Design of a 3-DOF Robotic Arm and implementation of D-H Forward Kinematics

Denis Manolescu[1] and Emanuele Lindo Secco[1]

[1] Robotics Lab, School of Mathematics, Computer Science and Engineering, Liverpool Hope University, UK
20203547@hope.ac.uk, seccoe@hope.ac.uk

Abstract. Robotic prototypes are gradually gaining more academic and research traction, while robots are becoming a more common encounter within the human social trusted cycles. Generally, in the fields of Robotics - besides the mandatory high standards of safety - engineers are keen to build their concepts with an intuitive, natural-like motion and characteristics. This project will go through the process of building a novel 3-degrees of freedom robotic arm. The study will also aim at introducing and demonstrating the importance of a proper kinematics: an analysis on how geometry computations are used to improve the control and motion of a robot is performed. Finally, the pros and cons and the constraints of the proposed system are discussed together with a set of approaches and solutions, providing an overall approach towards the design and implementation of closed hardware and software robotics solution.

Keywords: Robotic Arm, Forward Kinematics, Denavit-Hartenberg.

1 Introduction

Robotic arms, suitably named because they generally resemble the human arm and mimic their functionality, were mainly introduced into the heavy industries in the 1960s. The automotive industry was the first to adopt the technology; shortly after, other assembly-line factories followed (Moran, 2007). Even from the early stages, having a machine that can continuously do repetitive tasks with speed and precision, lift loads greater than a person's capacity and endure harsh environmental conditions – Robots seemed to be the logical future. They became attractive by offering efficiency and cost-effective advantages through automation (Smith, 2018).

Throughout time, robotic arms were an evolutionary direction that built and guided the Robotics industry and determined its existence today. Thanks to greater sensory awareness and novel programming algorithms - like artificial intelligence, deep learning or machine learning - robotic arms can now interact and fulfil side-by-side tasks with humans.

Typically, from an engineering perspective, a robotic arm consists of a steady base that anchors a multi-segmented structure and forms the robot body. The connections between segments are called joints, and individually they determine an axis of motion,

also known as the **degree of freedom**. The last link of the arm is the **end-effector** which is a specialised device designed to interact with the environment. The end-effector can take many shapes and forms, like gripper, vacuum head, magnets, soft-manipulation grabber, wielding tip, drilling or cutting tool, painting spray or camera module (Ernest L Hall, 2015).

In essence, most joints are built using actuators that are capable of initiating movements depending on their control signal feedback. In tandem with the linear or rotational motion of an actuator, joints can mostly be classified as **prismatic** or sliding, **revolute** or rotary, **spherical** or socket and **helical** or screw-based (*Fig. 1*). A higher number of joints enables more freedom of movement, and the majority of the robotic arms available have four to six joints (Intel.inc, 2016; Chand 2021; Chand 2022).

Coordinating the entire movement of a robotic arm while determining its position, orientation and velocity is a fundamental task in robotics.

Kinematics is the main mathematical option capable of accurately describing the relationship between the joint coordinates, the end-effector and their spatial layout (Illinois, 2015).

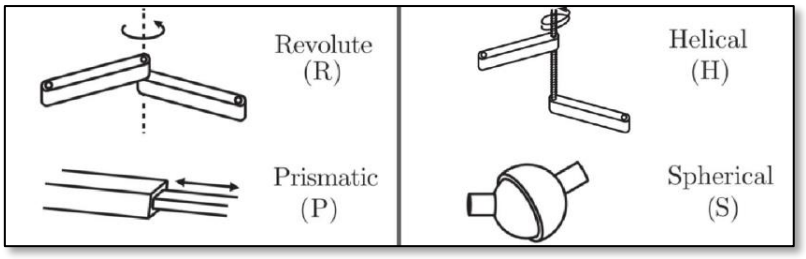


Figure 1 - Robotic joint types (image source: medium, 2022)

There are two ways to use kinematics equations in the context of a robotic arm:

1. **Forward kinematics** – uses the kinematics equations to calculate the position, orientation and velocity of the end-effector while knowing the joints angle.

In practice, the forward kinematics process uses the kinematic structure of the robot and the spatial configuration of the links to calculate the rotation and displacement of each frame. The result of forwarding kinematics is a single possible solution chain-linked to the entire structure of the robotic arm, no matter its movement.

2. **Inverse kinematics** – uses kinematics equations to compute a configuration of the joints position or angle that is necessary to place the end-effector in a given or desired location and orientation. Inverse kinematics is a more complex mathematical calculus than forward kinematics and usually produces multiple solutions (V. Kumar, 2016).

Once deployed into the robotic arm control algorithm, kinematics calculations can also be used to improve precision motion and prediction or objects collision detection.

This study will analyze the designing and building process of a 3-Degrees of Freedom robotic arm with revolute joints and will go through the process of calculating and applying Forward Kinematics.

The paper is organized as it follows: the Materials sand Methods section will talk about the characteristics of the robotic arm built in this project and the main components. The Forward Kinematics chapter will present the in-depth process of how kinematics work and how forward kinematics is calculated and integrated into the robot system. The Algorithm section will complete the Forward Kinematics chapter by going through the code developed to operate the robotic arm and the integration of forward kinematics. The Results section will point out the issues encountered, and the solutions adopted. It will also present the final view of the robotic arm and prove its functionality. The last section, namely the Conclusion – reports an objective overview of the entire project and reflect on the results achieved.

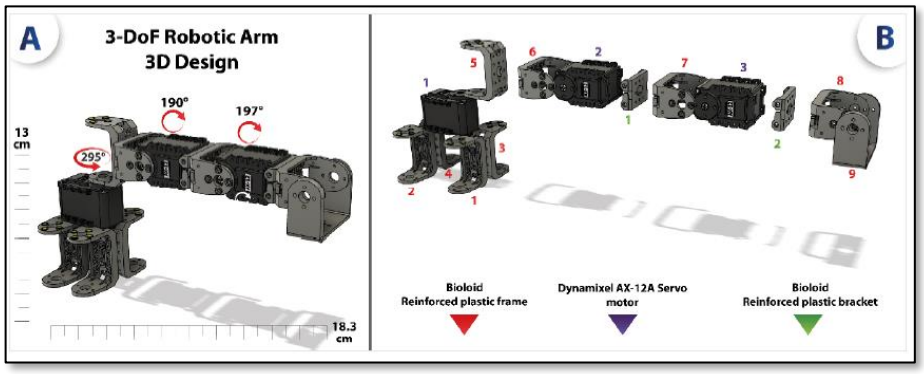


Figure 2 - Robotic Arm 3D Design made in Autodesk Fusion 360

2 Materials and Methods

The 3-degrees of freedom robotic arm built in this study has a vertical length of 26cm and a working envelope of 18.3cm radius (*Fig. 2A*).

The project uses three servo actuators connected by nine enforced plastic frames and three reinforced plastic brackets (*Fig. 2B*). The base is anchored with screws by a dense polyamide board, and the entire structure is kept in place by two gripping clamps.
The arm is controlled by an Arduino Mega 2560 Rev.3, which holds on top a servo driver shield connected to a 9.6V DC power supply (*Fig. 3B*).

The first join within the base rotates 295° left-right on the Z-axis, forming a displacement of 13cm from the ground to the next frame. The second joint rotates 190° up-down, constrained only by the position of the holding bracket. Between the first and second frame, there is a displacement of 11.75 cm. The third joint is identical to the previous one, while the distance between the second and third frame is 6.55cm. There is also an additional 5 cm distance from the servo rotation axis to the end-effector tip – i.e. the pencil (*Fig. 3, panel A and B*).

2.1 The Arduino Mega & Arduino Uno microcontrollers

The central computing power of the robot is generated by an **Arduino Mega** Rev3 board which is based on the 8-bit ATmega 2560 microcontroller produced by Microchip. The board operates on 5V and has a clock speed of 16Mhz while offering a convenient 54 digital connection pins and 16 analog pins (*Fig. 4C*).

Although the microcontroller is not used for complicated computations, it holds enough processing power to perform decently real-time kinematics calculations.

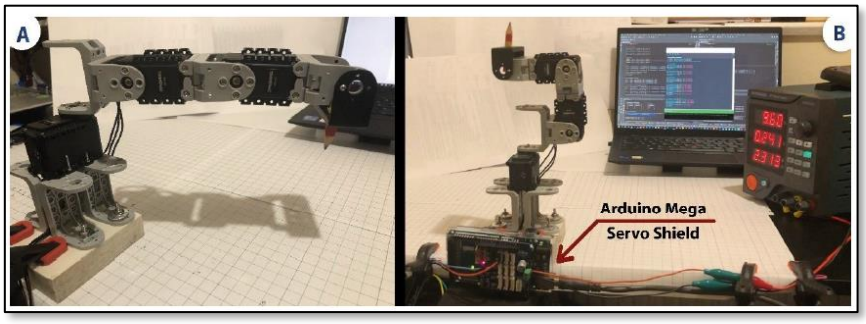
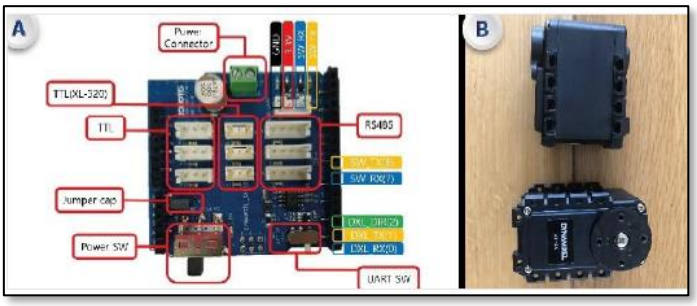


Figure 3 - Robotic Arm. All components overview (final version)

Additionally, in the early stage of development, an Arduino Uno had to be used in parallel with Mega, and its purpose was to capture the UART data pocket coming from the servo motor via the shield drive. More about this issue in the Challenges section below.



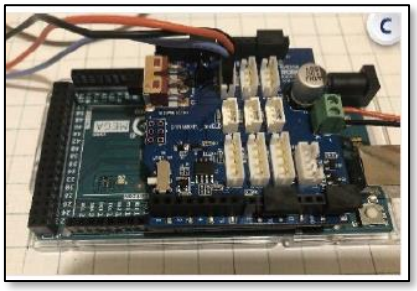


Figure 4 - Dynamixel servo driver shield layout, AX12A Dynamixel servo motor, Arduino Mega & Servo shield attached

2.2 Dynamixel Servo Shield

The servo shield used in the study is created by Dynamixel for their own servo motor series, and it is built to fit on top of Arduino Mega or Uno (*Fig. 4C*). The shield operates on 5-24V, and one of its best features is to allow the export of the data packets coming from the servo motors via its UART pins (Robotis, 2021).

An inconvenient flaw of the shield is the manual switch between the communication with the Arduino serial port and the servo motor control channel (Fig. 4A – UART SW). When uploading commands to the microcontroller, the UART needs to be set to Upload mode. For the servo motors to start reacting to the commands uploaded, the UART needs to be turned to Dynamixel mode. This manual switch becomes a real issue in the repetitive debugging stage of the servo motors and the robotic arm movements tests.

Software-wise, the producer and the community offer a wide range of support for testing and further development through C++ libraries - all fully compatible with the Arduino environment.

2.3 Dynamixel Servo Motors

The entire robotic arm design has been built around three Dynamixel AX12-A servo motors. These are advanced, high-performance robotics actuators designed to be modular and daisy chain connected (*Fig. 4B*).

Underneath the reinforced plastic housing, the actuator is composed of a 9-12V DC motor with 1.5N/m stall torque, a 254:1 spur gearbox, a build-in microcontroller with feedback, a driver and a half-duplex serial interface running at up to 1 Mbps through a UART/TTL serial link. This network feature enhances the servo capabilities and offers precise control and programmability of torque, speed, position, temperature and even voltage (Adafruit, 2020).

A very interesting feature of these intelligent servo motors is their capacity to be switched from a Joint mode – with a rotation capacity of 300°, to a Wheel mode – giving them the ability to endlessly rotate without constraints.

Another helpful feature is the daisy chain connection – meaning the servos only need to be connected to each other in a serial chain via a 3P cable with a Molex connector. In the daisy chain setup, only a single servo needs to be directly plugged into the Arduino shield. This setup is possible because of the asynchronous communication between devices and the use of unique IDs for each motor in the commands data package.

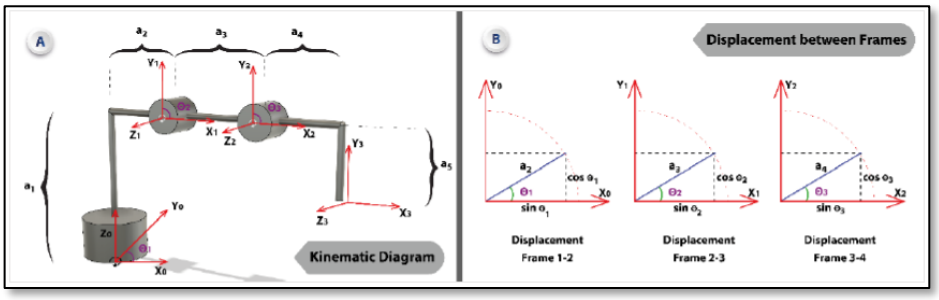


Figure 5 - Kinematics Diagram for Rotation Matrices & calculus. | Displacement layout between joint frames

3 Forward Kinematics Calculations

Forward kinematics is the mathematical process that uses the joint angles, in this case, $\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$, $\boldsymbol{\theta_3}$ (theta values), to determine the exact position of the end-effector (*Fig, 5A*).

The study will go through two different ways to apply forward kinematics, namely:

1. Conventional way - using the **rotation matrix** and the **displacement matrix** to calculate the **homogeneous transformation matrix**.
2. And the **Denavit-Hartenberg method**.

Both of these methods use the same rotation matrix calculus.

3.1 Rotation Matrices

Considering the robotic arm has three revolute joints, the rotation matrix formula for the entire system is:

$$R_3^0 = R_1^0 \cdot R_2^1 \cdot R_3^2$$

The first stage consists of calculating the projection matrices of each next frame onto the previous frame, with the joints in their initial state or position. This process will help determine the rotation of each joint in a frame space relationship.

1. Projection Matrix of Frame 1 to Frame 0:

	x_1	y_1	z_1
x_0	1	0	0
y_0	0	0	-1
z_0	0	1	0

2. The projection matrix of Frame 2 to Frame 1 and the projection matrix of Frame 3 to Frame 2 are the same and equal to the Identity Matrix. That is because the orientation of their axes is identical.

	x_2	y_2	z_2
x_1	1	0	0
y_1	0	1	0
z_1	0	0	1

(Frame 2 – Frame 1)

	x_3	y_3	z_3
x_2	1	0	0
y_2	0	1	0
z_2	0	0	1

(Frame 3 – Frame 2)

Once the projections are defined, the rotation matrices are calculated:

$$R_1^0 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{\cos\theta_1} & \mathbf{0} & \mathbf{\sin\theta_1} \\ \mathbf{\sin\theta_1} & \mathbf{0} & \mathbf{-\cos\theta_1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix}$$

Rotation of θ_1 which is a Z-axis rotation * Projection of Frame 1 to Frame 0

$$R_2^1 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{\cos\theta_2} & \mathbf{-\sin\theta_2} & \mathbf{0} \\ \mathbf{\sin\theta_2} & \mathbf{\cos\theta_2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Rotation of θ_2 which is a Z-axis rotation * Projection of Frame 2 to Frame 1 (identity matrix)

$$R_3^2 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{\cos\theta_3} & \mathbf{-\sin\theta_3} & \mathbf{0} \\ \mathbf{\sin\theta_3} & \mathbf{\cos\theta_3} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Rotation of θ_3 which is a Z-axis rotation * Projection of Frame 3 to Frame 2 (identity matrix)

Technically, there is no need to calculate the R_3^0 matrix to determine the homogeneous transformation matrices. But the Python algorithm developed will compute it anyway; more about that in the Code Section below.

3.2 Displacement Matrix

Displacement, in the context of the robotic arm, refers to the *x*, *y*, and *z* position of frame n in frame m. The general formula is:

$$d_n^m = \begin{bmatrix} x_n^m \\ y_n^m \\ z_n^m \end{bmatrix}$$

Using the *Fig. 5B* diagram, the displacement matrices are determined in the context of **θ_1, θ_2, θ_3,** as below:

$$d_1^0 = \begin{bmatrix} a_2 * \cos\theta_1 \\ a_2 * \sin\theta_1 \\ a_1 \end{bmatrix} \qquad d_2^1 = \begin{bmatrix} a_3 * \cos\theta_2 \\ a_3 * \sin\theta_2 \\ 0 \end{bmatrix} \qquad d_3^2 = \begin{bmatrix} a_4 * \cos\theta_3 \\ a_4 * \sin\theta_3 \\ 0 \end{bmatrix}$$

3.3 Homogenous Transformation Matrix

In Robotics, the *Homogeneous Transformation Matrix* (HTM) is a tool that combines the **rotation matrix** with the **displacement matrix** to generate the position and orientation of the end-effector. The HTM of the robot structure is calculated the same way as the rotation matrix by multiplying the HTM of all frames.

The HTM formula for the entire robotic arm is:

$$H_3^0 = H_1^0 \cdot H_2^1 \cdot H_3^2$$

where it holds:

$$H_n^m = H \begin{bmatrix} R_n^m & d_n^m \\ 0\ 0 & 0\ 1 \end{bmatrix}$$

, R – 3x3 rotation matrix
d – 3x1 displacement matrix

In this study, the final H_3^0 of the robotic system is computed by a Python algorithm. For testing purposes, the defined joint angles are $\theta_1 = 75°$, $\theta_2 = 147°$, $\theta_3 = 147°$, and the result is:

```
Homogeneous Transformation Matrices H_3_0 is:
 [[ 0.95630476  0.          0.2923717  17.93071417]
 [ 0.2923717   0.         -0.95630476  5.48196946]
 [ 0.          1.          0.          0.05      ]
 [ 0.          0.          0.          1.        ]]

Forward Kinematics - end results:
By using the Displacement & Rotation matrices we get end-effector (x, y, z) coordinates:
( 17.930714174306917 ,  5.481969463551313 ,  0.05000000000000071 )
```

= x
= y
= z

Where the end-effector tip, according the all calculations so far, is placed at the coordinate (17.93, 5.48, 0.05), in the 1st quadrant. Additional explanations of the result are discussed in the Results section below.

3.4 The Denavit-Hartenberg Method

The second method studied in applying forward kinematics to the robotic arm is the *Denavit-Hartenberg* (D-H) method. For this technique to work, four rules must be followed in the frame assignment stage (Angela Sodemann, 2020):

1. In the case of a revolute joint, the Z-axis is considered the rotation axis;
2. The X-axis must be perpendicular to the Z-axis of the previous frame;
3. The X-axis must intersect the Z-axis of the previous frame; this rule does not apply in the context of Frame 0;
4. The Y-axis must be determined from the X and Z axis by following the right-hand rule.

Once these rules are in place as shown in *Fig. 5A*, the next step is to determine the D-H **Parameters table** (Table 1).

Table 1. The D-H parameters of the designed Robotic Arm

Joint number	**θ theta angle**	**α angle** = the rotation around X-axis to get the previous Z-axis	**r** = distance along **X-axis** from the centre of the previous frame to the centre of the present frame	**d** = distance along **Z-axis** from the centre of the previous frame to the centre of the present frame
1	θ_1	90	a_2	$a_1 - a_5$
2	θ_2	0	a_3	0
3	θ_3	0	a_4	0
		Describes rotation		Describes displacement

The final step of the D-H method is to use the parameters above to calculate the homogeneous transformation matrices. In this case, the general formula is:

$$H_n^{n-1} = \begin{bmatrix} \cos\theta_n & -\sin\theta_n * \cos\alpha_n & \sin\theta_n * \sin\alpha_n & r_n * \cos\theta_n \\ \sin\theta_n & \cos\theta_n * \cos\alpha_n & -\cos\theta_n * \sin\alpha_n & r_n * \sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Each D-H homogeneous transformation matrix gets calculated in Python. The code is shown and explained below.

4 Algorithm – C++ and Python

4.1 The Arduino IDE & C++ libraries

The setup to operate the joint movement of the robotic arm is defined in the *Fig. 6* below: at first, the servo shield C++ library is included in the program, by calling the *DynamixelShield.h* header file provided by the developer. Line 3-11 in the code represents an automation process, where *if statement* makes the shield recognize the type of Arduino that will run the system.

```
1  #include <DynamixelShield.h>
2
3  #if defined(ARDUINO_AVR_UNO) || defined(ARDUINO_AVR_MEGA2560)
4    #include <SoftwareSerial.h>
5    SoftwareSerial soft_serial(7, 8); // DYNAMIXELShield UART RX/TX
6    #define DEBUG_SERIAL soft_serial
7  #elif defined(ARDUINO_SAM_DUE) || defined(ARDUINO_SAM_ZERO)
8    #define DEBUG_SERIAL SerialUSB
9  #else
10   #define DEBUG_SERIAL Serial
11 #endif
12
13 // robotic arm servo IDs
14 const uint8_t DXL_ID_1 = 1;
15 const uint8_t DXL_ID_2 = 2;
16 const uint8_t DXL_ID_3 = 3;
17
18 const float DXL_PROTOCOL_VERSION = 1.0; // declare servo firmware
19
20 DynamixelShield dxl;  // instance of the main library class
21
22 // This namespace is required to use Control table variables
23 using namespace ControlTableItem;
24
25 void setup() {
26   // set servos communication baud rate & debug baud rate
27   DEBUG_SERIAL.begin(115200);
28   dxl.begin(1000000);
29
30   // Set Port Protocol Version.
31   // This has to match with DYNAMIXEL protocol version.
32   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
33   // Get DYNAMIXEL information
34 //  dxl.ping(DXL_ID_1);
35 //  dxl.scan();
```

Figure 6 - Arduino sketch - Setups

The servo motors ID variables are declared in lines 14-16, while the firmware used to run the servo motors is defined on line 18. This *PROTOCOL_VERSION* is a default variable specific for the series of actuators used. Line 20 represents the instance of the library class *DynamixelShield* that gives access to all the motor functionalities, as they are made available by the producer. Line 23 also creates general access to the global variables used in the control of the servos, like ID, torque, temperature, position and many more. From Line 25 in the code, it begins the setup of the connection between Arduino and servo motors via the driving shield. The default baud rate to transmit the commands to the servos is 1 Mbps, set by command *dxl.begin(1000000)*.

```
38 void loop() {
39   // set speed:
40   dxl.setGoalVelocity(1, 70);
41   dxl.setGoalVelocity(2, 70);
42   dxl.setGoalVelocity(3, 70);
43
44   // check positions:
45   int check1 = dxl.getPresentPosition(DXL_ID_1);
46   int check2 = dxl.getPresentPosition(DXL_ID_2);
47   int check3 = dxl.getPresentPosition(DXL_ID_3);
48   int step = 0; // may use this later for precision motion check
49
50     // XYZ middle
51   dxl.setGoalPosition(DXL_ID_1, 147, UNIT_DEGREE);
52   dxl.setGoalPosition(DXL_ID_2, 147, UNIT_DEGREE);
53   dxl.setGoalPosition(DXL_ID_3, 147, UNIT_DEGREE);
54   delay(5000);
```

Figure 7 - Arduino sketch to run the robotic arm. Main loop code.

In the main loop, in the code lines 40-42, the speed of each of the three servo motors is set to a value of 70 (*Fig. 7*). The function used, *setGoalVelocity()*, can take raw data, rpm or percentage values; in this case, it is set to a raw value, and it can go up to 450-500.

The initial reading of the servo position is done using the *getPresentPosition()*, which can also take the raw or degrees values as a variable. At the same time, the data received is valuable feedback that can be used to drive the motor in the desired position with high accuracy.

The *setGoalPosition*() function rotates the specific actuator at the requested angle. This function is very powerful and useful in the kinematics and the motion of the robotic arm.

4.2 PyCharm 2022 and Python

The robotic arm project used Python programming language and PyCharm IDE to build two different algorithms:

1. One to execute the conventional forward kinematics calculus with Rotation matrices, Displacement matrices and the Homogeneous Transformation matrices (Fig. 9).
2. The other one applies the D-H method with the Rotation matrices, Parameters table and DH Homogeneous Transformation matrices (Fig. 8).

```
# Homogeneous Transformation Matrices
i = 0   # row set of parameters used to generate HTM from DH table
H_0_1 = [[cos(theta_1), -sin(theta_1) * cos(90), sin(theta_1) * sin(90), PT[i][2] * cos(theta_1)],
         [sin(theta_1), cos(theta_1) * cos(90), -cos(theta_1) * sin(90), PT[i][2] * sin(theta_1)],
         [0, sin(90), cos(90), PT[i][3]], [0, 0, 0, 1]]

i = 1    # row set of parameters used to generate HTM from DH table
H_1_2 = [[cos(theta_2), -sin(theta_2) * cos(90), sin(theta_2) * sin(90), PT[i][2] * cos(theta_2)],
         [sin(theta_2), cos(theta_2) * cos(90), -cos(theta_2) * sin(90), PT[i][2] * sin(theta_2)],
         [0, sin(90), cos(90), PT[i][3]], [0, 0, 0, 1]]

i = 2    # row set of parameters used to generate HTM from DH table
H_2_3 = [[cos(theta_3), -sin(theta_3) * cos(90), sin(theta_3) * sin(90), PT[i][2] * cos(theta_3)],
         [sin(theta_3), cos(theta_3) * cos(90), -cos(theta_3) * sin(90), PT[i][2] * sin(theta_3)],
         [0, sin(90), cos(90), PT[i][3]], [0, 0, 0, 1]]

# Final Homogeneous Matrix
H_0_3 = dot(dot(H_0_1, H_1_2), H_2_3,)
print("\nHomogeneous Transformation Matrices:\n", matrix(H_0_3))
print("\nForward Kinematics - Denavit-Hartenberg Method\nThe end-effector (x, y, z) coordinates are:")
print("(", H_0_3[0][3], ", ", H_0_3[1][3], ", ", H_0_3[2][3], ")")
```

Figure 8 - Python algorithm. D-H method in Forward Kinematics calculus

The results are compared and discussed in the Results section below. Both algorithms are running on the same theta values representing the joint angles, $\theta_1 = 75°$, $\theta_2 = 147°$, and $\theta_3 = 147°$.

```
# Displacement Matrices
D_0_1 = [[a2 * cos(theta_1)], [a2 * sin(theta_1)], [a1]]
D_1_2 = [[a3 * cos(theta_2)], [a3 * sin(theta_2)], [0]]
D_2_3 = [[a4 * cos(theta_3)], [a4 * sin(theta_3)], [0]]

# Homogeneous Transformation Matrices
H_0_1 = [[cos(theta_1), 0, sin(theta_1), a2 * cos(theta_1)], [sin(theta_1), 0, -cos(theta_1), a2 * sin(theta_1)],
         [0, 1, 0, a1], [0, 0, 0, 1]]
H_1_2 = [[cos(theta_2), -sin(theta_2), 0, a3 * cos(theta_2)], [sin(theta_2), cos(theta_2), 0, a3 * sin(theta_2)],
         [0, 0, 1, 0], [0, 0, 0, 1]]
H_2_3 = [[cos(theta_3), -sin(theta_3), 0, a4 * cos(theta_3)], [sin(theta_3), cos(theta_3), 0, (a4 * sin(theta_3)-a5)],
         [0, 0, 1, 0], [0, 0, 0, 1]]

# Final Homogeneous Matrix
H_0_3 = dot(dot(H_0_1, H_1_2), H_2_3,)
print("Homogeneous Transformation Matrices H_3_0 is:\n", H_0_3)

print("\nForward Kinematics - Conventional calculus\nThe end-effector (x, y, z) coordinates are:")
print("(", H_0_3[0][3], ", ", H_0_3[1][3], ", ", H_0_3[2][3], ")")
```

Figure 9 - Python algorithm for conventional Forward Kinematics calculus

Both, Arduino sketch and Python forward kinematics code are made available online at github.com/deemano.

Essential aspects to consider in the forward kinematics calculus are the transformation from degree into radians and the positioning of the joint axis, which starts from the initial position of the actuator (i.e. 0°). The 3D axis of the joints can be shifted conveniently to fit any desired layout without influencing the results.

5 Results

The only challenge this research faced was making the UART debugging process work. Typically, capturing all the servo motors feedback requires a dedicated device sold by the producer, which acts as a medium and can capture the half-duplex serial data stream and decode it. But in this case, using a second Arduino proved to be a viable solution in capturing the data stream. Solving this issue was important in changing the default ID assigned to each servo motor or other personalized parameters like protocols, which can be unknown.

In the final stage, the movement of the robotic arm is smooth and steady, without any issues. This outstanding performance of the servo actuators allowed the study to focus the entire efforts on implementing forward kinematics.

Although the conventional way to calculate forward kinematics seemed more straightforward and modulated, both methods were generating identical results in determining the tip of the robotic arm end-effector**.**

*In the end, and consistently with the literature, the D-H method proved to be a simpler way to emulate the robotic arm structure and motion, and faster to calculate***.**

6 Discussion & Conclusion

This work has been focused on creating a complete view of the process of building an operational robotic arm. Additionally, the research elaborated a comprehension of how forward kinematics algorithms can be achieved and integrated into the robot system. The 3D design of the robot was a laborious procedure with significant creative efforts but of great contribution to understanding the kinematics relationships within the robot structure. The electronics part of the project has proven to be a relatively simple stage, primarily because of the intuitive and accessible documentation and other resources related to the hardware used.

Forward Kinematics, on the other hand, has been the most challenging part, demanding significant amounts of research and raw pen-and-paper calculations. In its essence, when building your own robotic arm, kinematics is a personal aspect of the structural motion of the robot that can only work if it is understood correctly.

The industrial development of high-performance smart actuators is becoming accessible and more affordable. Once algorithm concepts like Deep-learning and AI get to be integrated into these devices by default, the Robotics field will be able to fully evolve and expand into all the other industries and aspects of human life [12-13].

ACKNOWLEDGMENTS

This work was presented in coursework form in fulfilment of the requirements for the BEng in Robotics Engineering for the student Denis Manolescu from the Robotics Laboratory, School of Mathematics, Computer Science and Engineering, Liverpool Hope University

References

1. Adafruit, 2020. *DYNAMIXEL Motor - AX-12A.* [Online] Available at: https://www.adafruit.com/product/4768 [Accessed 2022].
2. Angela Sodemann, A. A., 2020. *How to Assign Denavit-Hartenberg Frames to Robotic Arms.* [Online] Available at: https://automaticaddison.com/how-to-assign-denavit-hartenberg-frames-to-robotic-arms/ [Accessed 2022].
3. Association, R. I., 2015. *The First Industrial Robot.* [Online] Available at: https://www.automate.org/a3-content/joseph-engelberger-unimate [Accessed 2022].
4. Chand R, Chand RP, Kumar SA. 2022. Switch controllers of an n-link revolute manipulator with a prismatic end-effector for landmark nav. *J Computer Science* 8:e885
5. Chand RP, Kumar SA et al, Lyapunov-based Controllers of an n-link Prismatic Robot Arm, *2021 IEEE Asia-Pacific Conf on CS & Data Eng* (CSDE), 2021, 1-5.
6. Ernest L. Hall, U. o. C., 2015. *Introduction to Robotics - End Effectors.* [Online] Available at: https://www.researchgate.net/publication/282976515_Robotics_1_Lecture_7_End_Effectors [Accessed 2022].
7. Illinois, U. o., 2015. *Robot Kinematics.* [Online] Available at: http://motion.cs.illinois.edu/RoboticSystems/Kinematics.html [Accessed 2022].
8. Inc, I., 2021. *Moore's Law and Intel Innovation.* [Online] Available at: https://www.intel.co.uk/content/www/uk/en/history/museum-gordon-moore-law.html [Accessed 2022].
9. Intel.inc, 2016. *Industrial Robotic Arms: Changing How Work Gets Done.* [Online] Available at: https://www.intel.com/content/www/us/en/robotics/robotic-arm.html [Accessed 2022].
10. Medium, 2022. Available at: www.medium.com [Accessed 2022].
11. Moran, M. E., 2007. *Evolution of robotic arms.* [Online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4247431/#:~:text=Unimate%20introduced%20the%20first%20industrial,a%20robotic%20arm%20in%201993. [Accessed 2022].
12. Robotics, 2021. *DYNAMIXEL Shield.* [Online] Available at: https://emanual.robotis.com/docs/en/parts/interface/dynamixel_shield [Accessed 2022].
13. Smith, E., 2018. *Going Through The Motions.* [Online] Available at: https://tedium.co/2018/04/19/robotic-arm-history-unimate-versatran/ [Accessed 2022].
14. V. Kumar, P. E. U. o. P. S. o. E. a. A. S., 2016. *Introduction to Robot Geometry and Kinematics.* [Online] Available at: https://www.seas.upenn.edu/~meam520/notes02/IntroRobotKinematics5.pdf [Accessed 2022].
15. VD Manolescu, EL Secco, Design of an Assistive Low-Cost 6 d.o.f. Robotic Arm with Gripper, 7th International Congress on Information and Communication Technology (ICICT 2022), Lecture Notes in Networks and Systems (ISSN: 2367-3370)
16. S Procter, EL Secco, Design of a Biomimetic BLDC Driven Robotic Arm for Teleoperation & Biomedical Applications, Journal of Human, Earth, and Future (ISSN: 2785-2997), 2022