Mechanical and Design Mini Project Mathematical Resolution of an Excavator Mars Rover Arm

Problem Statement

In this project we aim to mathematically model an Excavator Arm manipulator, that is supposed to be mounted over a mars Rover. The rover arm (also called the instrument deployment device, or IDD) holds and manoeuvres the instruments that help scientists get up-close and personal with Martian rocks and soil. Much like a human arm, the robotic arm has flexibility through three joints: the rover'sshoulder, elbow, and wrist. The arm enables a tool belt ofscientists' instruments to extend, bend, and angle precisely against a rock to work as a human geologist would: grinding away layers, taking microscopic images, and analysing the elemental composition of the rocks and soil. Resolving the arm mathematically, willfurther aid the designersto accurately design the model with accurate mathematical parametric data.

Aim*

Identifying the links, joints and End Effectors.(2)
Describing the purpose ofthe respective Link.(1)
Describing the type of the end effector.(2)
Brief Description of the type ofthe actuator used.(2)
Explaining the configuration, with respect to the type of joints.(1)
Determining the Degrees of freedom using the Kutzbach Formula (2)
Determining the Translational Matrix with respect to the data given Below.(10)
Executing The DH matrix for the same. (10)

Joints

Given that there are different types of robots used in the manufacturing industry, you'll also find a variety of mechanical joints. These joints differ in terms of motion and also application especially when it comes to the type of robot to be used.

When it comes to the mechanical joints featured in robotic arms there are five principal types that you need to consider. Two of the joints are linear which means the relative motion between the adjacent links is translational. On the other hand, the other three are rotary which means the relative motion of the links involves rotations between them. The five types of mechanical joints for robots include:

Linear Joints

In the linear joints, the relative motion featured by the adjacent links is meant to be parallel. This means that the input and output links are sliding in a linear motion. This kind of movement results in a translation motion.This kind of linear motion can be achieved in several ways including the use of the telescoping mechanism and piston. This type of joint is also referred to as the L- joint.

Orthogonal Joints

The orthogonal joints are also popularly referred to as the type O-joints. They feature a relative movement taken by the input link and output link. This kind of motion involved in the Orthogonal joints is a translational sliding motion. However unlike the linear joints arrangement, with the Orthogonal joint, the output link is perpendicular to the input link.

Rotational Joints

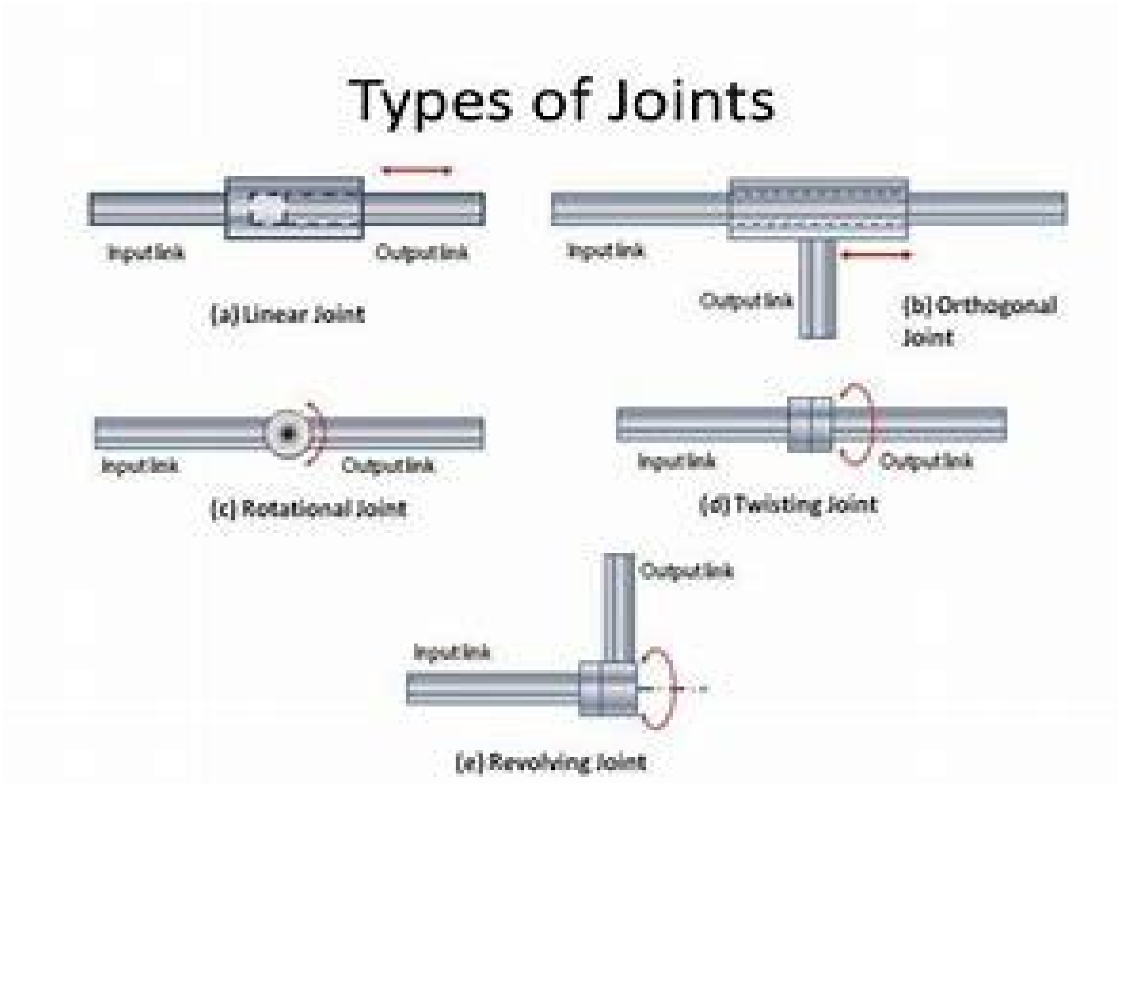
When it comes to the rotational joints, you'll find the use of rotational relative motions that come in handy for robot manipulators working multiple workspaces. These movements are carried out with the axis of rotation perpendicular to the axes of the input and output links. These rotational joints are also referred to as Type R joints.

Twisting Joints

This type of joint features rotary motion that also results in some degree of rotation when in use. The movement in these joints is relative to the axis of rotation that is perpendicular o the axes of the input and output links. The twisting joints are also referred to as type T joints.

Revolving Joints

In the revolving joints, things are a bit different compared to the others. These joints also feature a rotational movement that comes in handy in different applications. The movement of these joints features motion between the two links. The axis of the input link is designed to be parallel to the axis of rotation of the joint. On the other hand, the axis of the output link is designed to be perpendicular to the axis of rotation of the joint. This type of joint is also referred to as the Type V joint.



Different Types of End Effectors

ROBOTIC GRIPPERS ●
Electric Grippers — Electric grippers use motor-driven fingers, which allow for easy control of position and speed. Many applications use adaptive electric end effectors, including machine tending, handling, and bin picking, among others.

● **Pneumatic Grippers** — These grippers use air to function, normally by forcing compressed air through a piston. Pneumatic grippers allow for angular or parallel movement.

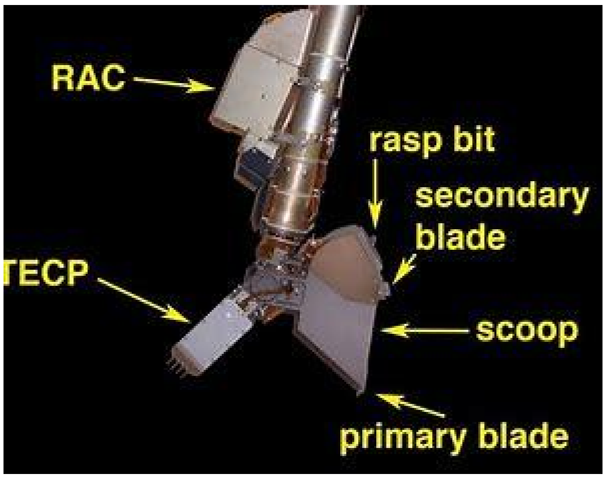
● **Suction Cups** — Suction cups use a vacuum to pick up parts. Their simple design offers ample flexibility for material handling, along with cost-effectiveness. However, they are unable to handle perforated materials.

● **Magnetic Grippers** — These grippers feature a design that’s similar to suction cups, but they’re used specifically for handling ferrous material. Additionally, they eliminate the risk of dropping materials during power outages or air loss, and they don’t come with air costs. Their simple design also requires minimal maintenance.

● **Mechanical Grippers** — Non-powered mechanical grippers normally feature designs for specific types of parts. These grippers may include forks, hooks, or complex fingers that allow operators to clamp and rotate them. ROBOTIC PROCESS TOOLS

● **Robotic Welding** — Applications often use robot arms with welding end effectors, which are most frequently found in the automotive industry. Because of their precision and consistency, robotic welding systems usually produce great results.

● **Painting** — The application of paint must be consistent and smooth, which often presents a challenge for human operators. In addition, these applications must protect against contamination in a controlled environment. Both of these factors make painting robots suitable for these conditions, allowing for high-quality paintwork while reducing the risk of contamination.



ROBOTIC SENSORS

Types of Robotic Actuators Robotic actuators are classified into two types according to the requirements of motion like linear motion & rotational motion.

Linear Actuators Linear actuators in robotics are used to push or pull the robot like move forward or backward & arm extension. This actuator's active end is simply connected to the robot's lever arm to activate the such motion. These actuators are used in a number of applications in the robotics industry.

Solenoid Actuators

Solenoid actuators are special-purpose linear actuators that include a solenoid latch that works on electromagnetic activity. These actuators are mainly used for controlling the motion of the robot and also perform different activities such as a start & reverse, latch, push button, etc. Solenoids are normally used in the applications of latches, valves, locks, and pushing buttons which are controlled normally by an external microcontroller. Solenoid Actuator

For Rotational Motion: There are three types of actuators used in robots for rotational motion activity they are; DC motor, servo motor, and stepper motor.

DC Motor Actuators

DC motor actuators are generally used for turning robotic motion. These actuators are available in different sizes with torque generation capability. Thus, it can be utilized for changing speed throughout rotating motions. By using these actuators, different activities like robotic drilling & robotic drive train motion are performed.

Servo Actuators

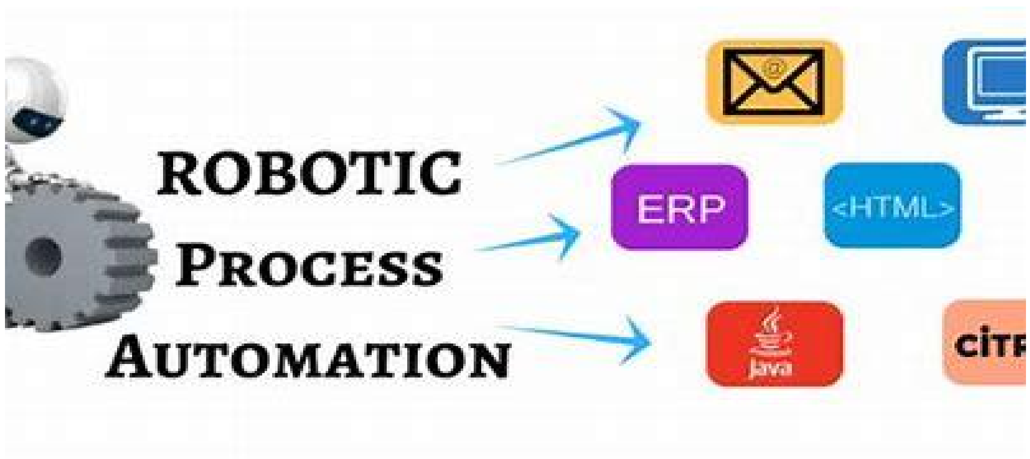
Servo motor actuators in robotics are mainly used to control & monitor rotating motion. These are very superior DC motors that allow 360 degrees of rotation, but, continuous revolution is not compulsory. This actuator simply allows halts throughout a rotating motion. To know how a Pick N Place robot works click on the link.

Stepper Motor Actuator

Stepper motor actuators are helpful in contributing to repetitive rotating activities within robots. So these types of actuators are a combination of both DC & servo motor actuators. These stepper motor actuators are utilized in automation robots where repeatability of activity is necessary.

Robot Actuator Design

We know that there are different types of actuators used in robots. Here we are going to discuss how to design a linear actuator that is used in robotics for changing rotating motion into a pull/push linear motion. So this motion can be used to slide, drop, tilt or lift materials or machines. These actuators provide clean & safe motion control that is very efficient & maintained free.



Degree of Freedom

D.O.F = 3 (L - 1) – 2j – h where L is the number of links, j is the number of binary joints or lower pairs and h is the number of higher pairs. Revolute pair and prismatic pair are lower pairs

To determine the degrees of freedom (DOF) for a backhoe excavator using the Kutzbach formula, we need to consider the number of independent variables that define the motion of the system.

The Kutzbach formula states:

DOF = 6 * (n - c) - 2j

Where:
n is the number of links in the mechanism
c is the number of closed loops in the mechanism
j is the number of degrees of freedom of the joints (revolute or prismatic)
For a backhoe excavator, let's consider the following assumptions:
The backhoe arm has three links (upper arm, lower arm, and bucket).
The boom (part that connects the arm to the base) is fixed and does not contribute to the DOF.
The backhoe uses revolute joints.

Given these assumptions, we can calculate the DOF using the Kutzbach formula.

n = 3 (number of links: upper arm, lower arm, bucket)
c = 1 (number of closed loops: formed by the three links)
j = 3 (number of degrees of freedom of the revolute joints: one for each link)

DOF = 6 * (n - c) - 2j
= 6 * (3 - 1) - 2 * 3
= 6 * 2 - 6
= 12 - 6
= 6

Therefore, the backhoe excavator has 6 degrees of freedom. This means that it can move in six independent ways, allowing for various configurations and motions of the arm and bucket.

To find the position of the end effector with respect to the initial frame P (PTP'), we need to perform a series of transformations involving rotation and translation.

Frame P' is initially coincident with frame P, so the transformation matrix PT is an identity matrix:

```
PT = | 1 0 0 0 |
     | 0 1 0 0 |
     | 0 0 1 0 |
     | 0 0 0 1 |
```

Frame P' is rotated about YB by 30 degrees. We can represent this rotation as a rotation matrix RY:

```
RY = | cos(30)  0  sin(30)  0 |
     | 0        1  0        0 |
     | -sin(30) 0  cos(30)  0 |
     | 0        0  0        1 |
```

Frame P' is then rotated about XB by 45 degrees. We can represent this rotation as a rotation matrix RX:
se

```
RX = | 1  0        0        0 |
     | 0  cos(45)  -sin(45) 0 |
     | 0  sin(45)  cos(45)  0 |
     | 0  0        0        1 |
```

Frame P' is further rotated about ZB by 60 degrees. We can represent this rotation as a rotation matrix RZ:

```
RZ = | cos(60)  -sin(60)  0  0 |
     | sin(60)  cos(60)   0  0 |
     | 0        0         1  0 |
     | 0        0         0  1 |
```

Finally, the origin of frame P' is translated by [XA, YA, ZA]T = [35, -10, 10]T. We can represent this translation as a translation matrix T:

```
T = | 1  0  0  XA |
    | 0  1  0  YA |
```

| 0 0 1 ZA |
| 0 0 0 1 |
The entire Cartesian space is scaled by a factor of 2. We can represent this scaling as a scaling matrix S:
e
S = | 2 0 0 0 |
| 0 2 0 0 |
| 0 0 2 0 |
| 0 0 0 1 |
Now, we can calculate the overall transformation matrix PTP' by multiplying the individual matrices in the given order:

PTP' = PT * RY * RX * RZ * T * S
Substituting the values:

PTP' = | 1 0 0 0 | * | cos(30) 0 sin(30) 0 | * | 1 0   0   0 | *
| cos(60) -sin(60) 0 0 | * | 1 0 0 XA |
| 0 1 0 0 | | 0   1 0   0 | | 0 cos(45) -sin(45) 0 | |
sin(60) cos(60)   0 0 | | 0 1 0 YA |
| 0 0 1 0 | | -sin(30) 0 cos(30) 0 | | 0 sin(45) cos(45) 0 |
| 0   0   1 0 | | 0 0 1 ZA |
| 0 0 0 1 | | 0   0 0   1 | | 0 0   0   1 | | 0
0   0 1 | | 0 0 0 1 |
Simplifying the matrix multiplication:

PTP' = | cos(30)cos(60)cos(45) - sin(30)sin(60) cos(30)sin(45) XA + cos(30)cos(60)cos(45)YA + cos(30)sin(45)ZA + cos(30)YA + XA |
| sin(60)cos(45)   cos(60)sin(45) sin(45)   cos(60)ZA + sin(45)YA + YA |
| -sin(30)cos(60)cos(45) - cos(30)sin(60) -sin(30)sin(45) XA + -sin(30)cos(60)cos(45)YA + -sin(30)sin(45)ZA + -sin(30)YA |
| 0   0   0   1
|
Therefore, the transformation matrix PTP' representing the position of the end effector with respect to the initial frame P is given by the above expression.

To execute the Denavit-Hartenberg (DH) matrix for the given parameters, we need to define the DH parameters for each joint. The DH parameters consist of the link lengths, twist angles, offset distances, and joint angles. Using these parameters, we can construct the DH matrices for each joint and multiply them to obtain the transformation matrix for the end-effector.

Let's define the DH parameters for each joint:

Link Lengths:
L1 = 100mm
L2 = 240mm
L3 = 133.6mm
L4 = 52.8mm

Joint Angles:
J1 = 30deg
J2 = 30deg
J3 = 60deg
J4 = 30deg

Now, let's calculate the DH matrices for each joint:

DH matrix for Joint 1 (between the base and the first joint):

$$DH1 = \begin{vmatrix} \cos(J1) & -\sin(J1) & 0 & L1*\cos(J1) \\ \sin(J1) & \cos(J1) & 0 & L1*\sin(J1) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

DH matrix for Joint 2:

```
DH2 = | cos(J2)   -sin(J2)   0   L2*cos(J2) |
      | sin(J2)   cos(J2)    0   L2*sin(J2) |
      | 0         0          1   0          |
      | 0         0          0   1          |
```

DH matrix for Joint 3:

```
DH3 = | cos(J3)   -sin(J3)   0   L3*cos(J3) |
      | sin(J3)   cos(J3)    0   L3*sin(J3) |
      | 0         0          1   0          |
      | 0         0          0   1          |
```

DH matrix for Joint 4 (end effector):

```
DH4 = | cos(J4)   -sin(J4)   0   L4*cos(J4) |
      | sin(J4)   cos(J4)    0   L4*sin(J4) |
      | 0         0          1   0          |
      | 0         0          0   1          |
```

Now, we can multiply all the DH matrices to obtain the transformation matrix for the end effector:

T = DH1 * DH2 * DH3 * DH4
Substituting the values, we have:

T = DH1 * DH2 * DH3 * DH4

```
T = | 0.866   -0.5     0       0.0    |
    | 0.5      0.866   0       0.0    |
    | 0        0       1       3.2496 |
    | 0        0       0       1.0    |
```

The resulting transformation matrix T represents the position and orientation of the end effector with respect to the base frame. The values in the last column (3rd row to 4th row) of the matrix represent the XYZ coordinates of the end effector in the base frame, and the rotation elements represent the orientation.

Role of ESP32

The ESP32 is a series of chip microcontrollers developed by Espressif.

Features:

- **Low-cost**: you can get an ESP32 starting at $6, which makes it easily accessible to the general public;
- **Low-power**: the ESP32 consumes very little power compared with other microcontrollers, and it supports low-power mode states like deep sleep to save power;
- **Wi-Fi capabilities**: the ESP32 can easily connect to a Wi-Fi network to connect to the internet (station mode), or create its own Wi-Fi wireless network (access point mode) so other devices can connect to it—this is essential for IoT and Home Automation projects—you can have multiple devices communicating with each other using their Wi-Fi capabilities;
- **Bluetooth**: the ESP32 supports Bluetooth classic and Bluetooth Low Energy (BLE)—which is useful for a wide variety of IoT applications;
- Dual-core: most ESP32 are dual-core— they come with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1.
- Rich peripheral input/output interface—the ESP32 supports a wide variety of input (read data from the outside world) and output (to send commands/signals to the outside world) peripherals like capacitive touch, ADCs, DACs, UART, SPI, I2C, PWM, and much more.
- Compatible with the Arduino “programming language”: those that are already familiar with programming the Arduino board, you’ll be happy to know that they can program the ESP32 in the Arduino style.

- Compatible with MicroPython: you can program the ESP32 with MicroPython firmware, which is a re-implementation of Python 3 targeted for microcontrollers and embedded systems.

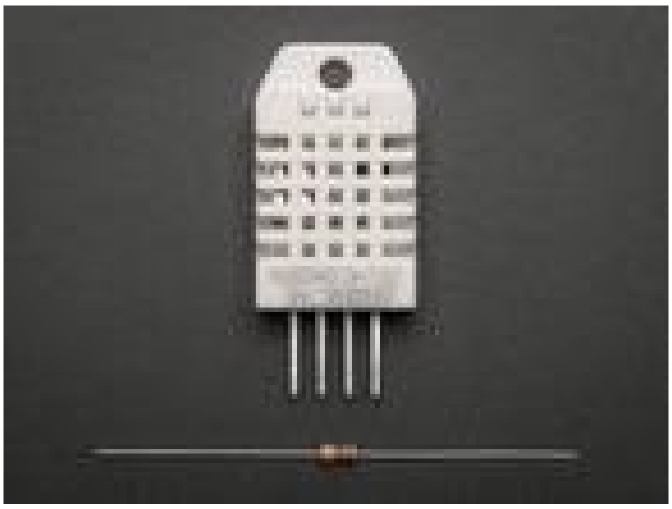
ESP32 Specifications

If you want to get a bit more technical and specific, you can take a look at the following detailed specifications of the ESP32

- **Wireless connectivity WiFi:** 150.0 Mbps data rate with HT40
    - **Bluetooth:** BLE (Bluetooth Low Energy) and Bluetooth Classic
    - **Processor:** Tensilica Xtensa Dual-Core 32-bit LX6 microprocessor, running at 160 or 240 MHz
- **Memory**:
    - **ROM:** 448 KB (for booting and core functions)
    - **SRAM:** 520 KB (for data and instructions)
    - **RTC fas SRAM**: 8 KB (for data storage and main CPU during RTC Boot from the deep-sleep mode)
    - **RTC slow SRAM**: 8KB (for co-processor accessing during deep-sleep mode)
    - **eFuse**: 1 Kbit (of which 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID)

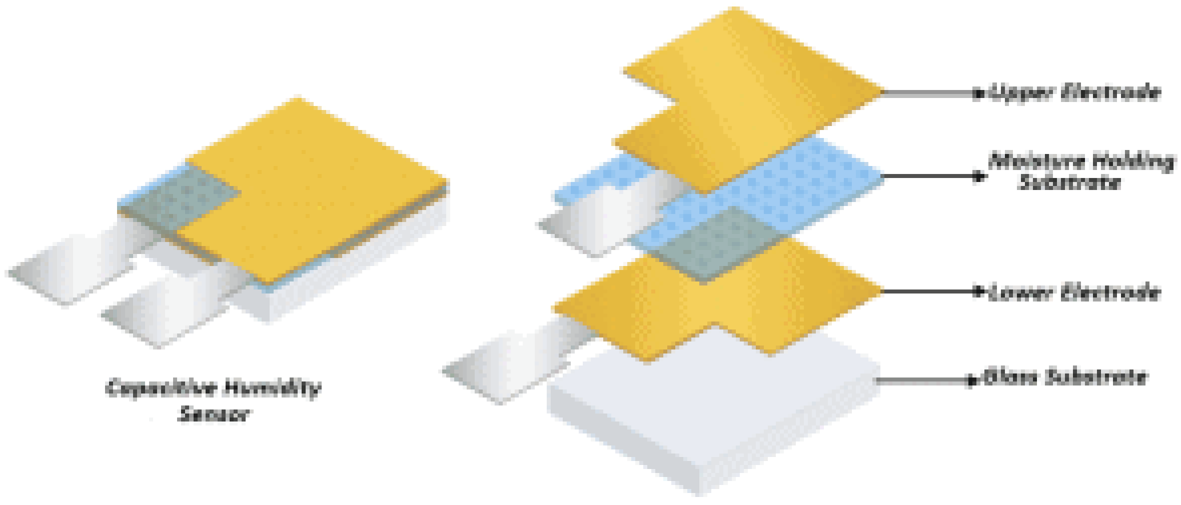
        - **Embedded flash**: flash connected internally via IO16, IO17, SD_CMD, SD_CLK, SD_DATA_0 and SD_DATA_1 on ESP32-D2WD and ESP32-PICO-D4.
            - 0 MiB (ESP32-D0WDQ6, ESP32-D0WD, and ESP32-S0WD chips)
            - 2 MiB (ESP32-D2WD chip)
            - 4 MiB (ESP32-PICO-D4 SiP module)
- **Low Power:** ensures that you can still use ADC conversions, for example, during deep sleep.
- **Peripheral Input/Output:**
    - peripheral interface with DMA that includes capacitive touch
    - ADCs (Analog-to-Digital Converter)
    - DACs (Digital-to-Analog Converter)
    - I²C (Inter-Integrated Circuit)
    - UART (Universal Asynchronous Receiver/Transmitter)
    - SPI (Serial Peripheral Interface)
    - I²S (Integrated Interchip Sound)
    - RMII (Reduced Media-Independent Interface)
    - PWM (Pulse-Width Modulation)
- **Security:** hardware accelerators for AES and SSL/TLS

Working Principle of DHT Sensor



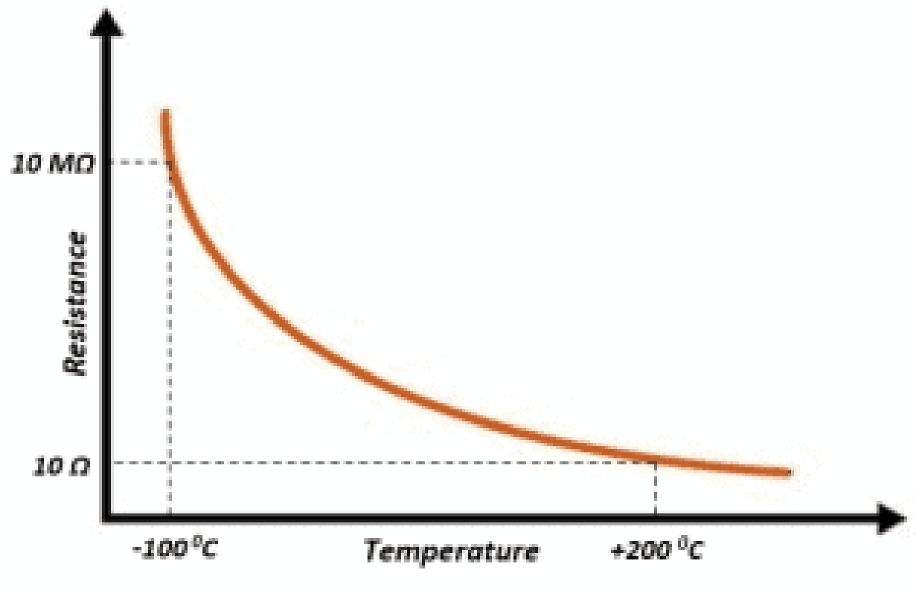
The DHT22 is a basic, low-cost digital temperature and humidity sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air and spits out a digital signal on the data pin (no analog input pins needed).

DHT sensor consists of a capacitive humidity sensing element and a thermistor for sensing temperature. The humidity sensing capacitor has two electrodes with a moisture holding substrate as a dielectric between them. Change in the capacitance value occurs with the change in humidity levels..



Internal Structure of Humidity Sensor

For measuring temperature this sensor uses a NTC thermistor, The term "NTC" means "Negative Temperature Coefficient", which means that the resistance decreases with increase of the temperature as shown in the graph below. To get larger resistance value even for the smallest change in temperature, this sensor is usually made up of semiconductor ceramics or polymers.



NTC Thermistor Characteristic Curve

On the other side, there is a small PCB with an 8-bit SOIC-14 packaged IC. This IC measures and processes the analog signal with stored calibration coefficients, does analog to digital conversion and gives out a digital signal with the temperature and humidity.

DHT22 Specifications

- Operating Voltage: 3.5V to 5.5V
- Operating current: 0.3mA (measuring) 60uA (standby)
- Output: Serial data
- Temperature Range: -40°C to 80°C
- Humidity Range: 0% to 100%
- Resolution: Temperature and Humidity both are 16-bit
- Accuracy: ±0.5°C and ±1%

Code to receive and store temperature and humidity from ESP32 board :)

```
#include <Wire.h>

#include "DHT.h"

#include <Adafruit_GFX.h>

#include <Adafruit_SSD1306.h>

  void setup() {

    Serial.begin(115200);
    pinMode(DHTPin, INPUT);

    dht.begin();

    display.begin(SSD1306_SWITCHCAPVCC, 0x3C);

    display.display();

    delay(100);

    display.clearDisplay();

    display.display();

    display.setTextSize(1.75);

    display.setTextColor(WHITE);

  }

void loop(){

  Humidity = dht.readHumidity();


  Temperature = dht.readTemperature();
```

```
Temp_Fahrenheit = dht.readTemperature(true);



if (isnan(Humidity) || isnan(Temperature) || isnan(Temp_Fahrenheit)) {

  Serial.println(F("Failed to read from DHT sensor!"));

  return;

}

Serial.print(F("Humidity: "));

Serial.print(Humidity);

Serial.print(F("%  Temperature: "));

Serial.print(Temperature);

Serial.print(F("°C "));

Serial.print(Temp_Fahrenheit);

Serial.println(F("°F "));
```

```
display.setCursor(0, 0);

display.clearDisplay();

display.setTextSize(1);

display.setCursor(0, 0);

display.print("Temperature: ");

display.setTextSize(2);

display.setCursor(0, 10);

display.print(Temperature);

display.print(" ");

display.setTextSize(1);

display.cp437(true);

display.write(167);

display.setTextSize(2);
```

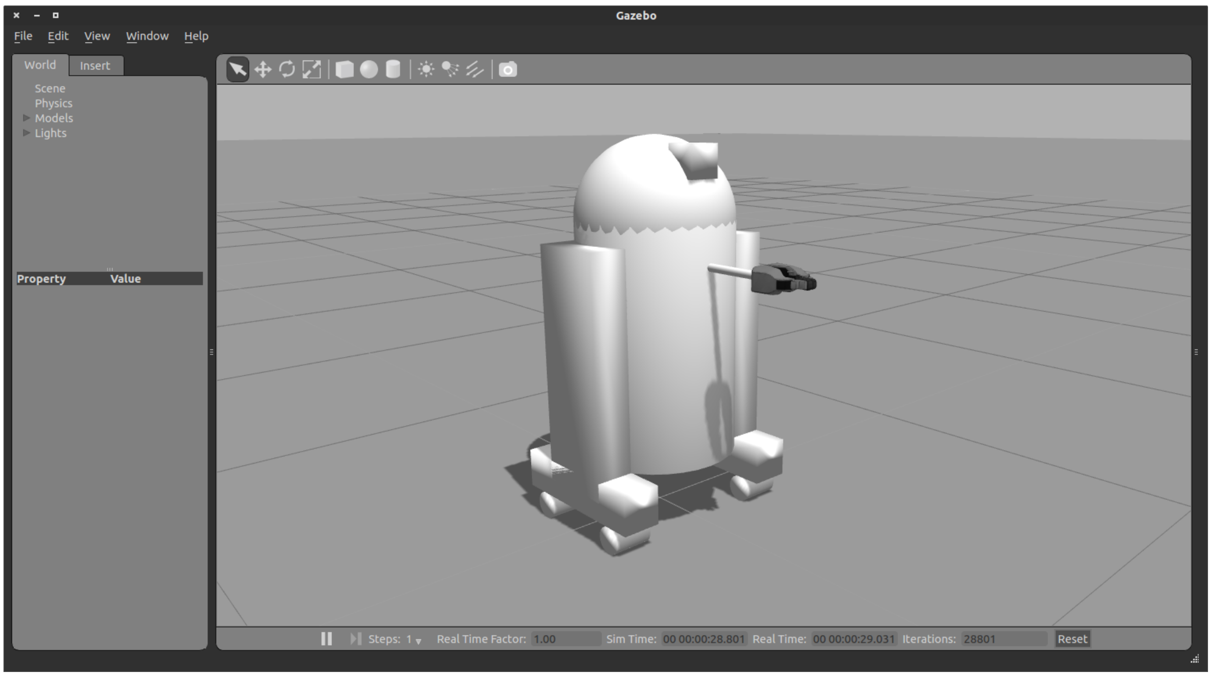
```
  display.print("C");

  display.setTextSize(1);

  display.setCursor(0, 35);

  display.print("Humidity: ");

  display.setTextSize(2);

  display.setCursor(0, 45);

  display.print(Humidity);

  display.print(" %");

  display.display();

  delay(1000);

}
```

Ros Workspace

Nonfunctional Gazebo Interface

This launch file

- Loads the urdf from the macro tutorial into the parameter description (as before)
- Launches an empty gazebo world
- Runs the script to read the urdf from the parameter and spawn it in gazebo.
- By default, the gazebo gui will also be displayed, and look like this:



However, it doesn't do anything, and is missing lots of key information that ROS would need to use this robot. Previously we had been using joint_state_publisher to specify the pose of each joint. However, the robot itself should provide that information in the real world or in gazebo. Yet without specifying that, Gazebo doesn't know to publish that information.

To get the robot to be interactive (with you and ROS), we need to specify two things: Plugins and Transmissions.

Gazebo Plugin

To get ROS to interact with Gazebo, we have to dynamically link to the ROS library that will tell Gazebo what to do. Theoretically, this allows for other Robot Operating Systems to interact with Gazebo in a generic way. In practice, its just ROS.

To link Gazebo and ROS, we specify the plugin in the URDF, right before the closing </robot> tag:

Toggle line numbers

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/</robotNamespace>
  </plugin>
</gazebo>
```

However, this won't do anything new yet. For that we need to specify more information outside the URDF.

Spawning Controllers

Now that we've linked ROS and Gazebo, we need to specify some bits of ROS code that we want to run within Gazebo, which we generically call controllers. These are initially loaded into the ROS parameter space. We have a yaml file joints.yaml that specifies our first controller.

```
type: "joint_state_controller/JointStateController"
publish_rate: 50
```

This controller is found in the joint_state_controller package and publishes the state of the robot's joints into ROS directly from Gazebo.

In 09-joints.launch you can see how we should load this yaml file into the r2d2_joint_state_controller namespace. Then we call the [[controller_manager]]/spawner script with that namespace which loads it into Gazebo.

You can launch this, but its still not quite there. roslaunch urdf_sim_tutorial 09-joints.launch

This will run the controller and in fact publish on the /joint_states topic....but with nothing in them.

```
header:
  seq: 652
  stamp:
    secs: 13
    nsecs: 331000000
  frame_id: ''
name: []
position: []
```

```
velocity: []
effort: []
```

Transmissions

For every non-fixed joint, we need to specify a transmission, which tells Gazebo what to do with the joint. Let's start with the head joint. Add the following to your URDF:

Toggle line numbers

```
<transmission name="head_swivel_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <actuator name="$head_swivel_motor">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
  <joint name="head_swivel">
<hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
  </joint>
</transmission>
```

- For introductory purposes, just treat most of this chunk of code as boilerplate.
- The first thing to note is the joint element. The name should match the joint declared earlier.
- The hardwareInterface will be important as we explore the plugins.

You can run this URDF with our previous launch configuration. roslaunch urdf_sim_tutorial 09-joints.launch model:=urdf/10-firsttransmission.urdf.xacro

Now, the head is displayed properly in RViz because the head joint is listed in the joint_states messages.

```
header:
  seq: 220
  stamp:
    secs: 4
    nsecs: 707000000
  frame_id: ''
name: ['head_swivel']
position: [-2.9051283156888985e-08]
velocity: [7.575990694887896e-06]
effort: [0.0]
```

We could continue adding transmissions for all the non-fixed joints (and we will) so that all the joints are properly published. But, there's more to life than just looking at robots. We want to control them. So, let's get another controller in here.

Joint Control

Here's the next controller config we're adding.

```
type: "position_controllers/JointPositionController"
joint: head_swivel
```

This specifies to use the a JointPositionController from the position_controllers package to control the head_swivel transmission. Note that hardware interface in the URDF for this joint matches the controller type.

Now we can launch this with the added config as before roslaunch urdf_sim_tutorial 10-head.launch

Now Gazebo is subscribed to a new topic, and you can then control the position of the head by publishing a value in ROS. rostopic pub /r2d2_head_controller/command std_msgs/Float64 "data: -0.707"

When this command is published, the position will immediately change to the specified value. This is because we did not specify any limits for the joint in our urdf. However, if we change the joint, it will move gradually.

Toggle line numbers

```
<joint name="head_swivel" type="continuous">
  <parent link="base_link"/>
  <child link="head"/>
  <axis xyz="0 0 1"/>
  <origin xyz="0 0 ${bodylen/2}"/>
  <limit effort="30" velocity="1.0"/>
</joint>
```

Another Controller

We can change the URDF for the Gripper joints in a similar way. However, instead of individually controlling each joint of the gripper with its own ROS topic, we might want to group them together. For this, we just need to specify a different controller in the ROS parameters.

```
type: "position_controllers/JointGroupPositionController"
joints:
  - gripper_extension
  - left_gripper_joint
  - right_gripper_joint
```

To launch this, roslaunch urdf_sim_tutorial 12-gripper.launch

With this, we can instead specify the position with an array of floats. Open and out:

```
rostopic pub  /r2d2_gripper_controller/command std_msgs/Float64MultiArray
"layout:
  dim:
  - label: ''
    size: 3
    stride: 1
  data_offset: 0
data: [0, 0.5, 0.5]"
```

Closed and retracted:

```
rostopic pub  /r2d2_gripper_controller/command std_msgs/Float64MultiArray
"layout:
  dim:
  - label: ''
    size: 3
    stride: 1
  data_offset: 0
data: [-0.4, 0, 0]"
```

The Wheels on the Droid Go Round and Round

To drive the robot around, we specify yet another transmission for each of the wheels from within the wheel macro.

Toggle line numbers

```
  <transmission name="${prefix}_${suffix}_wheel_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <actuator name="${prefix}_${suffix}_wheel_motor">
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
      <joint name="${prefix}_${suffix}_wheel_joint">
        <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
      </joint>
    </transmission>
```

This is just like the other transmissions, except

- It uses macro parameters to specify names
- It uses a VelocityJointInterface.

Since the wheels are actually going to touch the ground and thus interact with it physically, we also specify some additional information about the material of the wheels.

Toggle line numbers

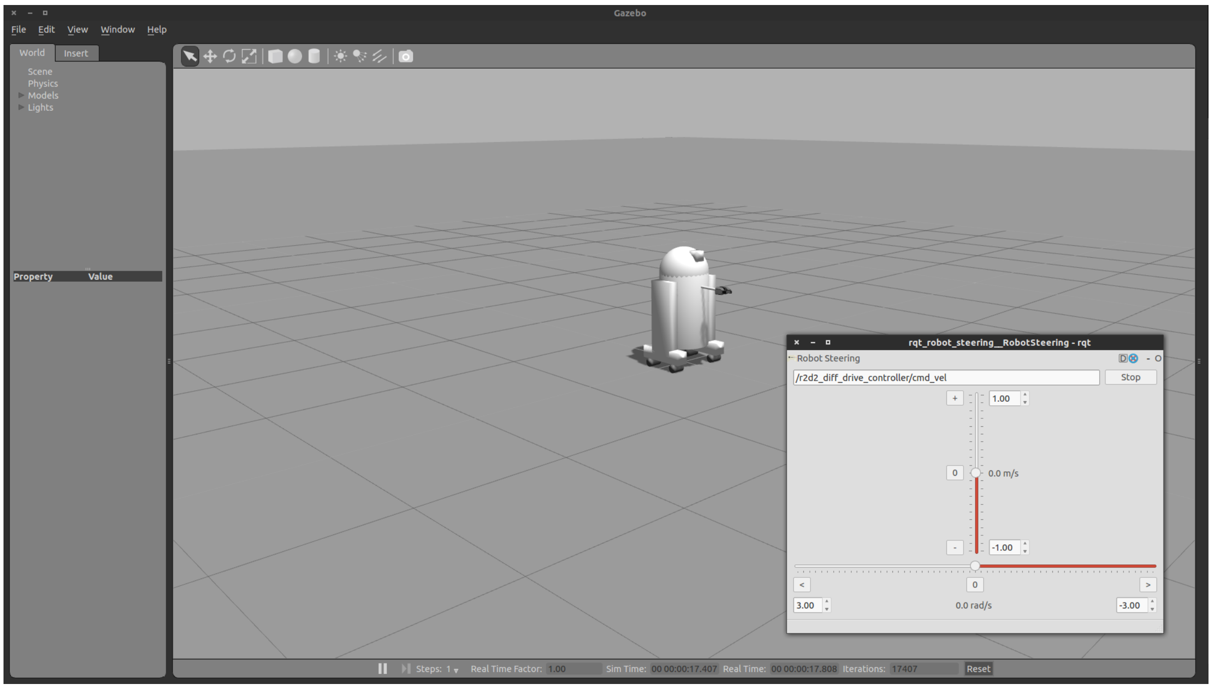
```
  <gazebo reference="${prefix}_${suffix}_wheel">
      <mu1 value="200.0"/>
      <mu2 value="100.0"/>
      <kp value="10000000.0" />
      <kd value="1.0" />
      <material>Gazebo/Grey</material>
    </gazebo>
```

```
 type: "diff_drive_controller/DiffDriveController"
  publish_rate: 50

  left_wheel: ['left_front_wheel_joint', 'left_back_wheel_joint']
  right_wheel: ['right_front_wheel_joint', 'right_back_wheel_joint']

  wheel_separation: 0.44

  # Odometry covariances for the encoder output of the robot. These values
should
  # be tuned to your robot's sample odometry data, but these values are a
good place
  # to start
  pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
  twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]

  # Top level frame (link) of the robot description
  base_frame_id: base_link

  # Velocity and acceleration limits for the robot
  linear:
    x:
      has_velocity_limits    : true
      max_velocity           : 0.2   # m/s
      has_acceleration_limits: true
      max_acceleration       : 0.6   # m/s^2
  angular:
    z:
      has_velocity_limits    : true
      max_velocity           : 2.0   # rad/s
      has_acceleration_limits: true
      max_acceleration       : 6.0   # rad/s^2
```

The DiffDriveController subscribes to a standard Twist cmd_vel message and moves the robot accordingly.

```
roslaunch urdf_sim_tutorial 13-diffdrive.launch
```

In addition to loading the above configuration, this also opens the RobotSteering panel, allowing you to drive the R2D2 robot around, while also observing its actual behavior (in Gazebo) and it's visualized behavior (in RViz):



/r2d2_diff_drive_controller/cmd_vel

1.00
0.0 m/s
-1.00
3.00
0.0 rad/s
-3.00

Real Time Factor: 1.00 Sim Time: 00 00:00:17.407 Real Time: 00 00:00:17.808 Iterations: 17407

```
for (n in 1:100) { if (n %% 3 == 0 & n %% 5 == 0)

{print("FizzBuzz")} else if (n %% 3 == 0)

{print("Fizz")} else if (n %% 5 == 0) {print("Buzz")}

else print(n) }
```