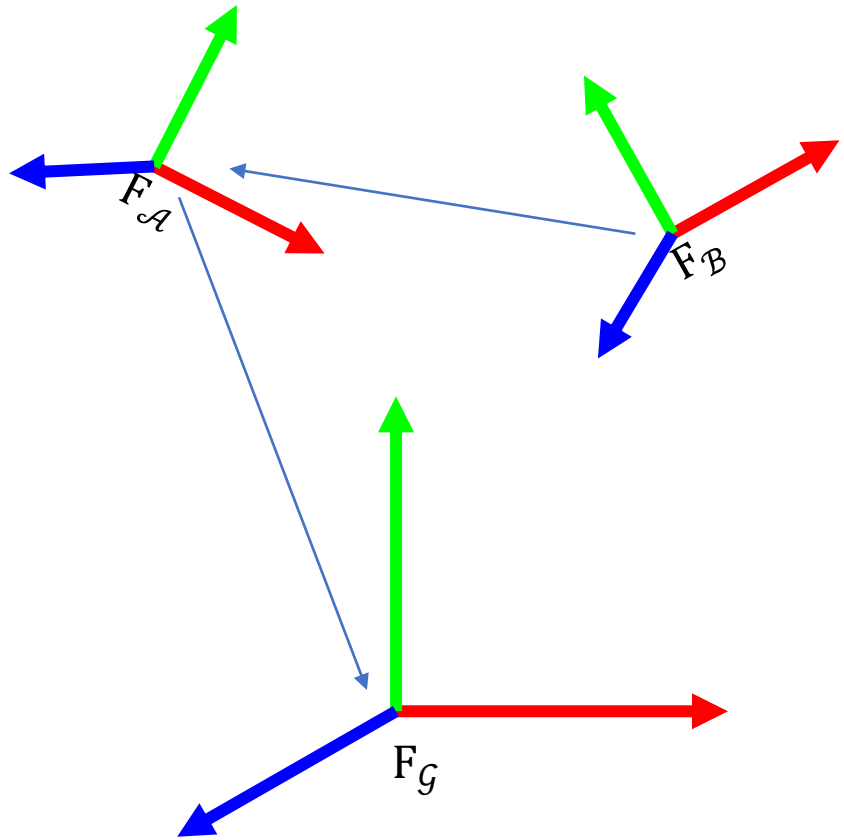
# Transform

A spatial relationship between coordinate frames can be described by a "***Transform***"

For example

The transform of $\mathrm{F}_{\mathcal{B}}$ with respect to $\mathrm{F}_{\mathcal{A}}$ describes the transformation it needs to construct $\mathrm{F}_{\mathcal{B}}$ from $\mathrm{F}_{\mathcal{A}}$

- $\mathrm{F}_{\mathcal{A}}$ is parent frame of $\mathrm{F}_{\mathcal{B}}$
- $\mathrm{F}_{\mathcal{B}}$ is child frame of $\mathrm{F}_{\mathcal{A}}$



# Components of a Transform

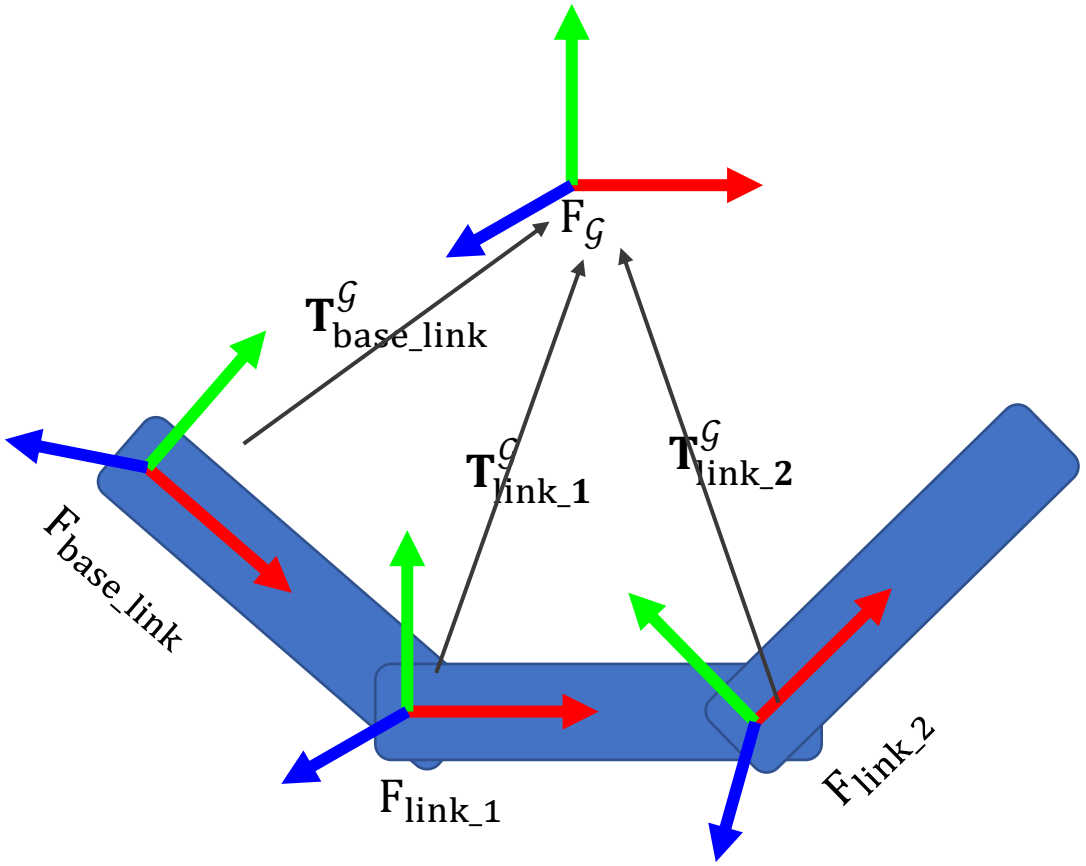
Rotation :3 DOF

- Quaternion: $\mathbf{q} = \langle q_x, q_y, q_z, q_w \rangle$

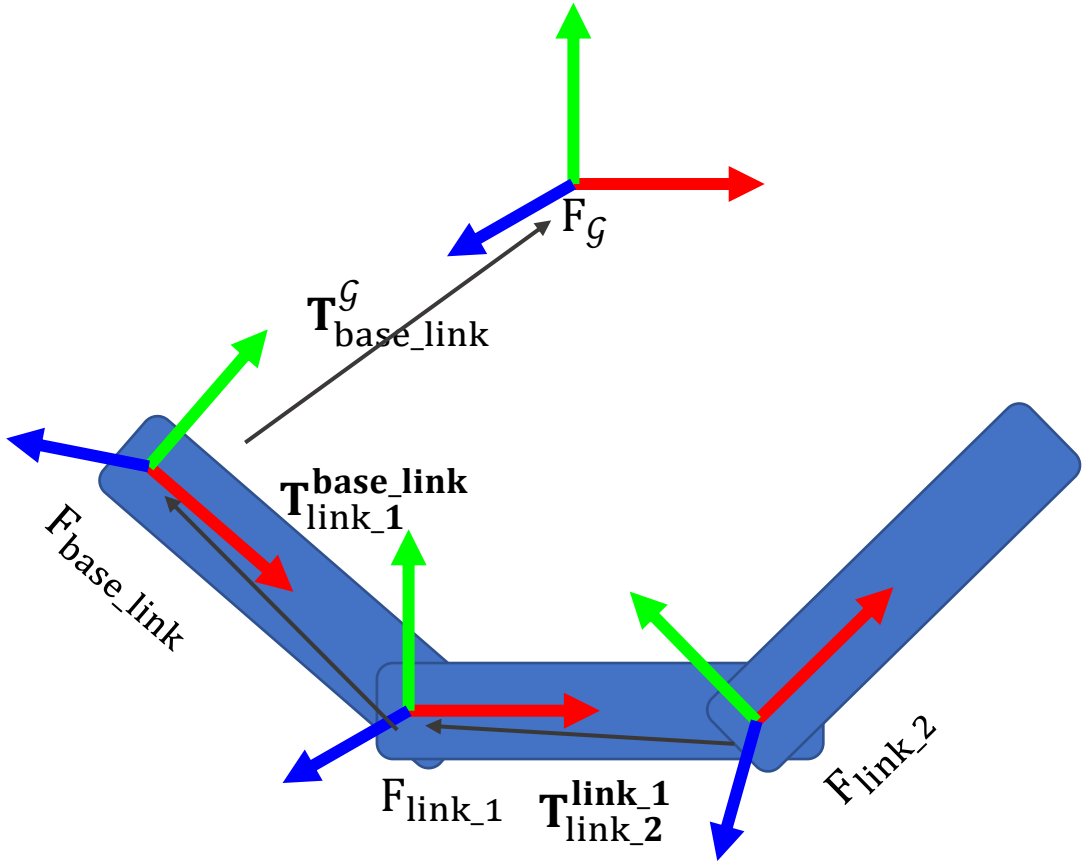
Translation : 3 DOF

- Vector3 : $\mathbf{p} = \langle p_x, p_y, p_z \rangle$

# Keeping track of Transforms




What visualizer uses to display



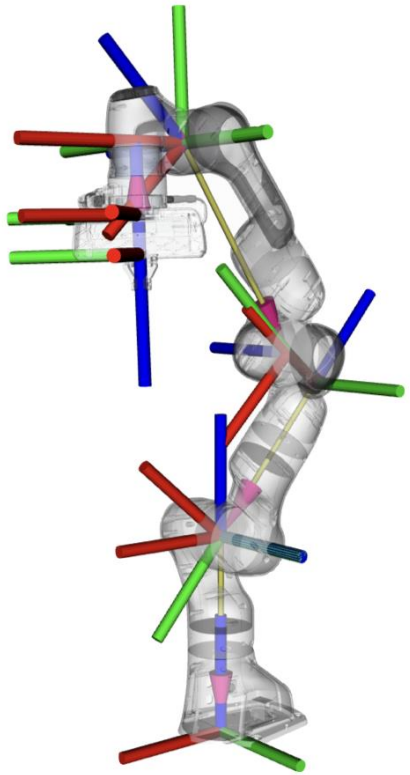

What we can use to describe

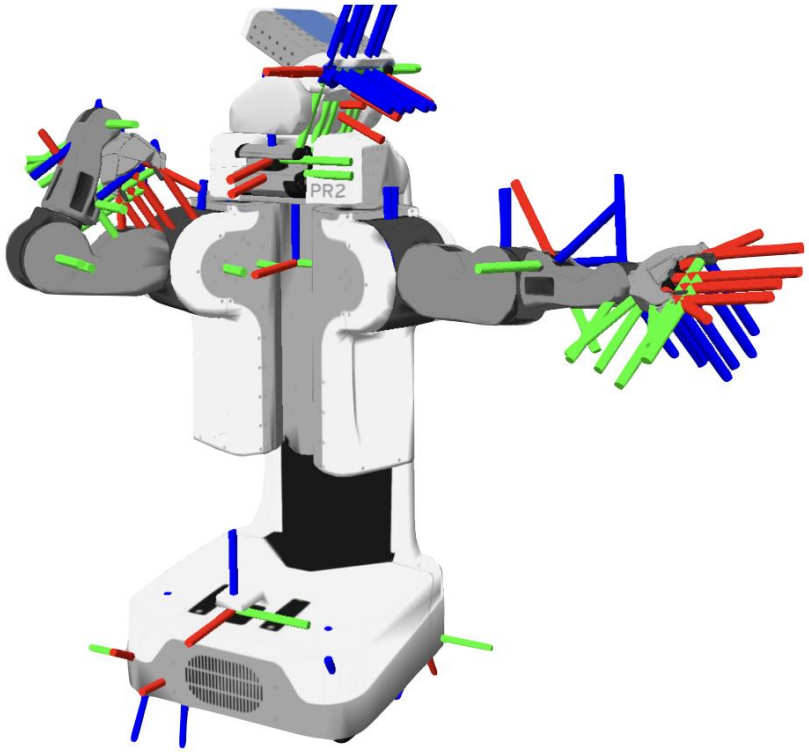
Transform

# tf2

We don’t have to create a system that track all coordinate frames. ROS2 already has a very special topic for this feature, which is known as tf2 .

ROS2 uses a topic “/tf”, which is an array of TransformStamped , to keep track of an entire tree (or multiple trees) in the system.



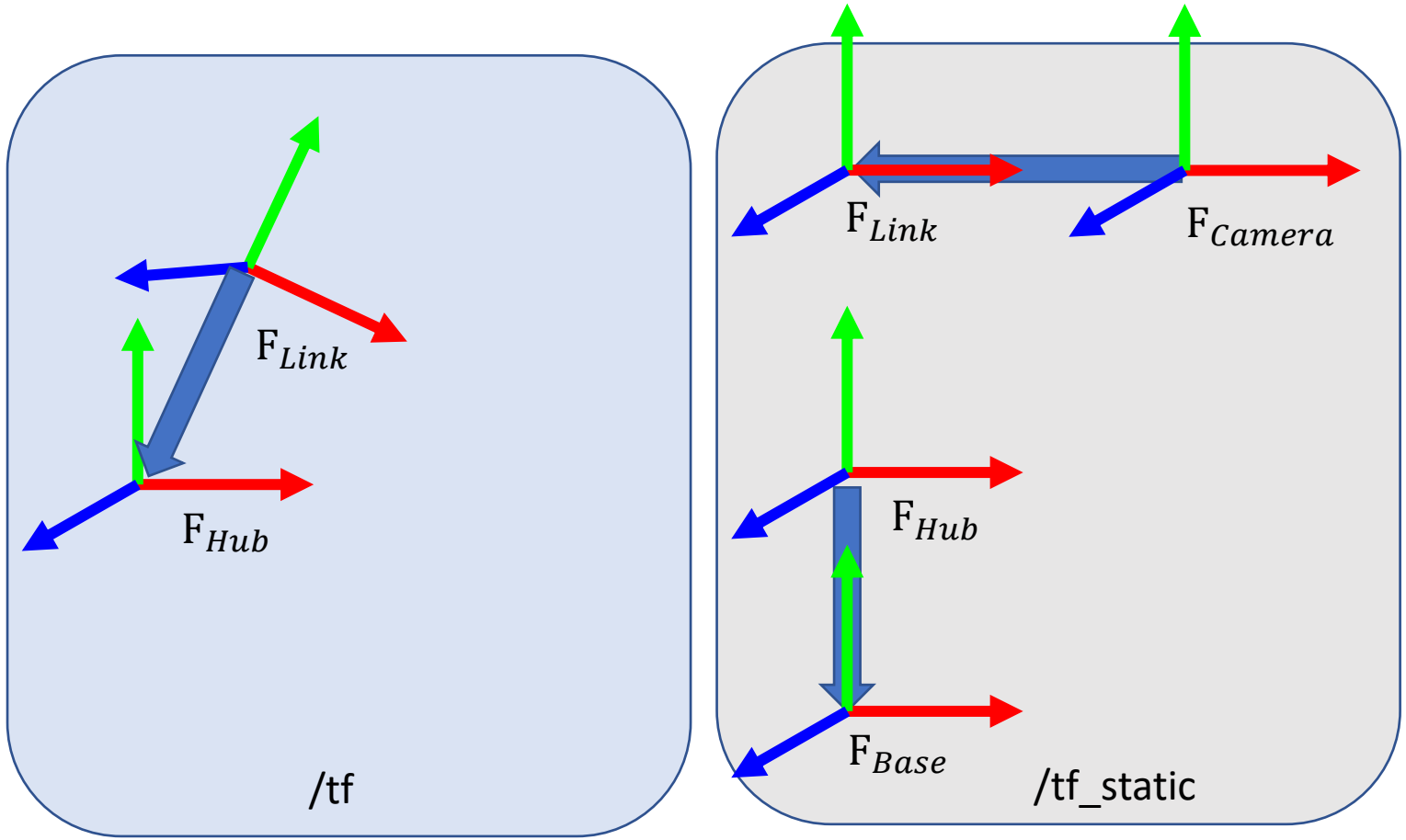



Transform

# tf2 in reality: tf & tf_static

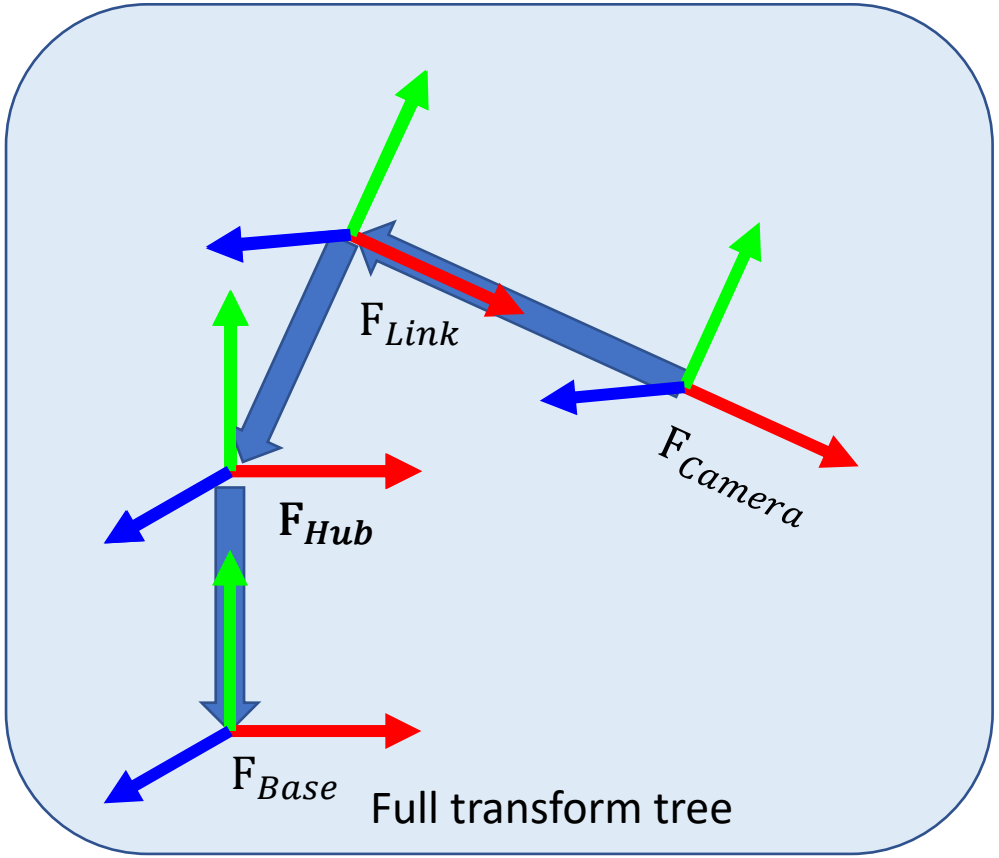
To optimize the performance, the system usually has 2 Transform trees:

- /tf : this regular tf is used to describe “varying” transforms such as a pose of a mobile robot w.r.t. a fixed landmark.
- /tf_static: this tf is used to describe any fixed transforms such as the pose between lidar link and its attached chassis.



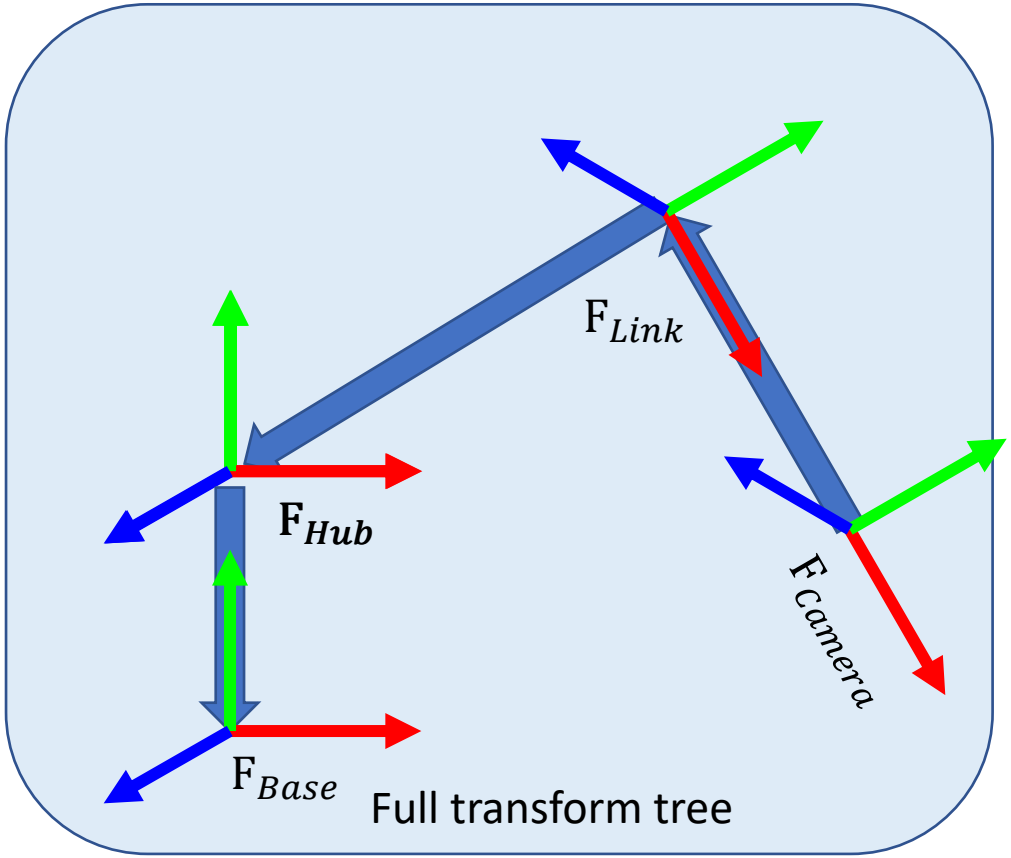
# tf2 in reality: tf & tf_static

ROS2 system combines both transform trees to compute the full tree.

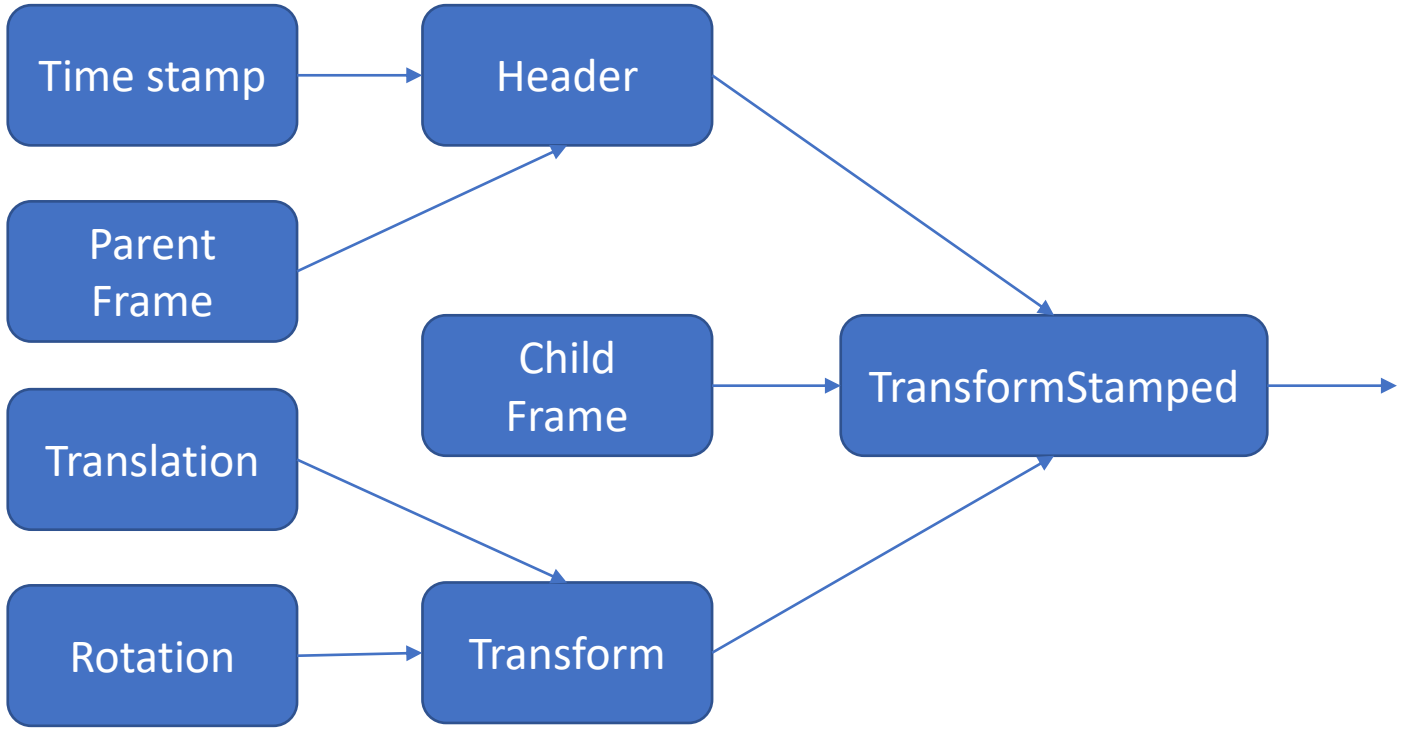


# tf2 in reality: tf & tf_static

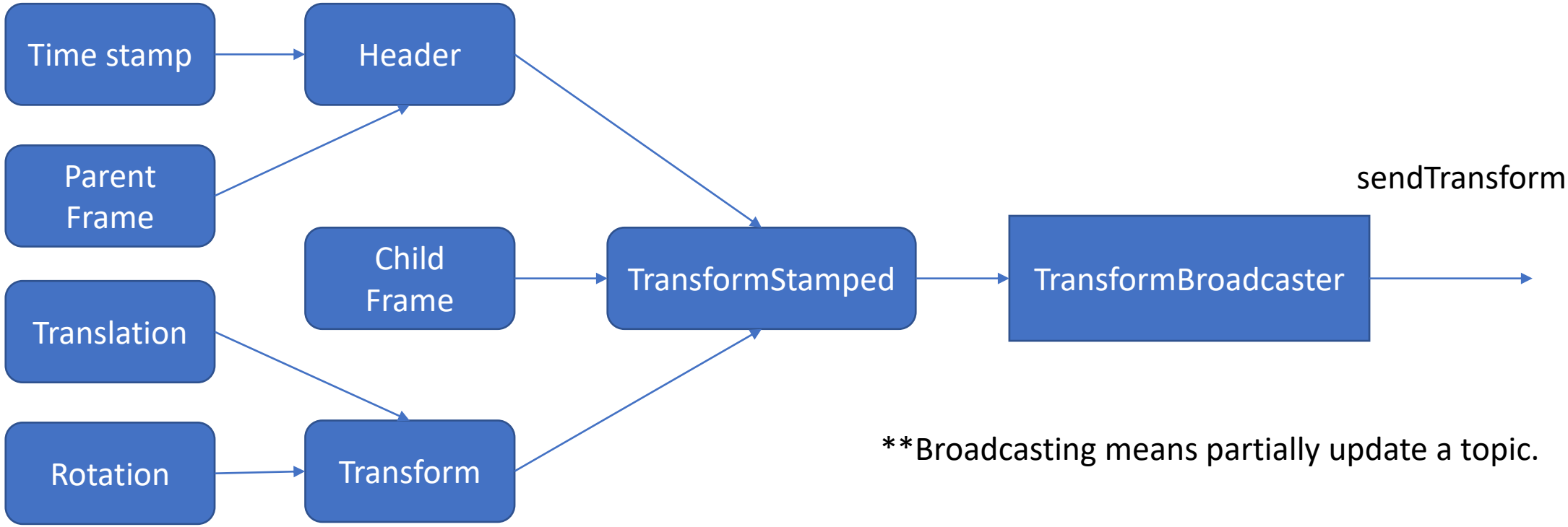
ROS2 system combines both transform trees to compute the full tree.



# Transform

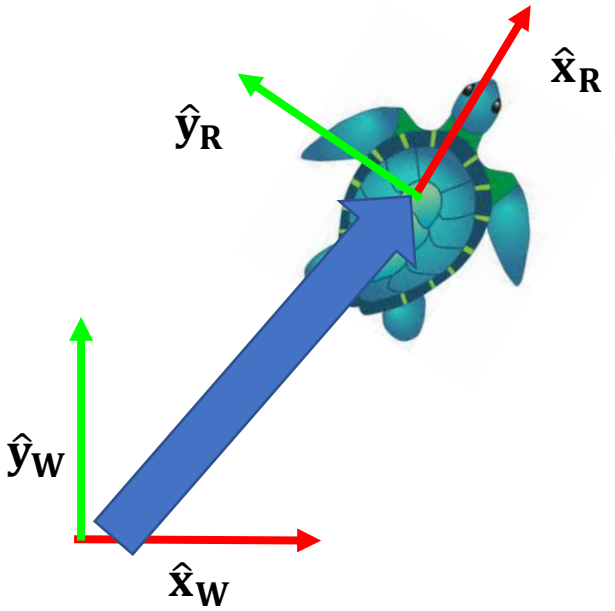

Time stamp
Header
Parent Frame
Child Frame
TransformStamped
Translation
Rotation
Transform

# TransformBroadcaster




Transform

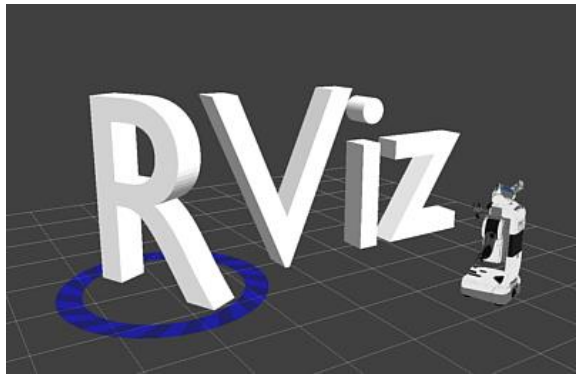
# Example: TF2 from Turtlesim




World Frame ($W$) : /world
Robot Frame ($R$) : /{turtlename}

# Visualization

# Visualizer vs (Physics) Simulator



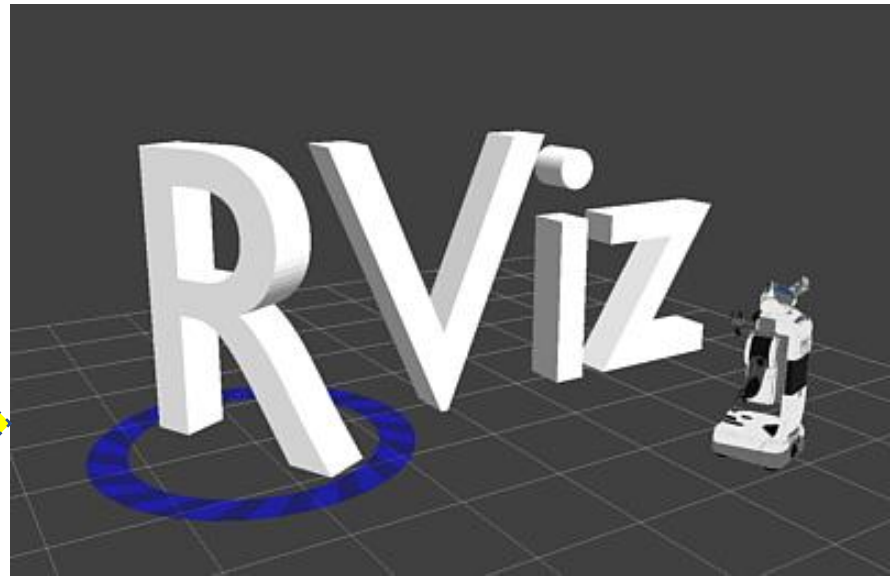

- For visualizing only
- Display whatever we tell them to

GAZEBO

- Has physics engine
- Calculate motion and dynamics of the entire system, then display them

# rviz2

/tf

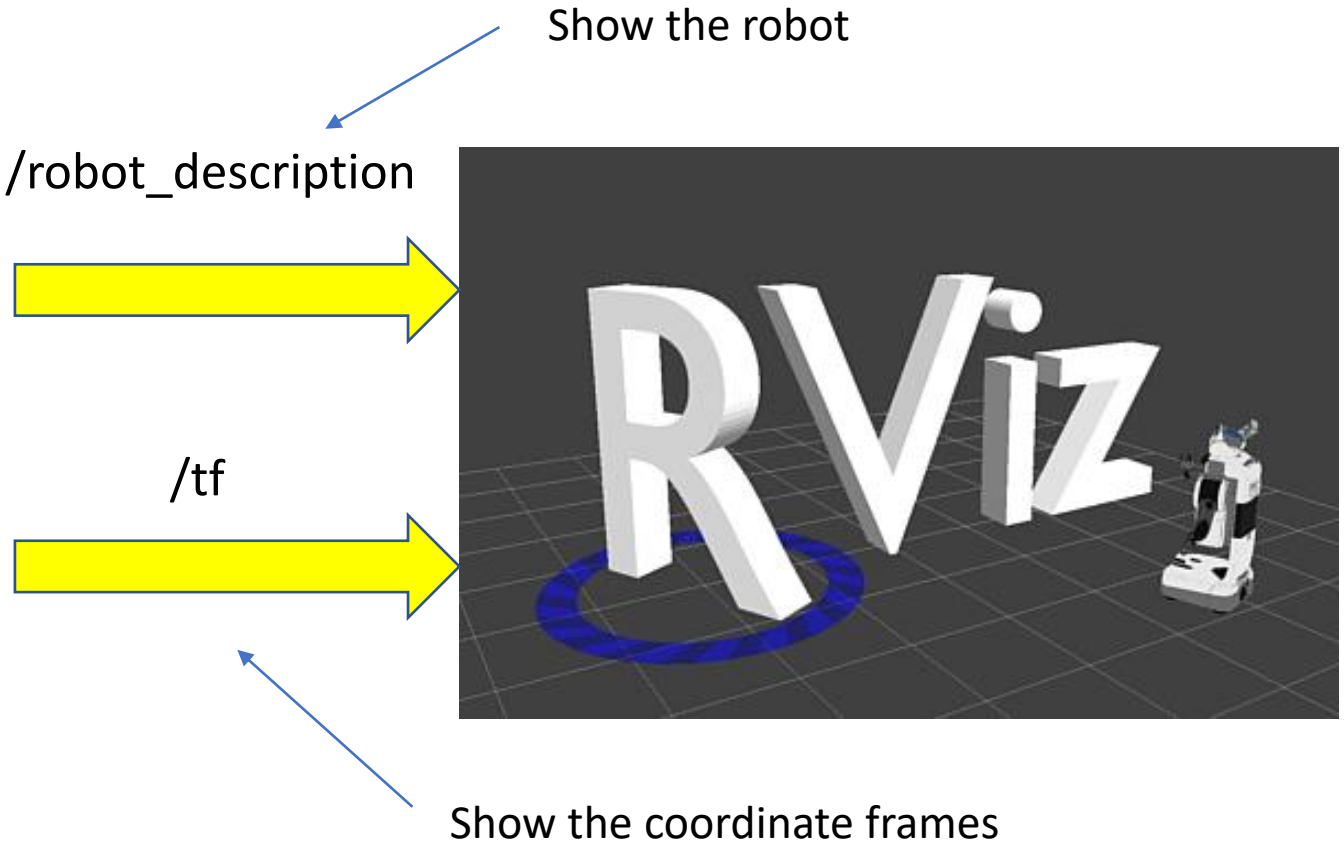



We can use rviz to visualize frames from /tf, but we need more than that to display an entire robot.

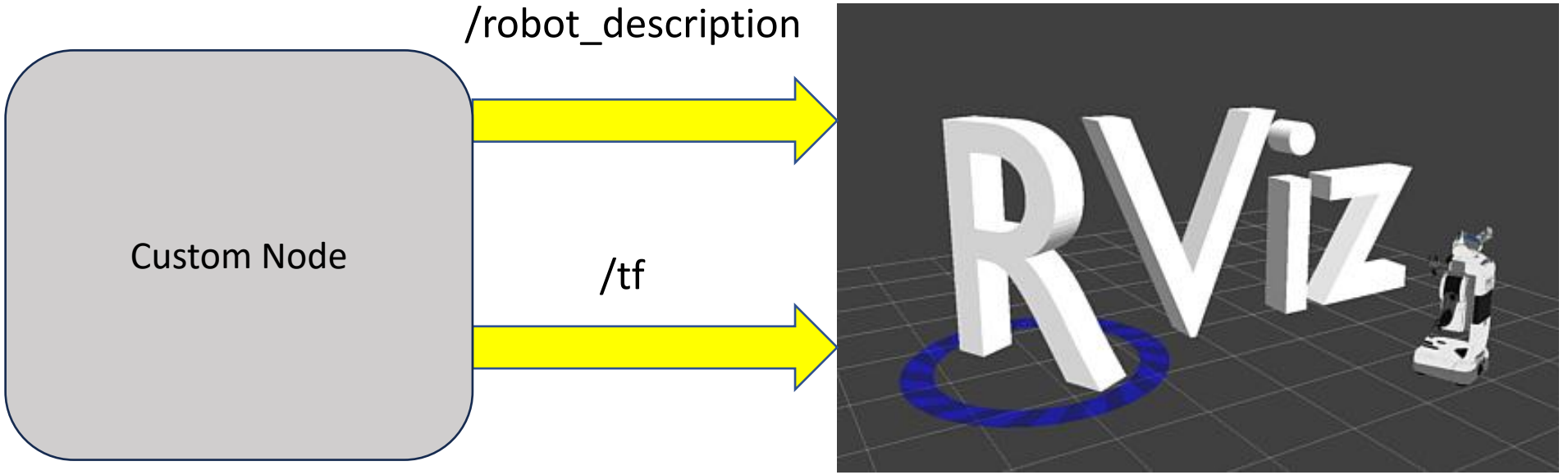
# robot_description

"robot_description" is a default topic that describes an appearance of a robot.

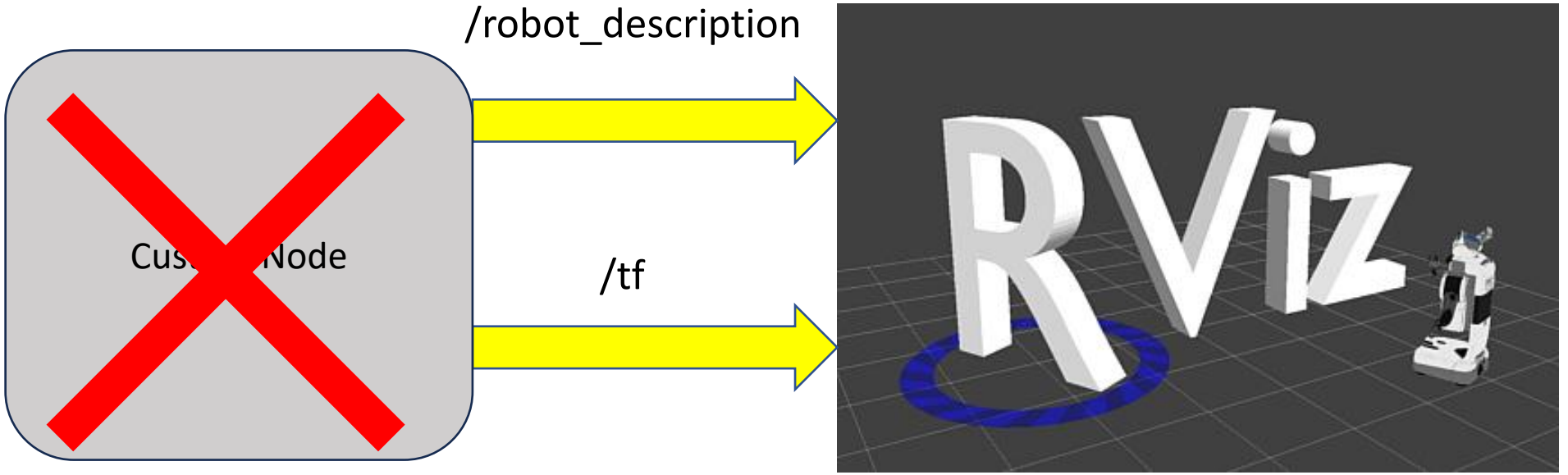
- "robot_description" is "std_msgs/String"
- The string must be written in a "specific" xml format .
- We can write a node to create the string, and publish to rviz, but someone already did that.



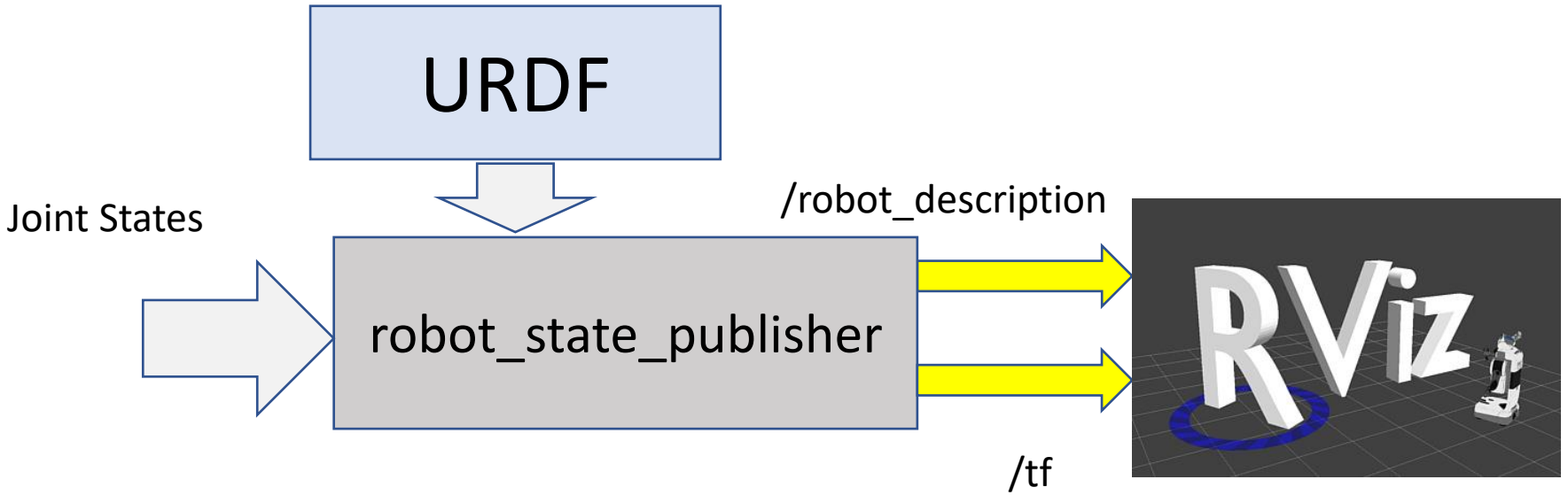
# robot_description



# robot_description

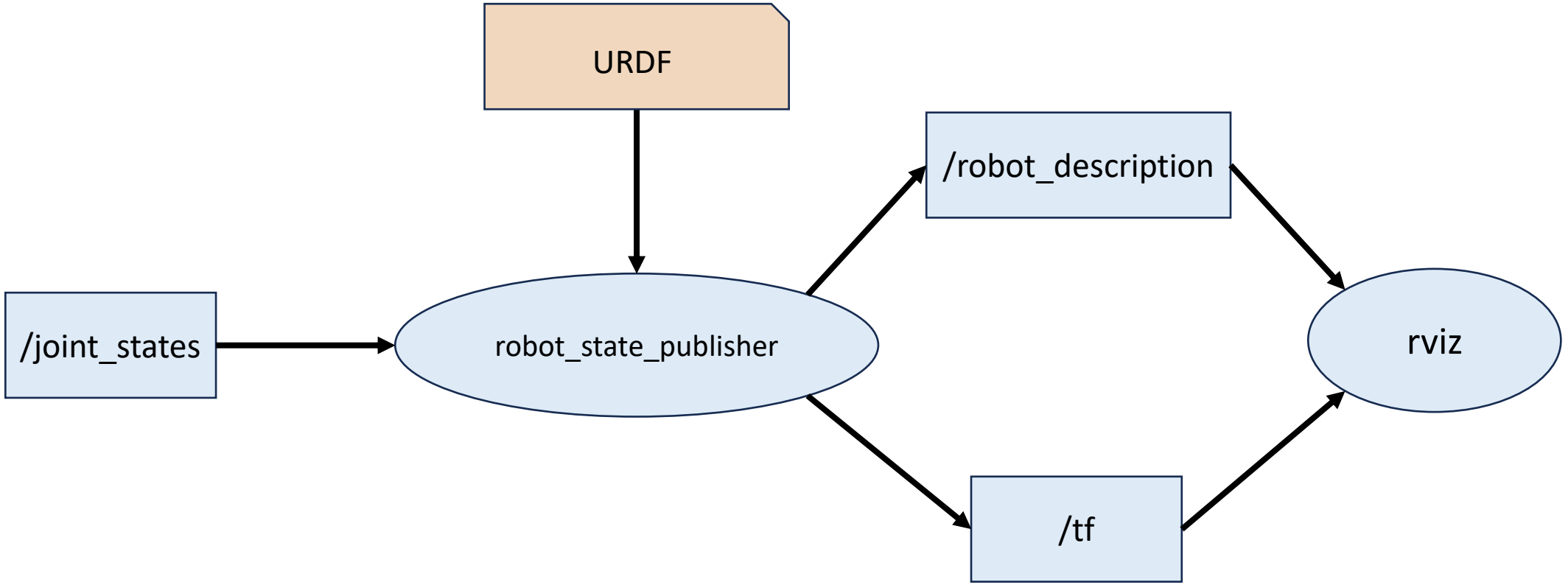


# robot_state_publisher

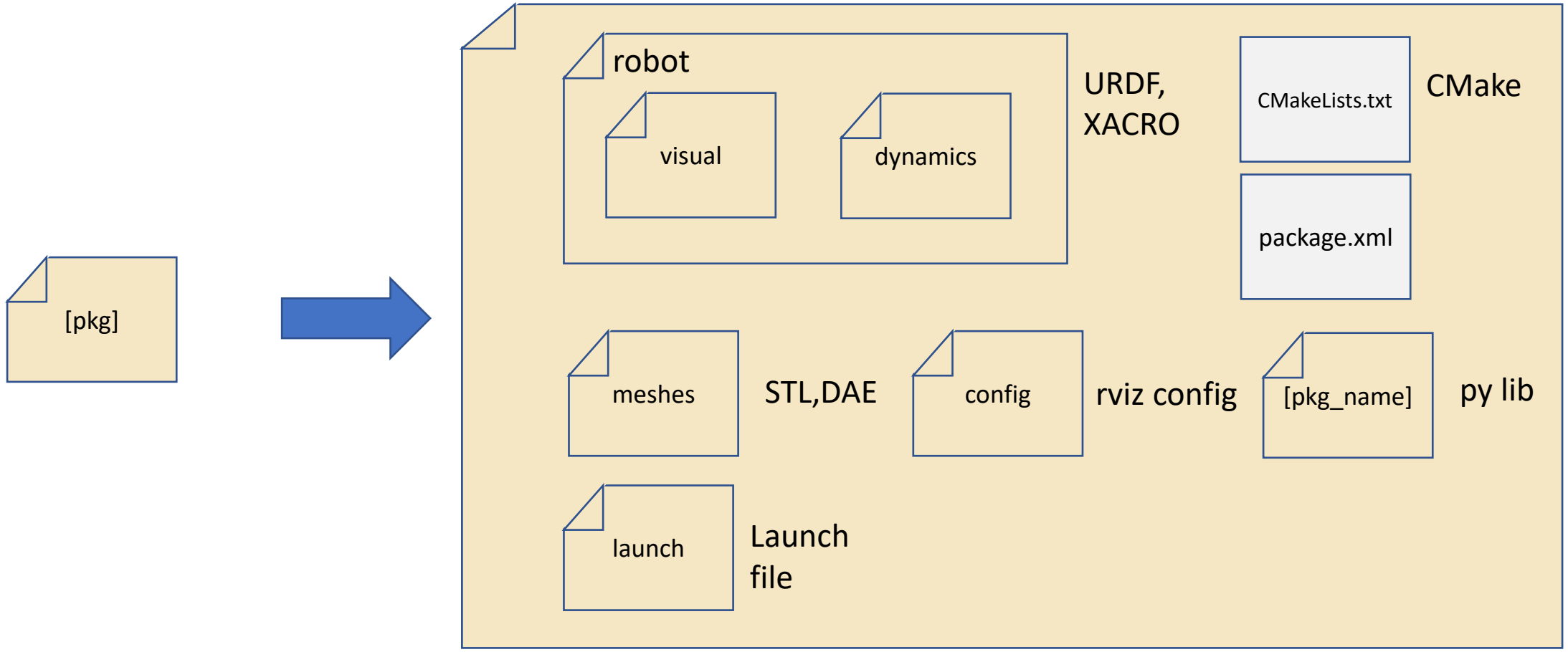



A node called “robot_state_publisher” will take a URDF file and joint states to compute forward kinematics and generate messages to both “robot_description” and “tf” automatically. !!

# What we must do for displaying a robot in rviz2



# The structure of {robot's name}_description



# URDF: Unified Robotics Description Format

# URDF

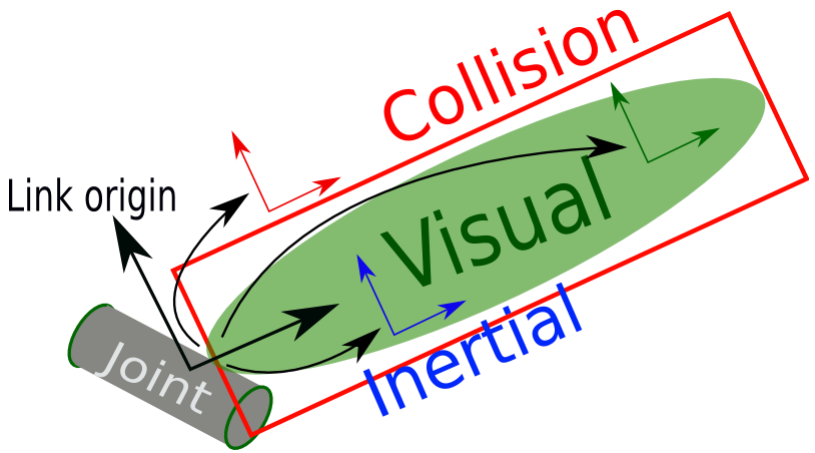
A format for describing the appearance and physical parameters of any open-kinematics chain.

Links

Joints

URDF: Unified Robotics Description Format

# Link




visual

inertial

collision

link's appearance: predefined shape, or mesh file

Inertial parameters: for physics engine to compute its dynamics

Collision Profile: for physics engine to compute collision dynamics

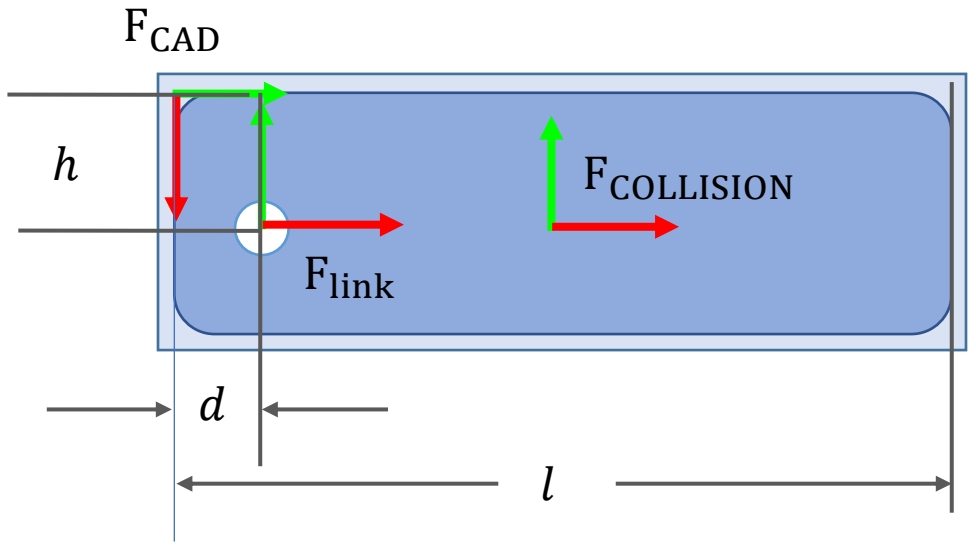
URDF: Unified Robotics Description Format

# Link

```
 1  <link name="my_link">
 2    <inertial>
 3      <origin xyz="0 0 0.5" rpy="0 0 0"/>
 4      <mass value="1"/>
 5      <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
 6    </inertial>
 7
 8    <visual>
 9      <origin xyz="0 0 0" rpy="0 0 0" />
10      <geometry>
11        <mesh file=file://$(find description_pkg)/meshes/>
12      </geometry>
13      <material name="Cyan">
14        <color rgba="0 1.0 1.0 1.0"/>
15      </material>
16    </visual>
17
18    <collision>
19      <origin xyz="0 0 0" rpy="0 0 0"/>
20      <geometry>
21        <cylinder radius="1" length="0.5"/>
22      </geometry>
23    </collision
```

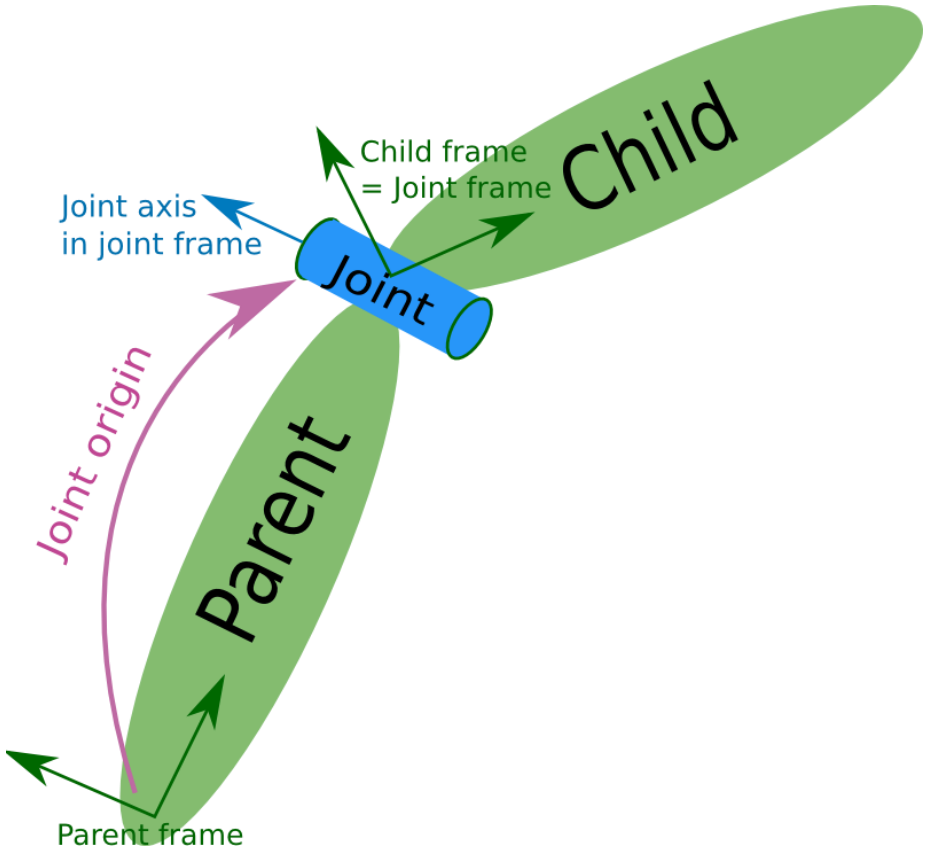
# Link (example)

```
 1  <link name="my_link">
 2    <inertial>
 3      <origin xyz="0 0 0.5" rpy="0 0 0"/>
 4      <mass value="1"/>
 5      <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
 6    </inertial>
 7
 8    <visual>
 9      <origin xyz="0 0 0" rpy="0 0 0" />
10      <geometry>
11          <mesh file=file://$(find description_pkg)/meshes/>
12      </geometry>
13      <material name="Cyan">
14        <color rgba="0 1.0 1.0 1.0"/>
15      </material>
16    </visual>
17
18    <collision>
19      <origin xyz="0 0 0" rpy="0 0 0"/>
20      <geometry>
21        <cylinder radius="1" length="0.5"/>
22      </geometry>
23    </collision
```




URDF: Unified Robotics Description Format

# Joint



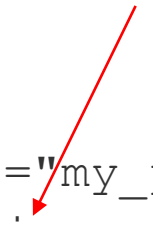

Types:

- Revolute
- Continuous
- Prismatic
- Fixed
- Floating
- planar

URDF: Unified Robotics Description Format

# Joint

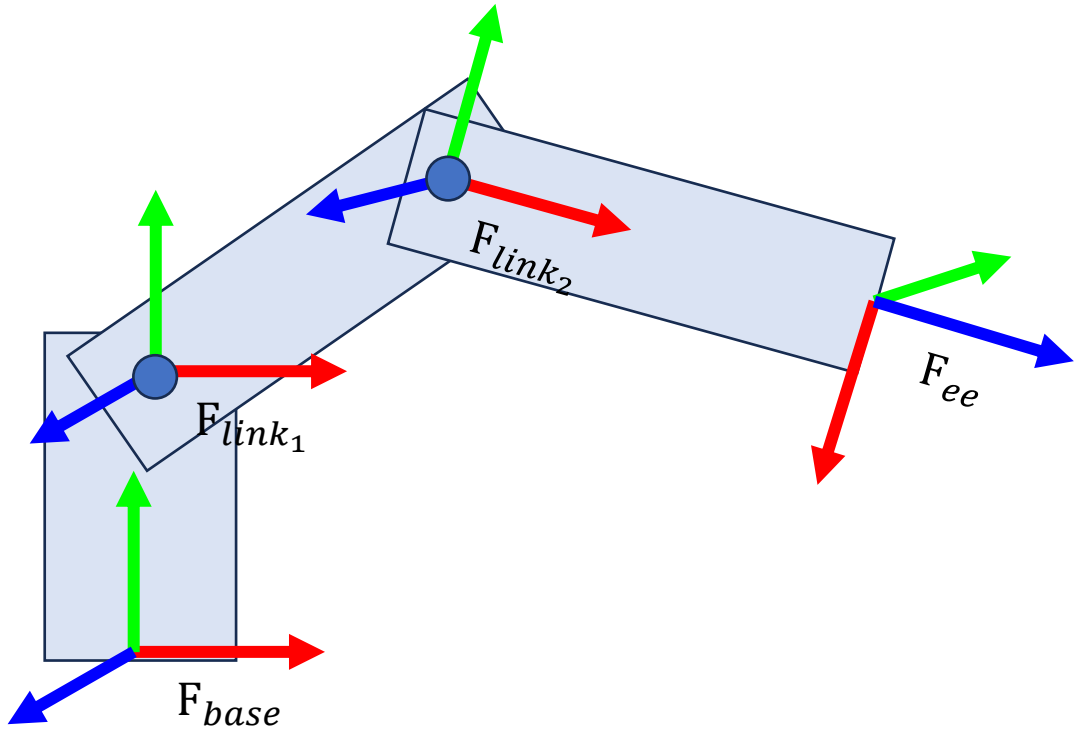
Fixed transformation before the actuation



```
1  <joint name="my_joint" type="floating">
2      <origin xyz="0 0 1" rpy="0 0 3.1416"/>
3      <parent link="link1"/>
4      <child link="link2"/>
5
6      <dynamics damping="0.0" friction="0.0"/>
7      <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
8
9  </joint>
```

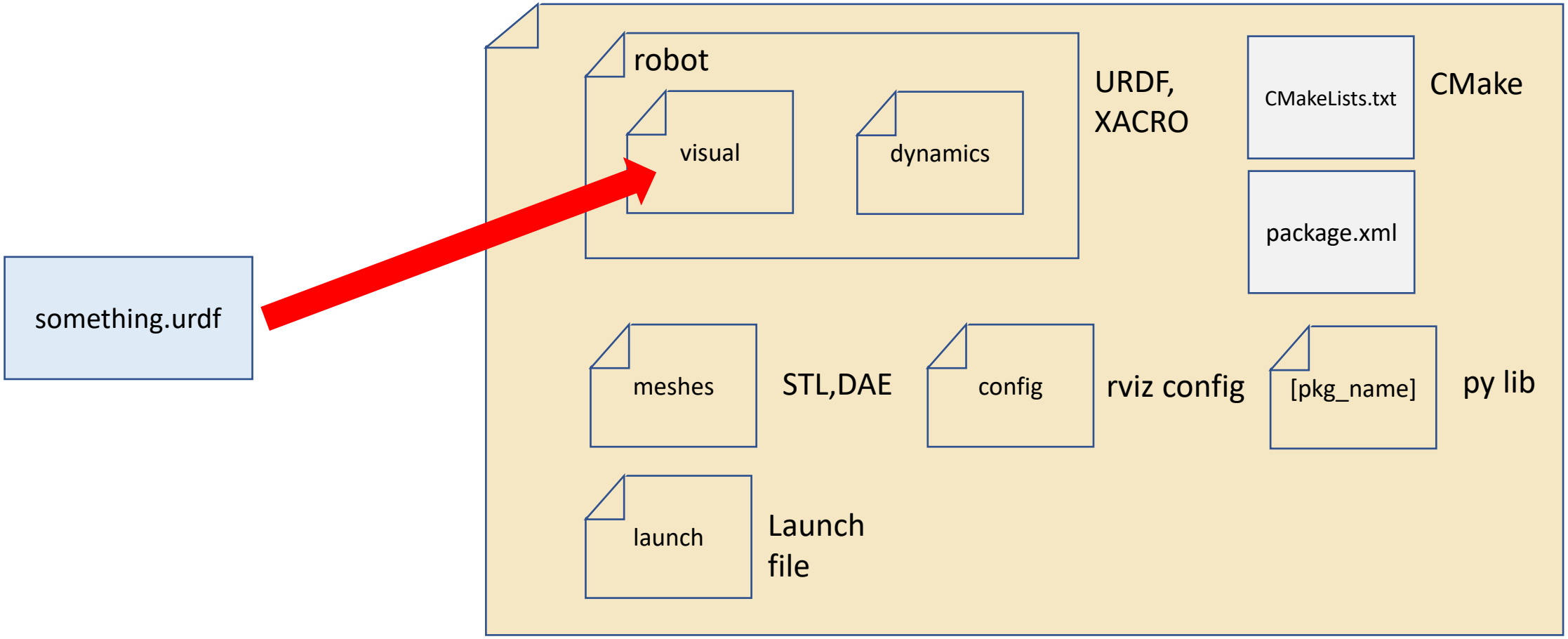
# URDF File's content

```
<?xml version="1.0"?>
<robot name="name">
    <link name="base">
            ...
    </link>
    <link name="link_1">
            ...
    </link>
    <link name="link_2">
            ...
    </link>
    <link name="end_effector">
            ...
    </link>
    <joint name="joint_1" type="revolute">
            ...
    </joint>
    <joint name="joint_2" type="revolute">
            ...
    </joint>
    <joint name= "end_effector_offset" type="fixed">
            ...
    </joint>
</robot>
```




URDF: Unified Robotics Description Format

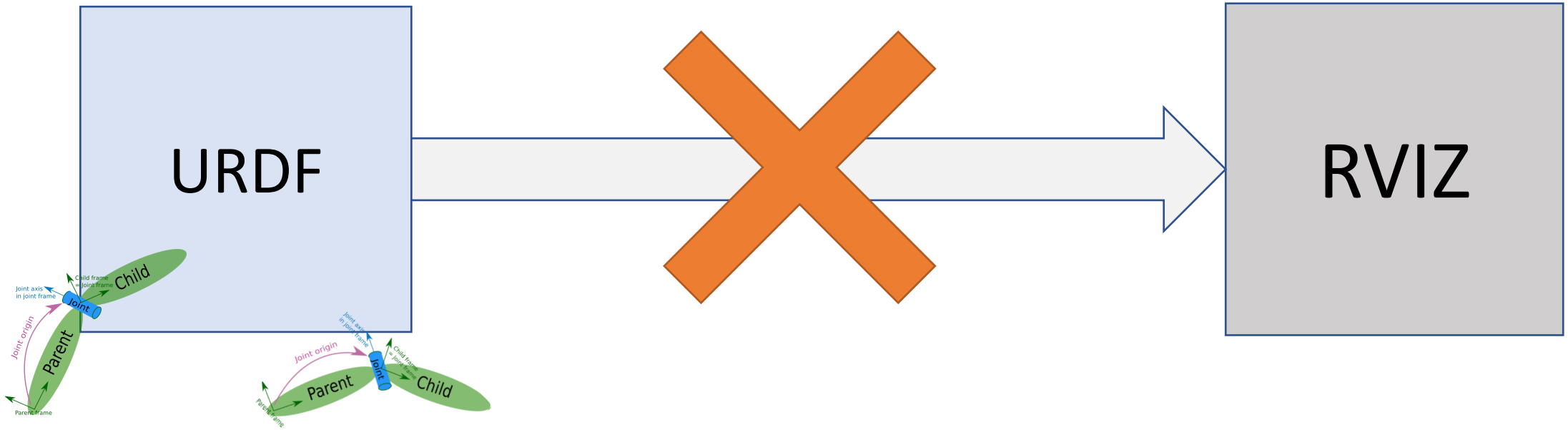
# Where do we put our URDF



# From URDF to RVIZ


URDF: Unified Robotics Description Format

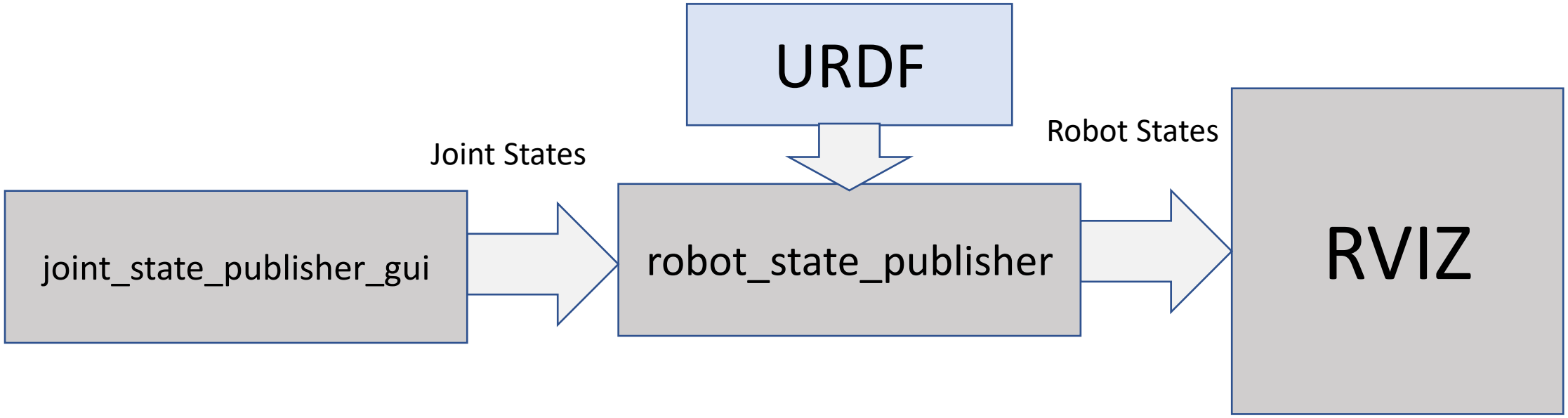
# From URDF to RVIZ




URDF does not describe what the robot is currently doing, but RVIZ requires its current state.

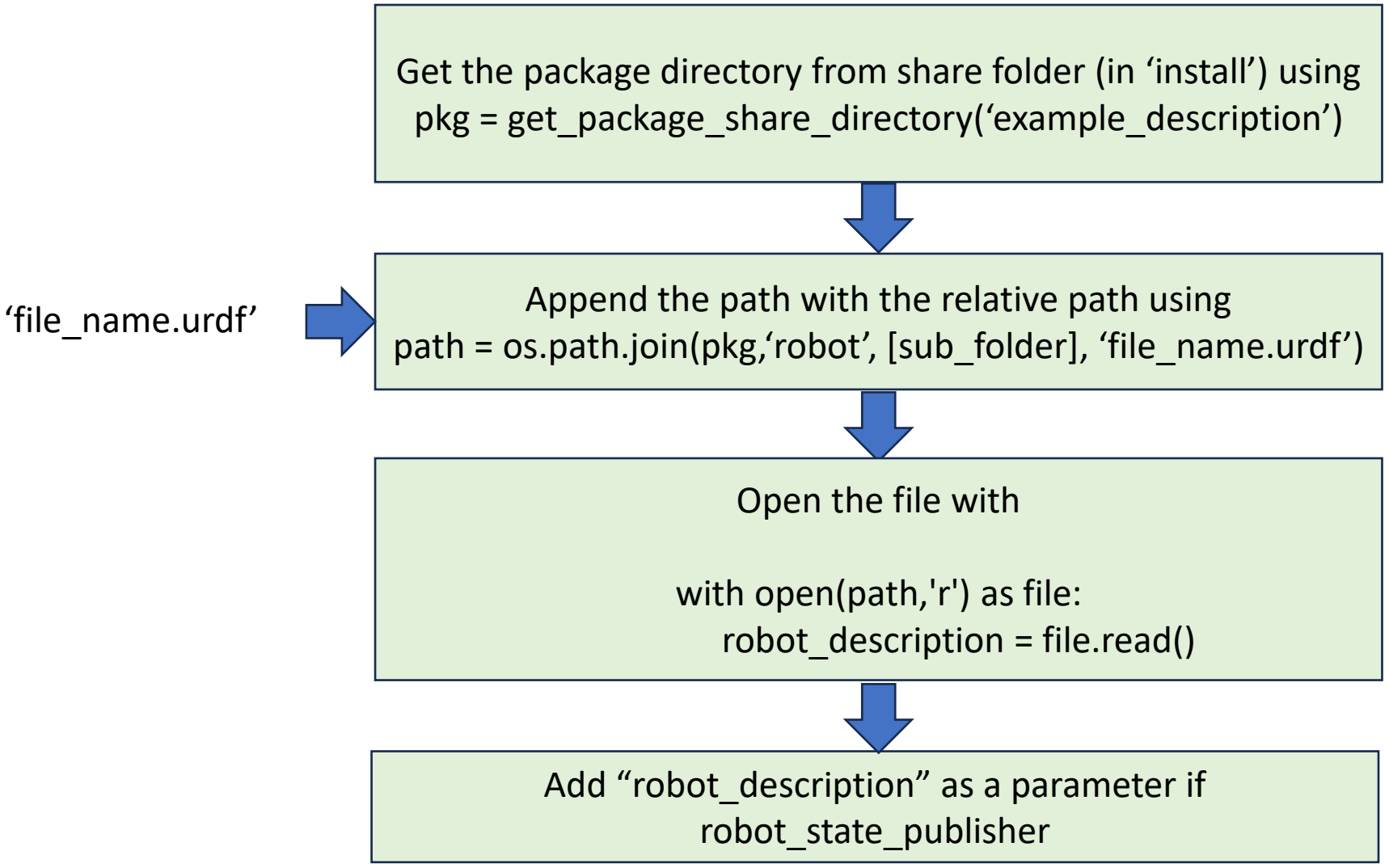
URDF: Unified Robotics Description Format

# joint_state_publisher

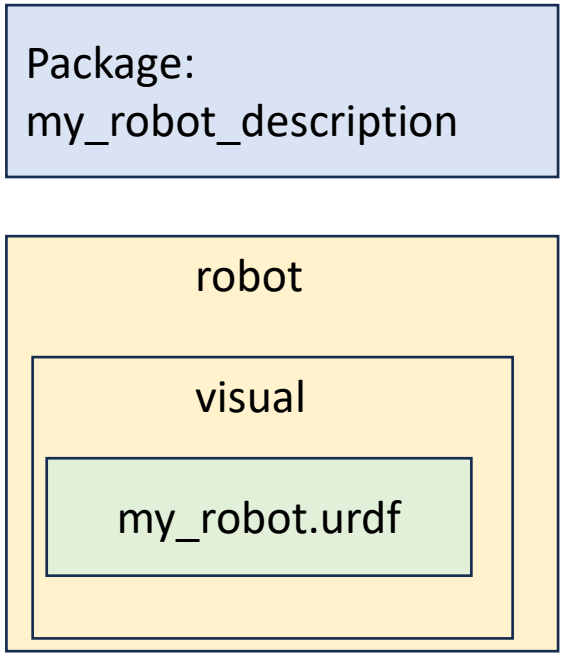



URDF: Unified Robotics Description Format

# General steps for displaying a robot URDF in RVIZ

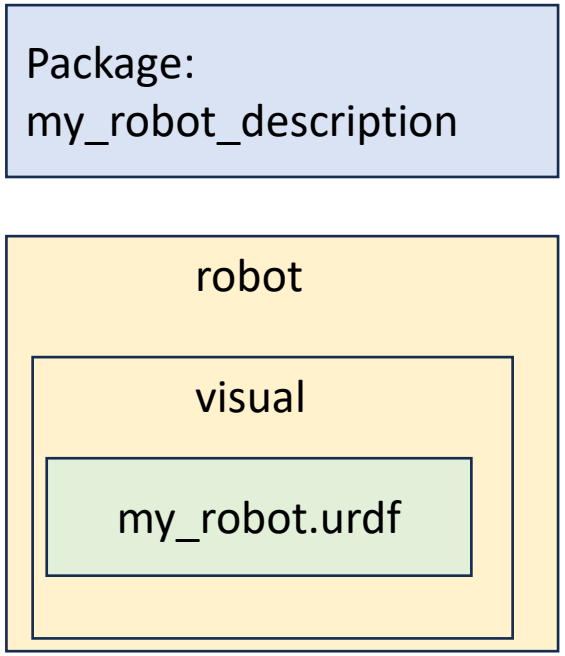


# Example




URDF: Unified Robotics Description Format

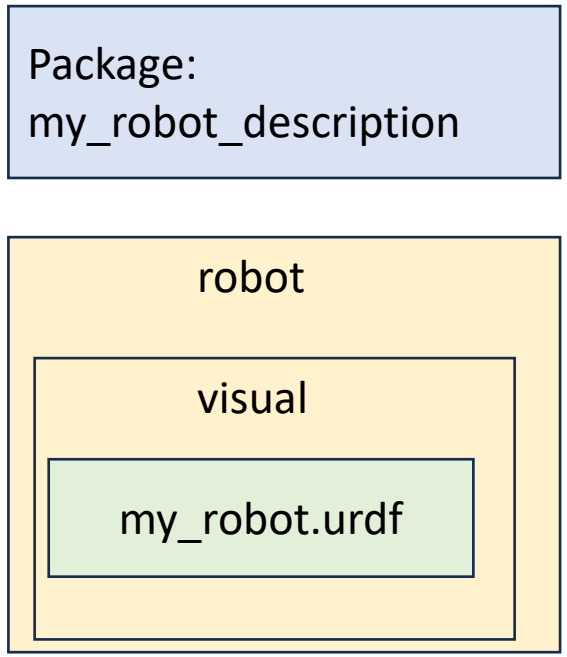
# Example




```
from launch import LaunchDescription


def generate_launch_description()









    launch_description = LaunchDescription()

    return launch_description
```

URDF: Unified Robotics Description Format

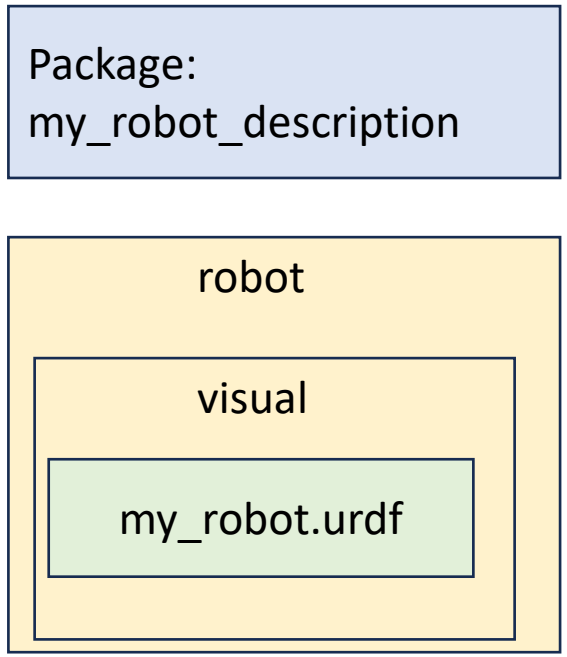
# Example




```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description()


    path = ...




    robot_state_publisher = Node(
        package = 'robot_state_publisher',
        executable = 'robot_state_publisher',
        parameters = [{'robot_description':robot_desc}]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(robot_state_publisher)

    return launch_description
```

URDF: Unified Robotics Description Format

# Example




```
from launch import LaunchDescription
from launch_ros.actions import Node



def generate_launch_description()

    pkg = get_package_share_directory('my_robot_description')

    path = os.path.join(pkg,'robot','visual','my_robot.urdf')

    with open(path,'r') as file:
        robot_desc = file.read()


    robot_state_publisher = Node(
        package = 'robot_state_publisher',
        executable = 'robot_state_publisher',
        parameters = [{'robot_description':robot_desc}]
    )

    launch_description = LaunchDescription()

    launch_description.add_action(robot_state_publisher)

    return launch_description
```

URDF: Unified Robotics Description Format

# Example

Package:
my_robot_description

robot

visual

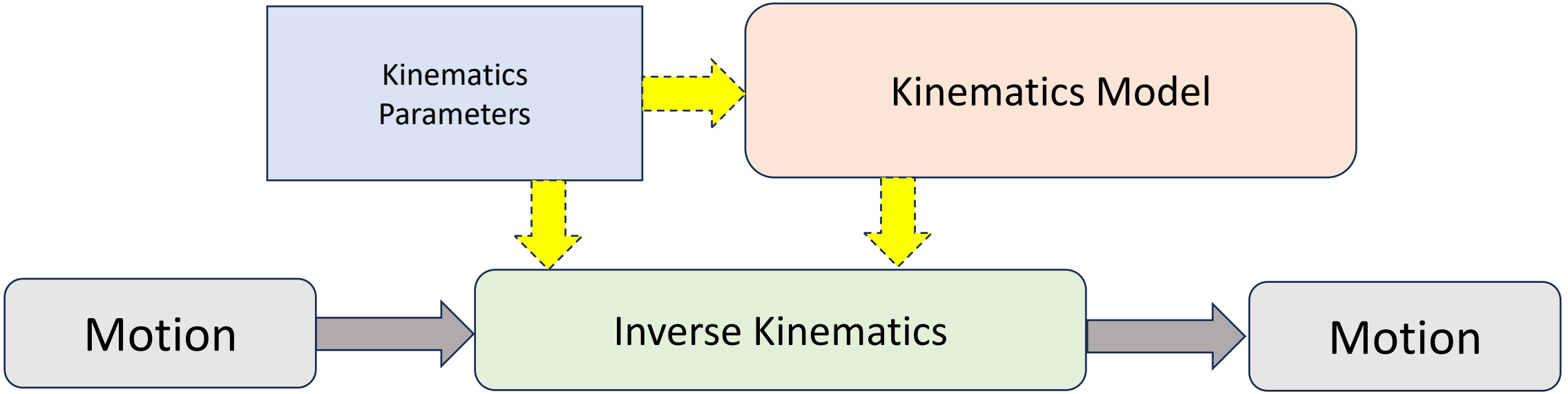
my_robot.urdf

```
from launch import LaunchDescription
from launch_ros.actions import Node



def generate_launch_description()

    ...

    robot_state_publisher = ...

    rviz = ...

    joint_state_publisher_gui = ...

    launch_description = LaunchDescription()

    launch_description.add_action(robot_state_publisher)

    launch_description.add_action(rviz)

    launch_description.add_action(joint_state_publisher_gui)

    return launch_description
```

URDF: Unified Robotics Description Format

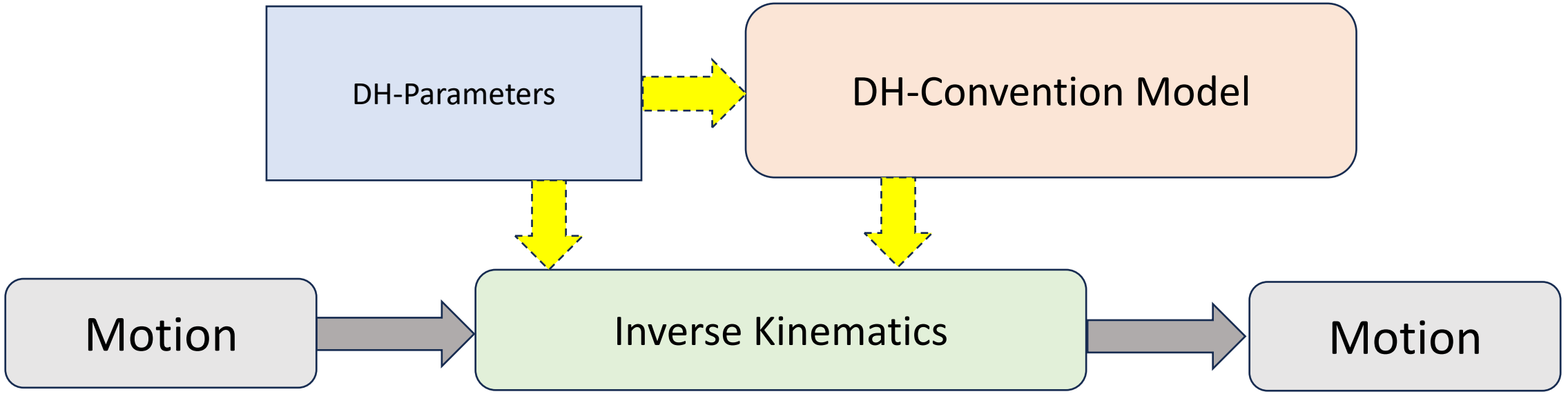
# Robot Parameters

# Parameters vs. Models

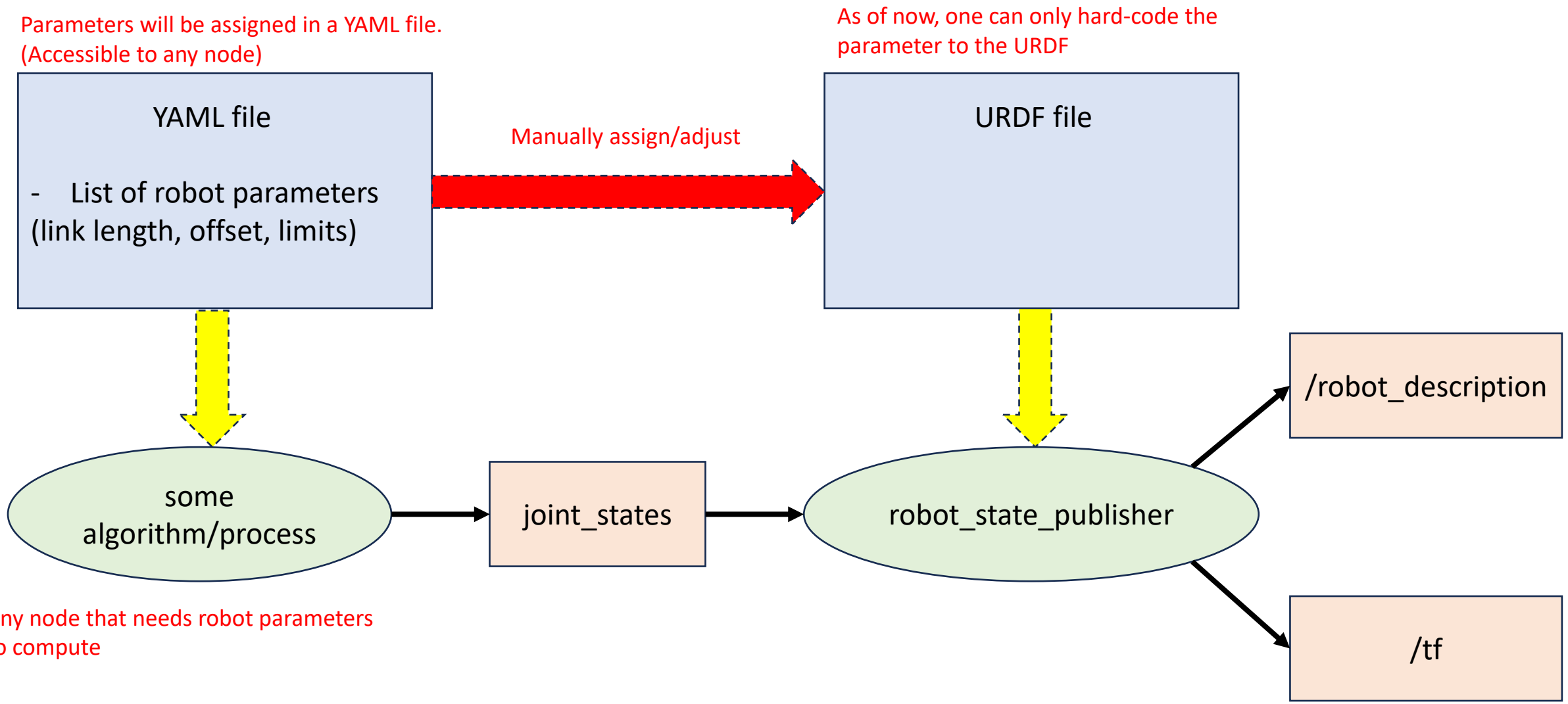



- A robot model is a mathematical representation of a robot.
- A robot parameter is a numerical value that is used to describe robot's property.

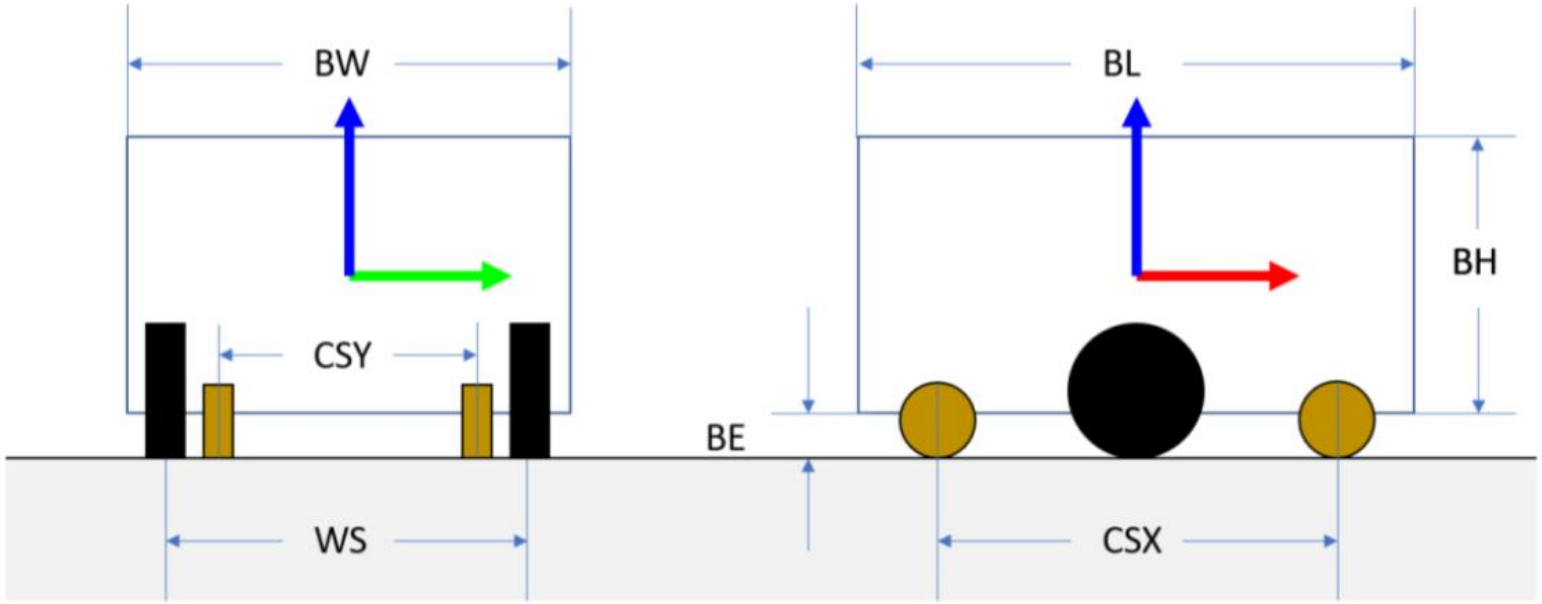
# Example: Serial Manipulator



# Possible Software Implementation



# Example: Differential Drive Robot

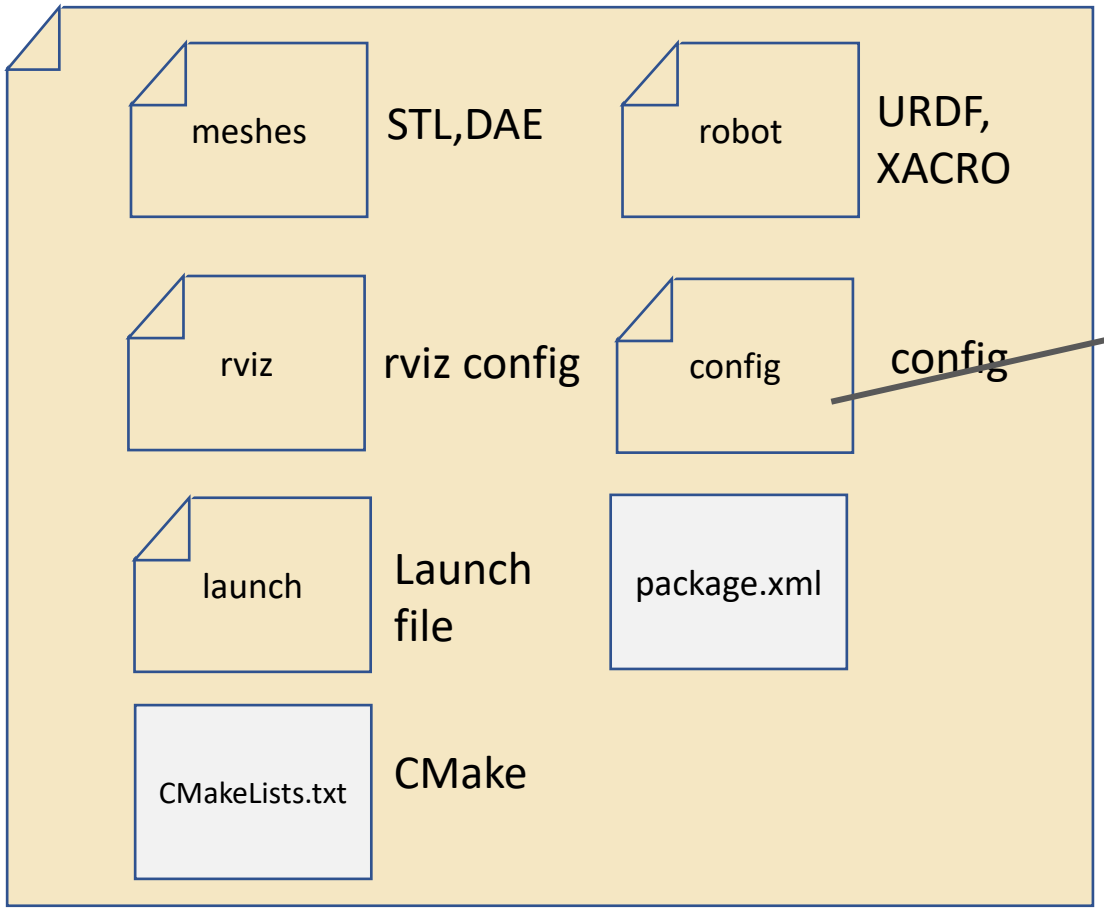



BASE_LENGTH
BASE_WIDTH
BASE_HEIGHT
BASE_ELEVATION

WHEEL_RADIUS
WHEEL_SEPARATION

CASTOR_RADIUS
CASTOR_SEPARATION_X
CASTOR_SEPARATION_Y

# [name]_description

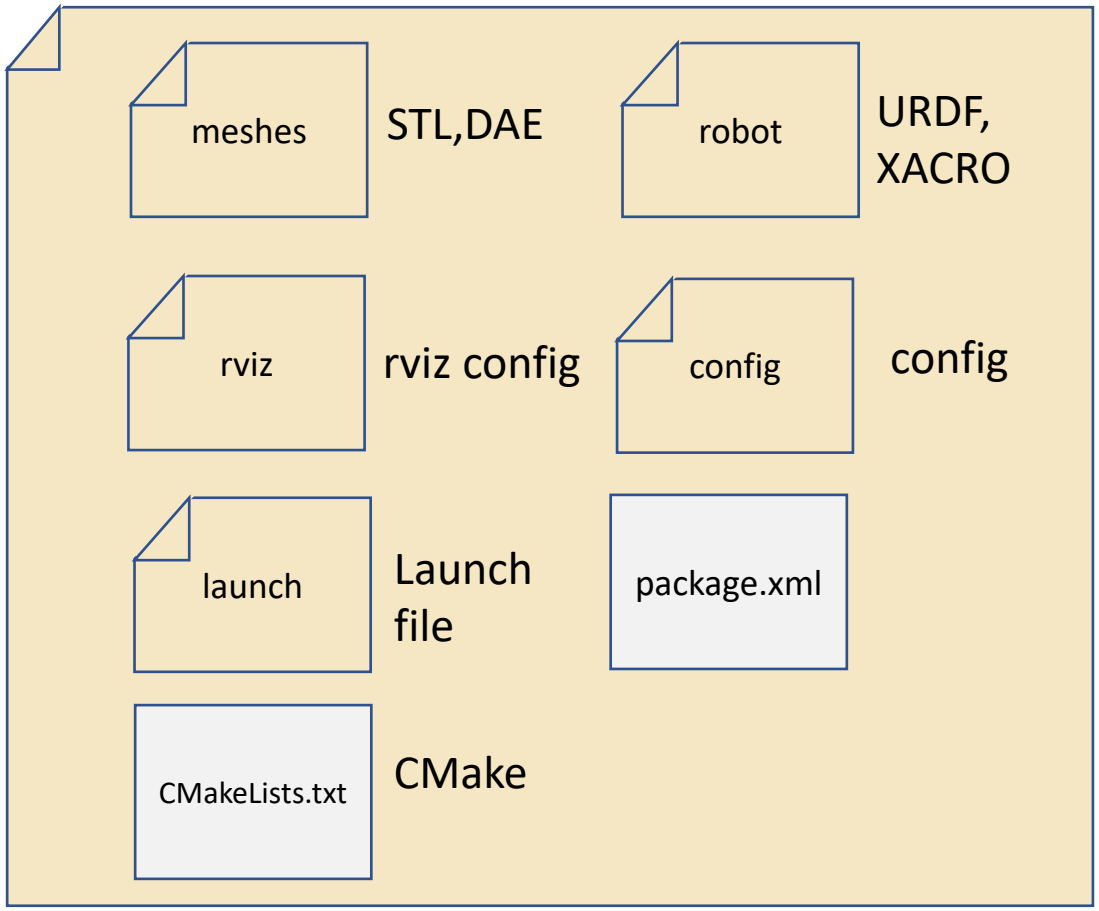



parameters.yaml

```yaml
# wheel visual properties
wheel_radius: "0.08"
wheel_thickness: "0.04"
wheel_separation: "0.45"
wheel_color: "0 0 0 1"
# base visual properties
base_length: "0.65"
base_width: "0.55"
base_height: "0.25"
base_elevation: "0.04"
package_name: "xxx_description"
base_mesh_path: "meshes/coconut.stl"
base_mesh_z_offset: "0.018"
base_color: "1 1 1 1"
# castor visual properties
castor_radius: "0.035"
castor_thickness: "0.02"
castor_separation_x: "0.46"
castor_separation_y: "0.32"
castor_color: "0.5 0.5 0 1"
# pole contact properties
pole_radius: "0.015"
pole_height: "0.05"
pole_separation_x: "0.47"
pole_separation_y: "0.4"
 # lidar visual properties
lidar_radius: "0.035"
lidar_thickness: "0.04"
lidar_offset: "0.22"

# camera visual properties
camera_offset: "0.317"
camera_height: "0.1"
```

Robot Parameters

# example_description




kinematics_parameter.yaml

```
base:
  orientation: 0.0 0.0 0.0
  position: 0.0 0.0 0.0
types:
- revolute
- revolute
- revolute
links:
- orientation: 0.0 0.0 0.0
  position: 0.0 0.0 0.2
- orientation: 0.0 -1.5707963267948966 -1.5707963267948966
  position: 0.0 -0.12 0.0
- orientation: 0.0 0.0 0.0
  position: 0.25 0.0 0.1
joints:
- lower: -3.141592653589793
  upper: 3.141592653589793
  velocity: 1000.0
  effort: 1000.0
- lower: -3.141592653589793
  upper: 3.141592653589793
  velocity: 1000.0
  effort: 1000.0
- lower: -3.141592653589793
  upper: 3.141592653589793
  velocity: 1000.0
  effort: 1000.0
end_effector:
  orientation: 1.5707963267948966 0.0 1.5707963267948966
  position: 0.28 0.0 0.0
```

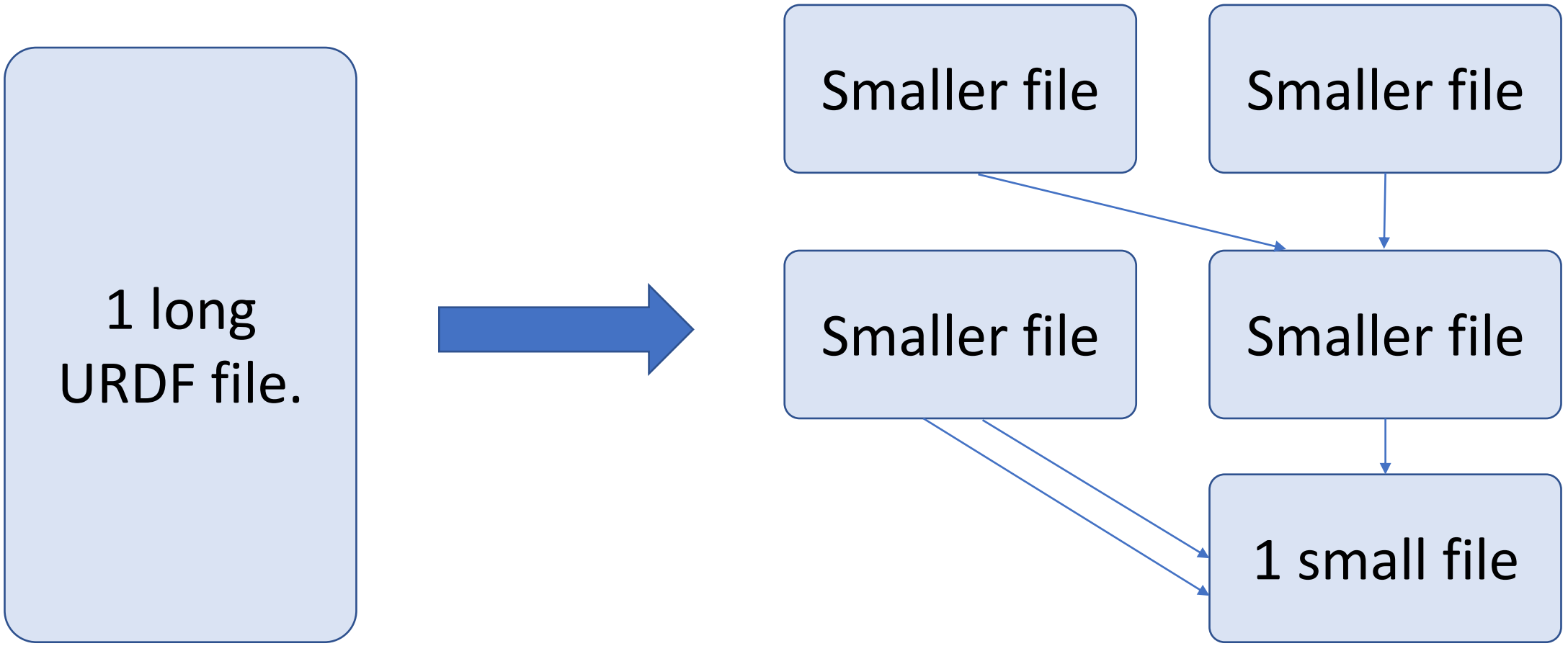
Robot Parameters

# Limitation of URDF

We cannot use common parameters.

We have to write every single part regardless of their copies.

# What if ?


1 long
URDF file.
Smaller file
Smaller file
Smaller file
Smaller file
1 small file

# xacro : XML macros

# What can we do with xacro?

We can use xacro to define macros in xml format.

URDF is also an xml.

# Processing xacro file in command line

Before generating a file, it is time-saving to test whether the resultant URDF file can be correctly generated. One can use the following command to test their xacro file.

```
>>xacro [xacro_file.xacro] [optional arguments] > [urdf_file.urdf]
```

Open the generated URDF file to check the content. If errors exist in the xacro file, “xacro” process will display the corresponding error message.

# Processing xacro file in Python Launch script

To obtain “robot_description” from the xacro file, one can use the following function.

```
robot_description = xacro.process_file(full_path_to_xacro_file).toxml()
```

This is equivalent to reading a URDF file.

# Performing basic arithmetics

xacro recognizes a tag “${}” as performing calculation. Therefore, you do not have to compute the parameters manually.

```
<origin xyz="${1/2} ${2+3} ${2*4}" />
```

```
<origin xyz="0.5 5 8" />
```

# Defining a macro

"macro" is a substitution. One can also pass parameters to a macro.

```
<xacro:macro name="something" params="param1 param2 param3">

      <visual>
            <geometry>
                  <box size="${param1} ${param2} ${param3}">
            </geometry>
      </visual>
</xacro:macro>
```

Anything can be put inside the macro.

# Using the macro

```
<xacro:something param1="0.2" param2="0.3" param3="0.4"/>
```

```
<visual>
     <geometry>
          <box size="0.2 0.3 0.4">
     </geometry>
</visual>
```

# Using the macro

```
<robot>
   <link name="base"/>
   <xacro:robot_link index="0"/>
   <xacro:robot_link index="1"/>
   <xacro:robot_link index="2"/>
   <xacro:robot_link index="3"/>
   <link name="end_effector"/>
   <xacro:base_offset/>
   <xacro:robot_joint index="1"/>
   <xacro:robot_joint index="2"/>
   <xacro:robot_joint index="3"/>
   <xacro:end_effector_offset/>
</robot>
```

# Including another file

When you include another xacro file, you essential put all its content in the file.

```
<xacro:include filename="$(find package_name)/sub_folder/other_file.xacro"/>
```

Anything from the included file will be put in this file.

# Declaring a property

A “property” in xacro is global. Once it is declared, any subsequent lines can access its value.

```
<xacro:property name="foo" value="1.5"/>
```

# Using the property

One can use the calculation tag to substitute the value of the property

```
<origin xyz="${foo} ${foo/2} ${foo*1.5}" />
```

# Open dictionary from YAML

“find” acts like “get_package_share_directory”

```
<xacro:property name="params_path" value="$(find name_description)/config/parameters.yaml"/>

<xacro:property name="my_dict" value="${xacro.load_yaml(params_path)}"/>


<xacro:property name="param_1" value="${my_dict['key_1']}"/>

<xacro:property name="param_2" value="${my_dict['key_2']['key_a']}"/>
```

# Using “if”

“if” allows one to conditionally create an xacro expression. If the value is evaluated as True, the expression inside will be generated.

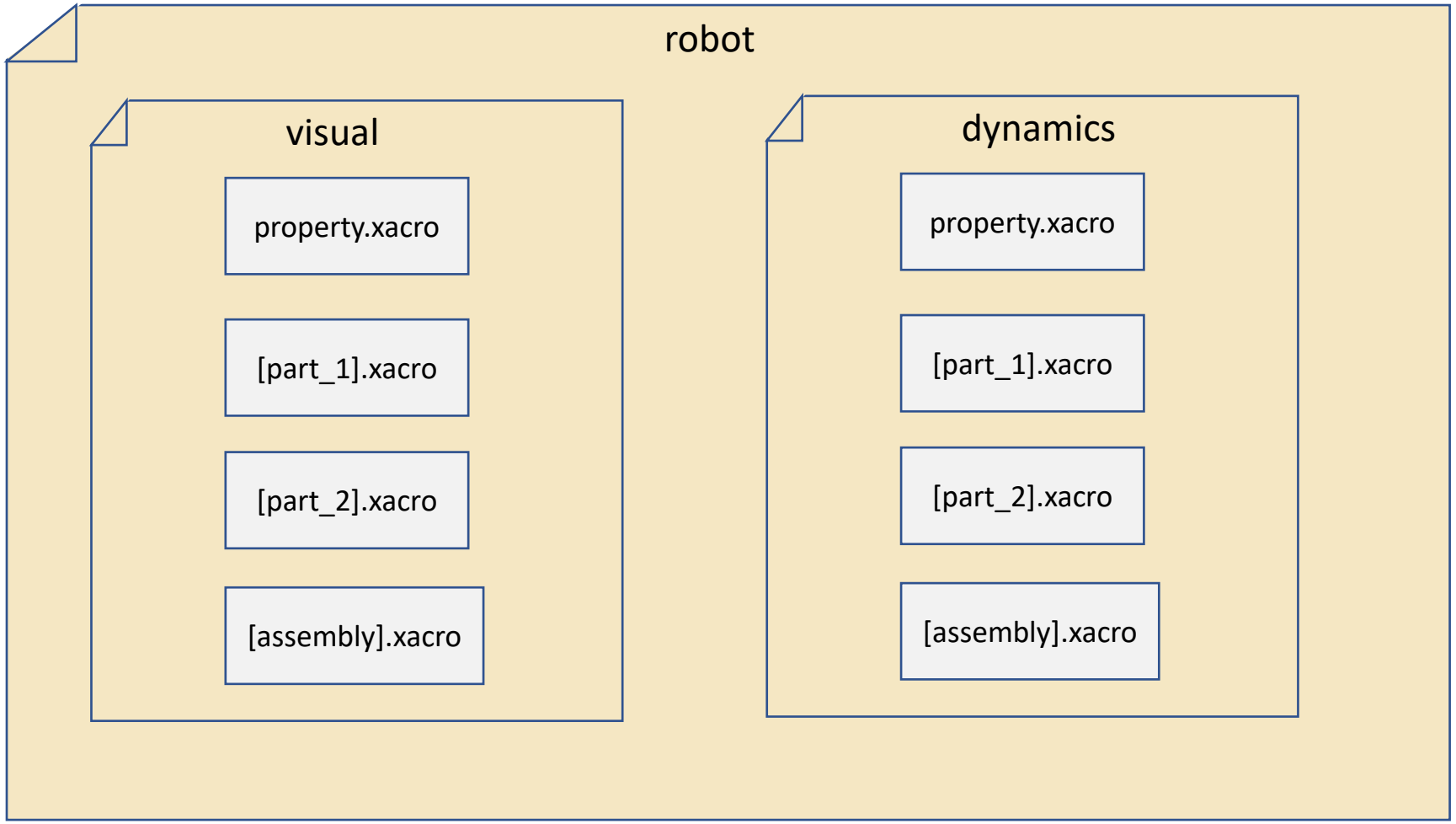
```
<xacro:if value="${name=='left'}">
      <xacro:property name="y" value="${d}">
</xacro:if>
```

# Using “unless”

“unless” also allows one to negate the condition. If the value is evaluated as False, the expression inside will be generated.

```
<xacro:unless value="${name=='left'}">
      <xacro:property name="y" value="${-d}">
</xacro:unless>
```

# Possible xacro file management




xacro : XML macros

# property.xacro

```
<?xml version='1.0'?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:property name="description_package" value="example_description"/>

  <!-- File paths-->
  <xacro:property name="params_path" value="$(find ${description_package})/config/parameters.yaml"/>
  <xacro:property name="param_dict" value="${xacro.load_yaml(params_path)}"/>

  <!-- Mathematics Constants -->
  <xacro:property name="PI" value="3.1415926535897931"/>

  <!-- Robot Properties -->
  <xacro:property name="LENGTH" value="${param_dict['length']}"/>
  <xacro:property name="WIDTH" value="${param_dict['width']}"/>
  <xacro:property name="OFFSET" value="${param_dict['offset']}"/>

</robot>
```

xacro : XML macros

# [part_1].xacro

```
<?xml version='1.0'?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name= "my_macro" params="id">
        <visual>
            <origin xyz="${OFFSET[id-1]}"/>
            <geometry>
                <mesh filename="file://$(find example_description)/meshes/link_${id-1}.stl"/>
            </geometry>
            <material name="link_color">
                <color rgba="${COLOR[id-1]}"/>
            </material>
        </visual>
  </xacro:macro>

</robot>
```

xacro : XML macros

# [assembly].xacro

```
<?xml version='1.0'?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="example" >
  <xacro:arg name="robot_name" default=""/>
  <xacro:property name="namespace" value="$(arg robot_name)"/>

  <xacro:include filename="$(find example_description)/robot/visual/properties.xacro"/>
  <xacro:include filename="$(find example_description)/robot/visual/part_1.xacro"/>

  <link name=base>
      <xacro:my_macro id="0"/>
  </link>

  <link name=link_1>
      <xacro:my_macro id="1"/>
  </link>

  <joint name="joint_1">
      . . .
  </joint>

</robot>
```

xacro : XML macros

# Processing xacro file in command line

Before generating a file, it is time-saving to test whether the resultant URDF file can be correctly generated. One can use the following command to test their xacro file.

```
>>xacro [xacro_file.xacro] [optional arguments] > [urdf_file.urdf]
```

Open the generated URDF file to check the content. If errors exist in the xacro file, “xacro” process will display the corresponding error message.

# Processing xacro file in Python Launch script

To obtain “robot_description” from the xacro file, one can use the following function.

```
robot_description = xacro.process_file(full_path_to_xacro_file).toxml()
```

This is equivalent to reading a URDF file.

# What do we have so far ?

- A way to display a robot in rviz

# What do we have so far ?

- A way to display a robot in rviz

What if we want to display 2 identical robots simultaneously?

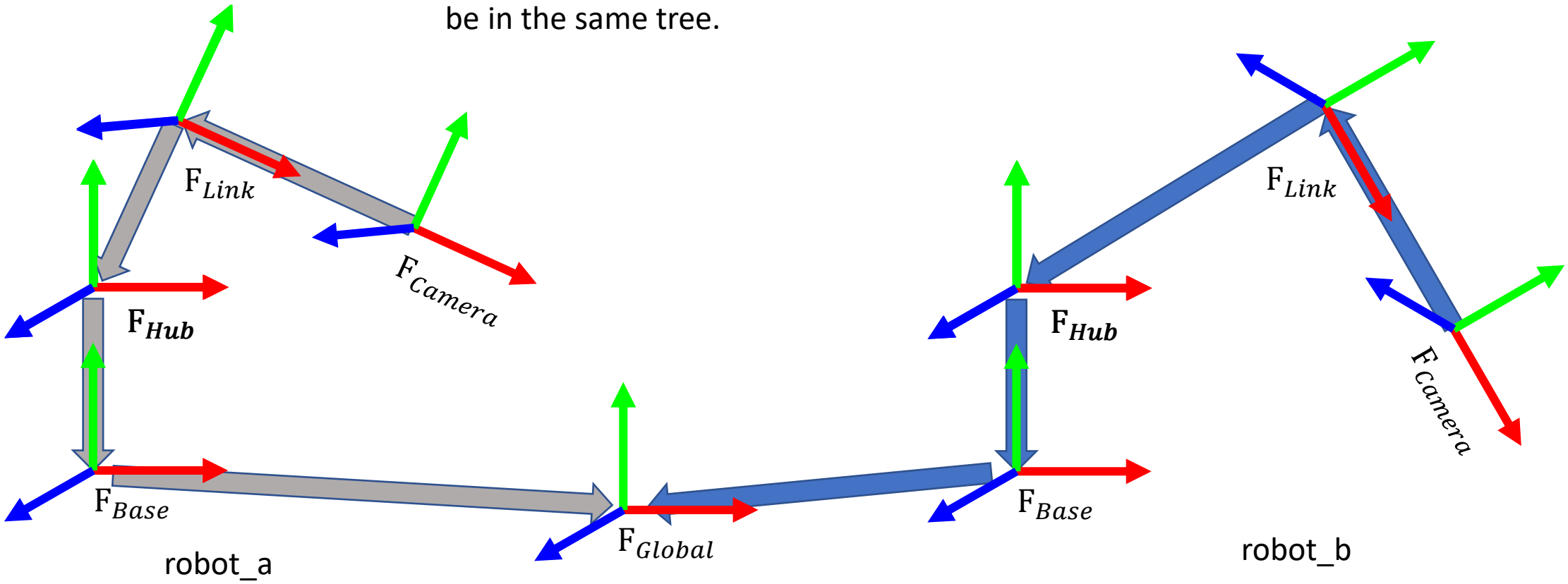
# Concern 1: File Duplication

One can create a file that auto-generates a new set of robot description xacro files with the added namespace.

This works okay if we have several robots. But we can be more efficient.

# Concern 2: Connecting tree

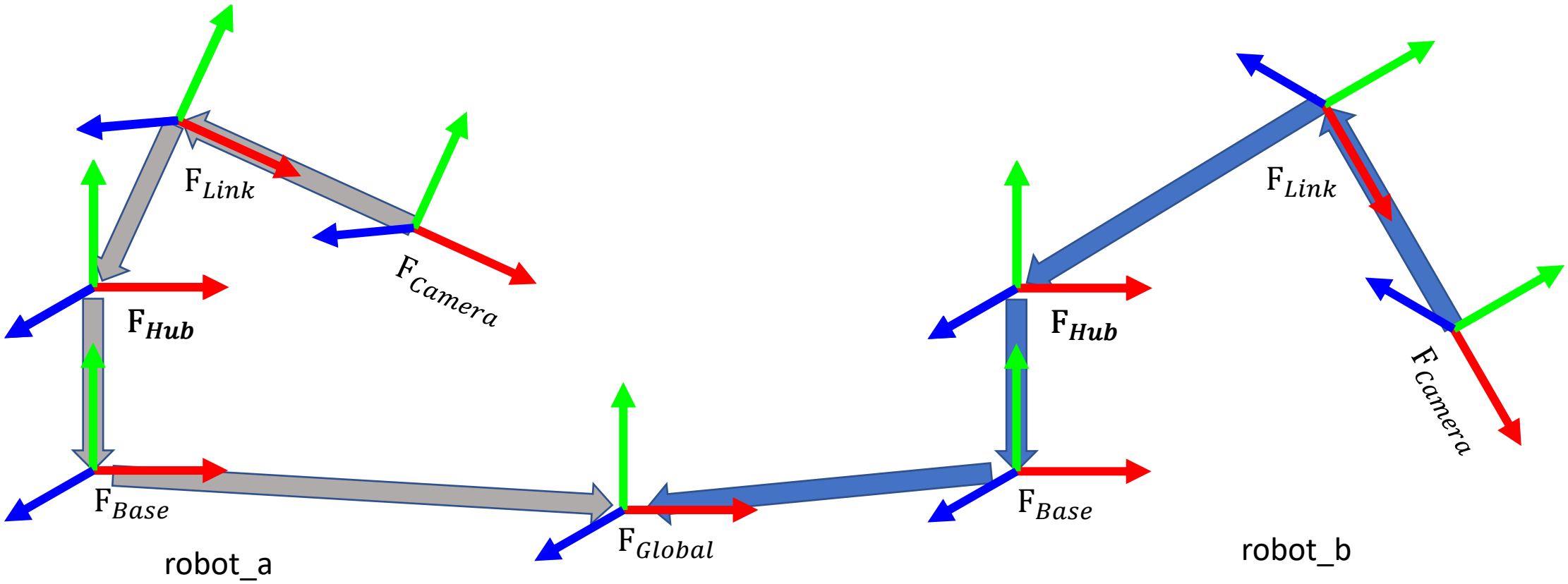
To display both robots together, they must be in the same tree.




xacro : XML macros

# Concern 2: Connecting tree

Their frames must be unique !!!

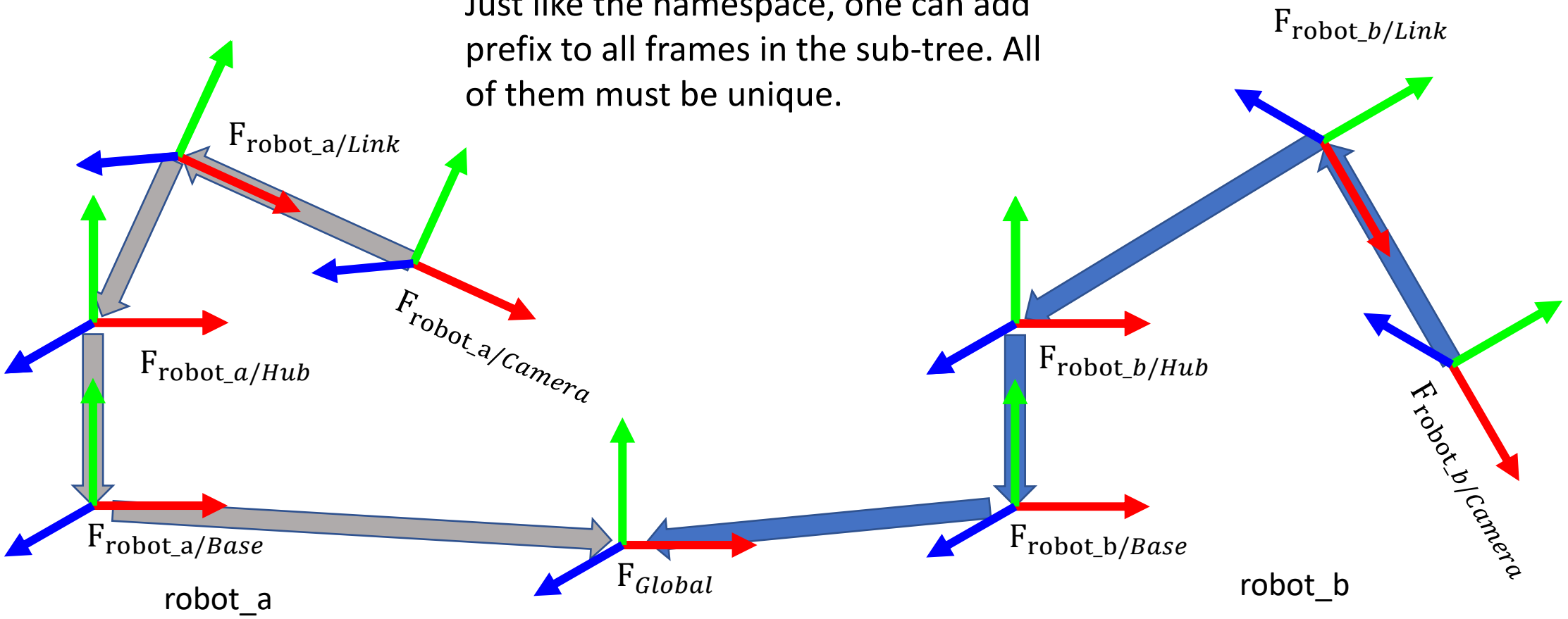



xacro : XML macros

# Frame Prefix

# Resolving conflict using Frame Prefix

Just like the namespace, one can add prefix to all frames in the sub-tree. All of them must be unique.



# Parameters of “robot_state_publisher”

```
robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        namespace=namespace_str,
        parameters=[{
            'robot_description’:’turtle1',
            'frame_prefix’: ‘turtle1'+'/'
        }]
    )
```

These will automatically modify the “robot_description” topic and “tf”.

Frame Prefix

# Passing namespace as a Launch Argument

```
robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        namespace=namespace_str,
        parameters=[{
            'robot_description’:namespace,
            'frame_prefix’: namespace+'/'
        }]
    )
```

A Launch configuration cannot be appended with a Python string.

Frame Prefix

# Passing namespace as a Launch Argument

```
robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        namespace=namespace_str,
        parameters=[{
            'robot_description’:namespace,
            'frame_prefix’: namespace+'/'
        }]
    )
```

We can use Opaque function.

Frame Prefix

# Passing namespace as a Launch Argument

```
frame_prefix = LaunchConfiguration('frame_prefix',default= [namespace, '/'])

robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        namespace=namespace_str,
        parameters=[{
           'robot_description':namespace,
           'frame_prefix': frame_prefix
        }]
    )
```

We can use Opaque function.
But we can also do this.

This allows us to combine Launch configuration with a Python string. The result is another Launch configuration.

Frame Prefix

# Modifying rviz programatically

Unfortunately, modifying rviz configuration file requires us to strictly use Python string.  An OpaqueFunction is out only option.

```
new_rviz_file = modify_rviz_config(description_package,'config.rviz',namespace)
```

This must be a Python string.

# Modifying rviz config

```
from launch import LaunchDescription, LaunchContext
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node
from launch.substitution import LaunchConfiguration
from ament_index_python.packages import get_package_share_directory

def render_namespace(context:LaunchContext,launch_description:LaunchDescription,namespace:LaunchConfiguration) -> None
    namespace_str = context.perform_substitution(namespace)

    my_pkg = get_package_share_directory('example_description')
    config_path = os.path.join(my_pkg,'config','config.rviz')
    new_config_path = os.path.join(my_pkg,'config', 'config_'+namespace_str+'.rviz')

    modify_rviz_namespace(config_path,new_config_path,namespace_str)

    rviz = Node(
        package = 'rviz2',
        executable = 'rviz2',
        arguments = ['-d', new_config_path ]
    )

    launch_description.add_action(rviz)
```

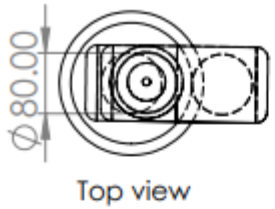
Frame Prefix

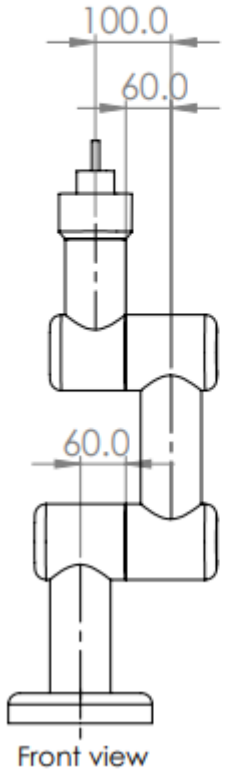
# Summary

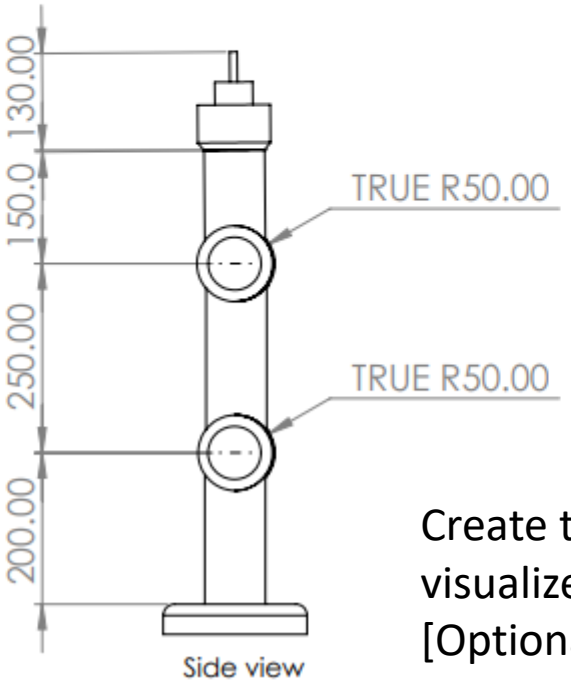
- purpose
- interface
- coordinate frame
- visualization
- URDF
- XACRO
- Frame prefix

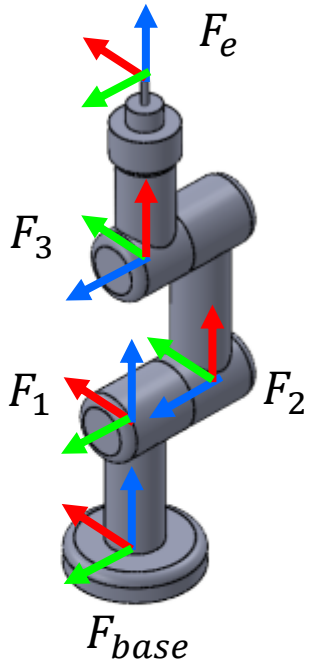
# Exercise

# Exercise




Create the robot “kinematics” model and visualize it in rviz.
[Optional] It should take namespace as well.

# Example Repo- Branch: robot-modelling-solution

https://github.com/kittinook/FRA501/tree/robot-modelling-solution

Download & add “example_description” to the “src” directory of your workspace. Then build these packages.

Follow the instruction on README.md