Table of contents

Chapter 1 Kinematics of 2 DOF Arm Robot

1-1 Finding the hand position from the joint angles

First we consider the (forward) kinematics of a two-degree-of-freedom arm robot. Kinematics is the task of finding the position and posture of the hand (End Effector) given the state of the robot's drive joint (rotary / prismatic). Kinematics does not take dynamics into consideration, but simply considers the relationship between the robot joint coordinate system and the working coordinate system of the hand. This document deals only with robots that consist of rotating joints.

Figure 1-1 shows a schematic diagram of a 2-axis (2 degrees of freedom) arm robot. A robot with link (arm) lengths l_1 and l_2 has reached the position P (x, y) with joint angles q_1 and q_2. There are two postures in which the second joint is in a mountain bend and a valley bend.

Kinematics is the calculation of where the hand position P (x, y) go when the joint angles are set to q_1 and q_2. For 2 degrees of freedom case, using trigonometric functions, it's

$$x = l_1 * cos(q_1) + l_2 * cos\,(q_1 + q_2)$$
$$y = l_1 * sin(q_1) + l_2 * sin\,(q_1 + q_2) \qquad (1.1)$$

The joint angles q_1 and q_2 should be given in radians, and counterclockwise is the positive direction. The hand position can be obtained by the equation (1.1) for both the mountain bend and the valley bend.

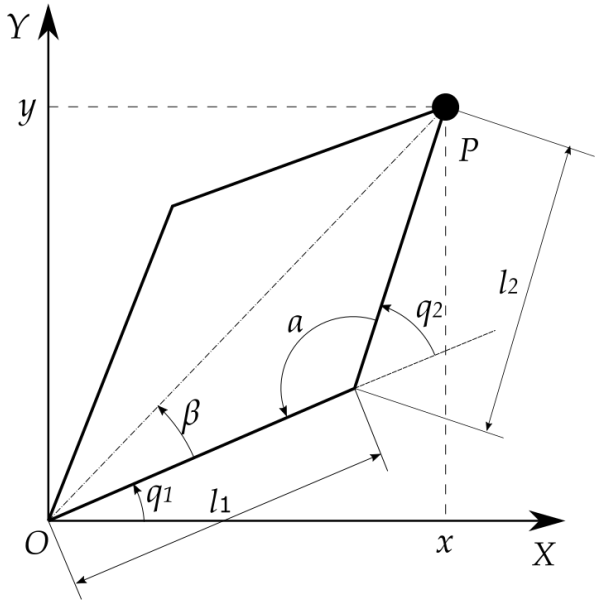

Figure 1-1

1-2 Making Unity Models

Create a 3D project with New button in Unity Hub and open the initial screen. For those who are not familiar with Unity, to roughly explain the screen configuration with the default layout, it's like, Hierarchy on the upper left is the parts list, Project on the lower is the parts box including not used, Inspector on the right is the detailed specifications of each part, and Scene in the center is the work desk.

Check how to use the mouse in the Scene view. Use the middle button to move, the right button to rotate, the wheel to zoom in and out, and the left button for normal selection. You can also set the projection direction using the scene gizmo in the upper right. Click Persp below it to change the projection method.

It should be noted that in Unity, the Y axis is pointing upward and the Z axis is the depth direction pointing into the page. As for the direction of rotation, counterclockwise (left-handed screw) facing the direction of travel of the axis is positive rotation. In the explanation of a general robot, I think that the coordinate system with the Z axis pointing upward is used. In this book, we will use a general coordinate system in the explanation of the principle, and convert it appropriately at the stage of applying it to Unity.

The Hierarchy view initially has a Main Camera and a Directional Light. The screen when it comes to a game or application will be seen through this Main Camera. You can see the camera image by selecting the Game tab next to the Scene tab. After selecting the second Move Tool from the buttons lined up in the upper left of the screen you can move the camera using the gizmo (arrow) that appears when you click the camera icon in the Scene view. At that time, in the Inspector view, make sure that the value of the part labeled Transform changes.

We will not touch Directional Light this time.

Now, let's make a Unity model of a two-degree-of-freedom arm robot. Please note that it is not a model of an actual product robot, but a toy model for studying.

We will make it from the parts at the base of the arm robot. Right-click in the blank area of the Hierarchy view and select 3D Object | Cylinder.

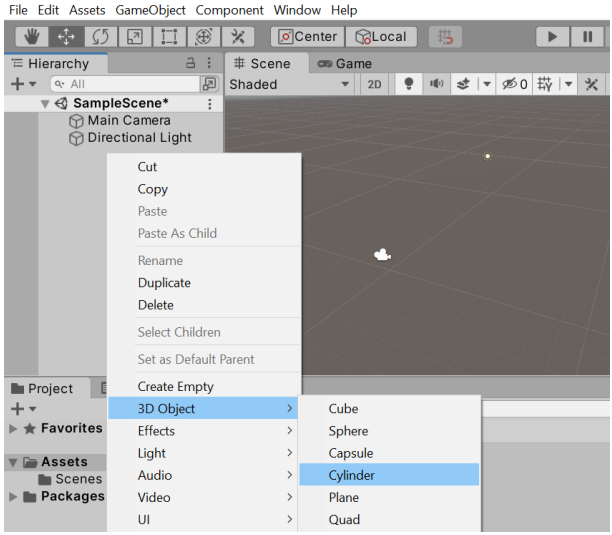

Figure 1-2

A scale 1 cylinder has been created. This will be the axis of rotation of the first link. Currently, it is installed vertically at a random position, so move it to the 3D origin and lay it on its side. You can set in the Inspector view with the cylinder selected in the Hierarchy or Scene view. Rewrite X, Y, Z of Position to 0 and X of Rotation to -90 (negative 90). Reset from the 3-dot button can also set numbers to zeros. Rotation means that you have rotated 90 degrees clockwise around its own X-axis.

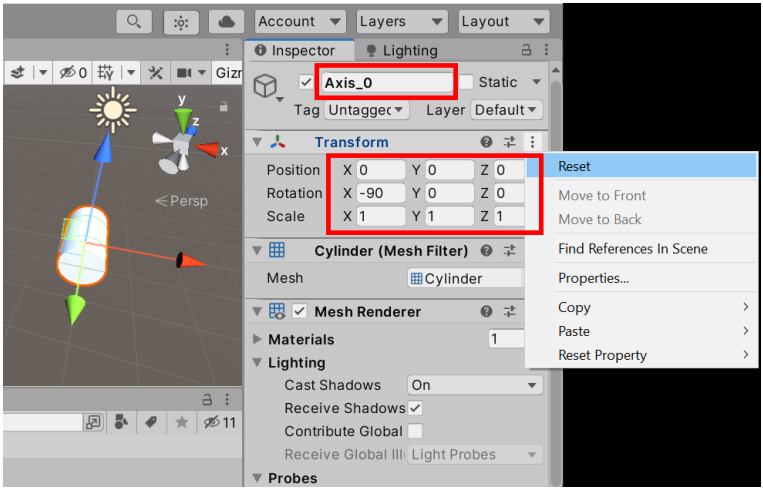

Figure 1-3

Regarding Scale, as you can see from the appearance, the height of the Cylinder seems longer even though X, Y, and Z are set to 1. In Unity the default Cylinder of scale 1 has diameter 1 and height 2.

Next, we make the arm part. Create 3D Object |Cube in the Hierarchy view as in the case of Cylinder. Set the name and Transform as shown in Figure 1-4. The left edge of the Cube should be at the center of the Cylinder.

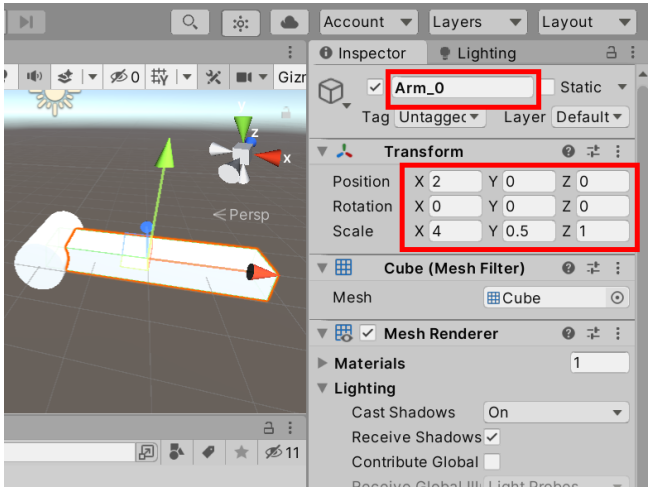

Figure 1-4

The unit of dimensions in Unity is meters by the way. Please admit that the robot is incredibly huge.

Now, we want to rotate this arm around the axis, but at present, when you change the value of Rotation(Inspector view | Transform | Rotation) of Arm_0, it rotates around the center of the part. It is not possible to change the center of rotation (pivot) of a part (GameObject is the name in Unity), so we need to devise a solution. This time, we will introduce an empty object whose pivot is at the center of rotation of the whole arm(It's the world origin in this case.), and have Axis_0 and Arm_0 as its child elements. Right-click in the Hierarchy view, create the object named Joint_0 with Create Empty, and drag and drop Axis_0 and Arm_0 there to make them child elements of Joint_0. Reset the Joint_0 Transform to zero. Now, if you enter a value in the Z-axis Rotation of Joint_0, it will rotate around the world origin(0, 0, 0).

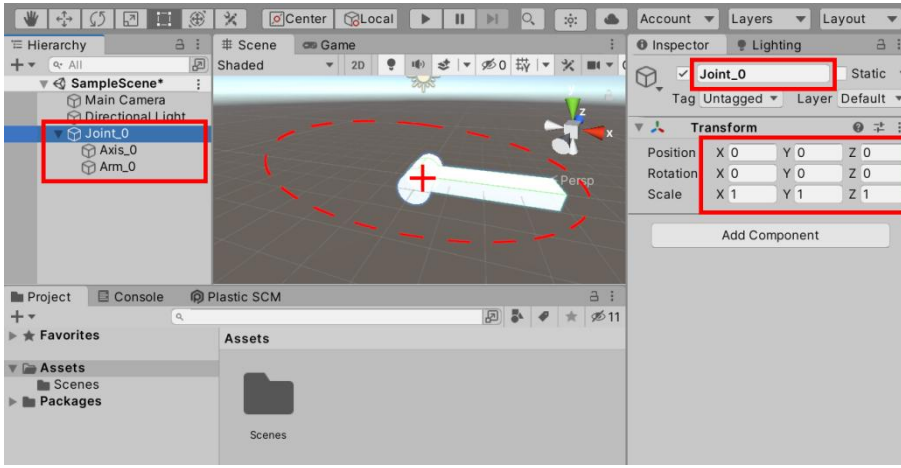

Figure 1-5

The first link is complete, but we'll color it to make it look good (to distinguish the links). To color an object in Unity, you have to create a Material with color information and add it to the object as a specification.

We will create two Materials for colors, so create a folder with the name Materials by selecting Create | Folder in Assets in the Project view. In that folder, select Create |Material to create two materials, Blue and Green. With each material selected, click the white box in the Inspector view and set the color in the palette that appears.

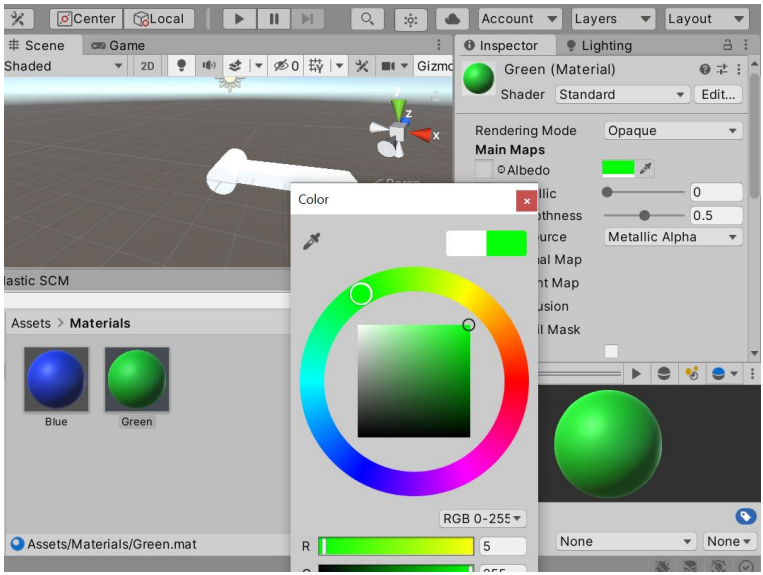

Figure 1-6

To apply a material to an object, with the object selected in the Hierarchy view, drag and drop the created material to near the button labeled Add Component at the bottom of the Inspector view. Or drag and drop the material directly into the Scene view.

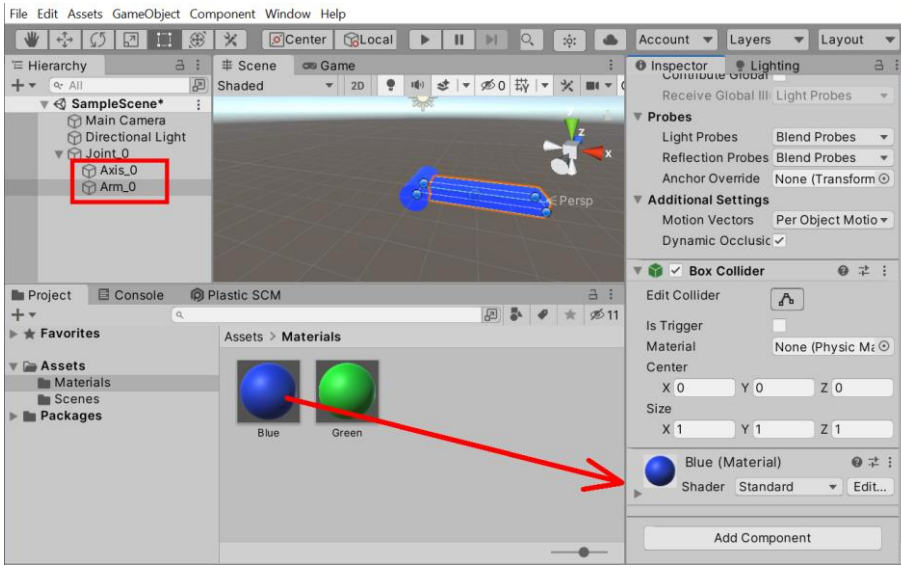

Figure 1-7

Next, we will make a second link. The second link should move around with the rotation of the first link, and need to rotate around the tip of the first link. To satisfy this, the structure is the same as the 1st link, but the pivot position of the empty parent object should be at the right end of the 1st arm, and the second link as a whole should be a child of the 1st link Joint_0.

First, right-click on Joint_0 in the Hierarchy view and select Create Empty to create Joint_1 as a child of Joint_0. The value of Transform is (4, 0, 0) to make it stay at the right end of Arm_0. The setting values of Axis_1 and Arm_1 are the same as Axis_0 and Arm_0, respectively. Set the green material for the second link.

	Pos X	Pos Y	PosZ	Rot X	Rot Y	Rot Z

Joint_1	4	0	0	0	0	0
Axis_1	0	0	0	-90	0	0
Arm_1	2	0	0	0	0	0

	Scale X	Scale Y	Scale Z
Joint_1	1	1	1
Axis_1	1	1	1
Arm_1	4	0.5	1

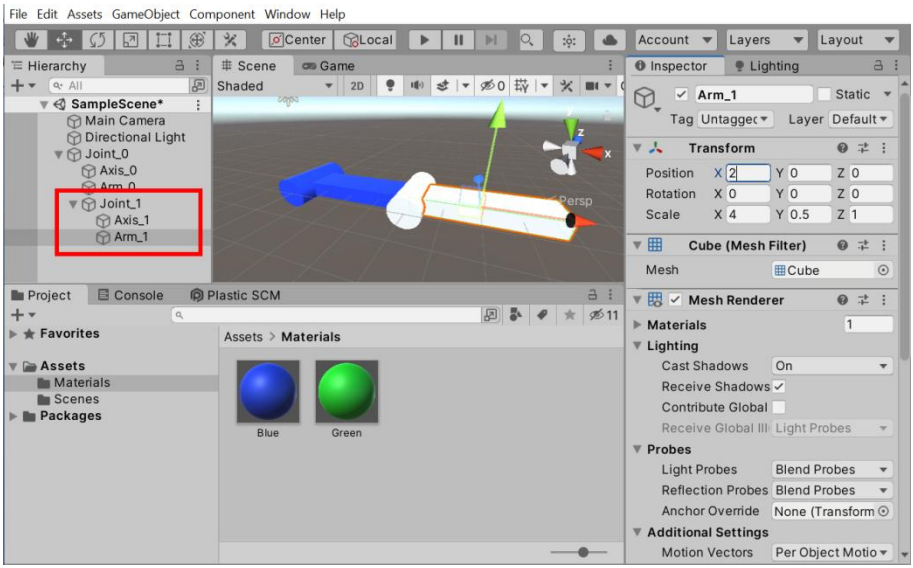

Figure 1-8

1-3 Coding Forward Kinematics

As the arm model with 2 degrees of freedom is completed, we will create an application that displays the coordinates of the tip (hand) by setting

the angle of each axis using the slider. The UI displayed on the screen is created on a 2D plane called Canvas separately from the 3D model. Right-click in the Hierarchy view and select UI | Canvas to generate a Canvas. At the same time, an Event System will be created. With the 2D button turned on in the Scene view, double-click Canvas in the Hierarchy view to display the entire Canvas. Change UI Scale Mode to Scale With Screen Size in Canvas Scaler in the Inspector view.

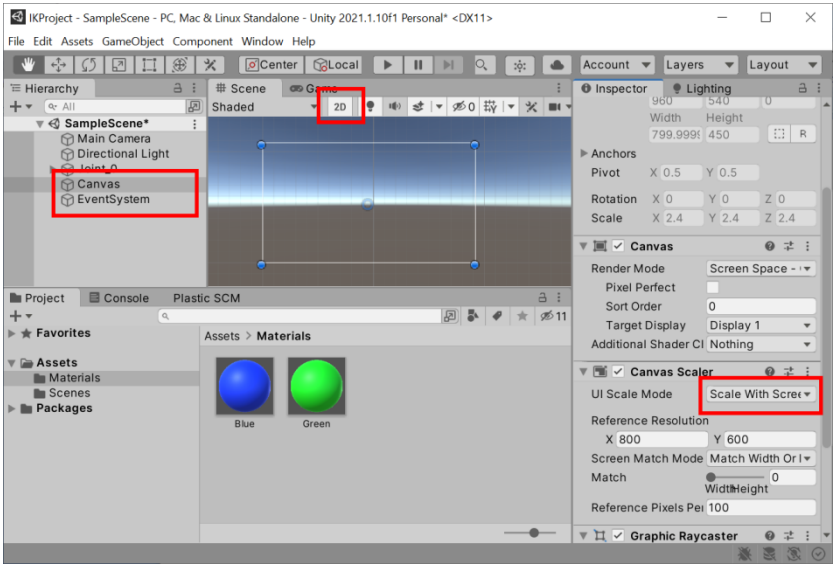

Figure 1-9

We create Sliders for setting angles and some Texts in Canvas. Refer to Figure 1-10 and set UI | Slider and UI | Text as child elements of Canvas. The dimensions are as you like, but the names of the Sliders and the Texts that display the numbers should be Slider_0, Slider_1, Angle_0, Angle_1, Pos_X, Pos_Y. This is because they will be referenced later from the script (program). Also, set the Sliders "Min Value" and "Max Value" to -90 and 90.

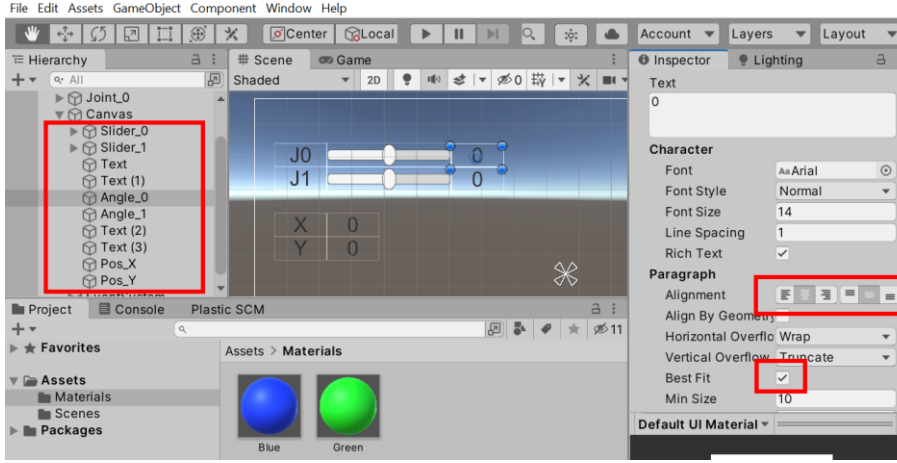

Figure 1-10

With Text selected, set Alignment to Center Align and Middle Align and check Best Fit. This will make Texts look nicer.

Next, we will finally write a script for the main subject of kinematics. Create a folder named Scripts in Assets in the Project view, and create a script named JointController in it by selecting Create | C # Script. When the application (game) starts, the script runs and issues commands to the objects in the Scene to move.

Double-clicking Joint Controller will launch Visual Studio. The bare bone code will be displayed, so let's add the original program here. Here is the code for the first half.

List 1-1

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.UI;

namespace InverseKinematics
{
    public class JointController : MonoBehaviour
    {
        // robot
        private GameObject[] joint = new GameObject[2];
        private GameObject[] arm = new GameObject[2];
        private float[] armL = new float[2];
        private Vector3[] angle = new Vector3[2] ;

        // UI
        private GameObject[] slider = new GameObject[2];
        private float[] sliderVal = new float[2];
        private GameObject[] angText = new GameObject[2];
        private GameObject[] posText = new GameObject[2];
```

```
using UnityEngine.UI;
```

This line is needed when you use UIs in the app.

```
namespace InverseKinematics
```

Enclose the entire class with namespace. In fact, it doesn't matter if you don't have this.

The first part of the class declares variables that represent GameObjects in the Scene view. **armL** is the length of the arm. **angle** can be float, but since Unity handles the rotation angle as a set of X, Y, and Z as a type called Vector 3, it may be reasonable to use it.

List 1-2

```
void Start()
{
  // robot
    for (int i = 0; i < joint.Length; i++)
    {
        joint[i] = GameObject.Find("Joint_" + i.ToString());
        arm[i] = GameObject.Find("Arm_" + i.ToString());
        armL[i] = arm[i].transform.localScale.x;
    }

    // UI settings
    for (int i = 0; i < joint.Length; i++)
    {
        slider[i] = GameObject.Find("Slider_" + i.ToString());
        angText[i] = GameObject.Find("Angle_" + i.ToString());
        sliderVal[i] = Slider[i].GetComponent<Slider>().value;
    }
    posText[0] = GameObject.Find("Pos_X");
    posText[1] = GameObject.Find("Pos_Y");
}
```

Start () function is executed only once at the beginning. In here, robot parts and UI parts are associated with variables declared in the script. At the same time, the arm length (Scale in the X direction) is assigned to the variable **armL[ ]**.

List 1-3

```
void Update()
{
```

```
        for (int i = 0; i < joint.Length; i++)
        {
            sliderVal[i] = slider[i].GetComponent<Slider>().value;
            angText[i].GetComponent<Text>().text
                = SliderVal[i].ToString("f2");
            angle[i].z = sliderVal[i];
            joint[i].transform.localEulerAngles = angle[i];
        }
        float px = armL[0] * Mathf.Cos(angle[0].z * Mathf.Deg2Rad) +
armL[1] * Mathf.Cos((angle[0].z + angle[1].z) * Mathf.Deg2Rad);
        float py = armL[0] * Mathf.Sin(angle[0].z * Mathf.Deg2Rad) +
armL[1] * Mathf.Sin((angle[0].z + angle[1].z) * Mathf.Deg2Rad);

        posText[0].GetComponent<Text>().text = px.ToString("f2");
        posText[1].GetComponent<Text>().text = py.ToString("f2");
    }
```

Update() function is executed every time the screen is redrawn. First, the values of the sliders are read and Joints are rotated to those angles. To set the angle of **joint[]**, change the Z-axis component of **angle[]** and then set it to **transform.localEulerAngles** of **joint[]**.

The next 4 lines (actually 2 lines, no line breaks) are the part of the kinematics calculation of equation (1.1). Angles should be converted from degrees to radians.

After saving the script, with Joint_0 selected in the Hierarchy view, drag and drop the JointController.cs script to the Inspector view. It doesn't have to be Joint_0 to attach the script, it can be another object, or you can add a new empty object.

Press the play button at the top to run the app. Move the slider and check that the robot moves and the hand position is displayed with a reasonable value.

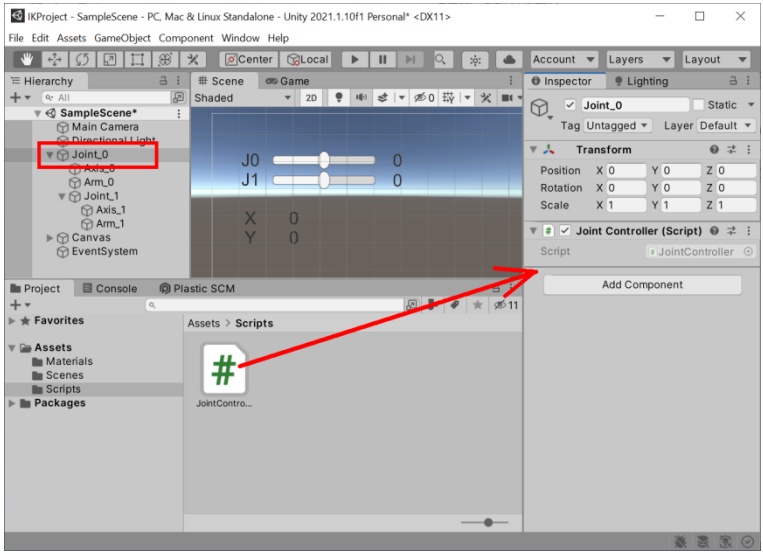

Figure 1-11

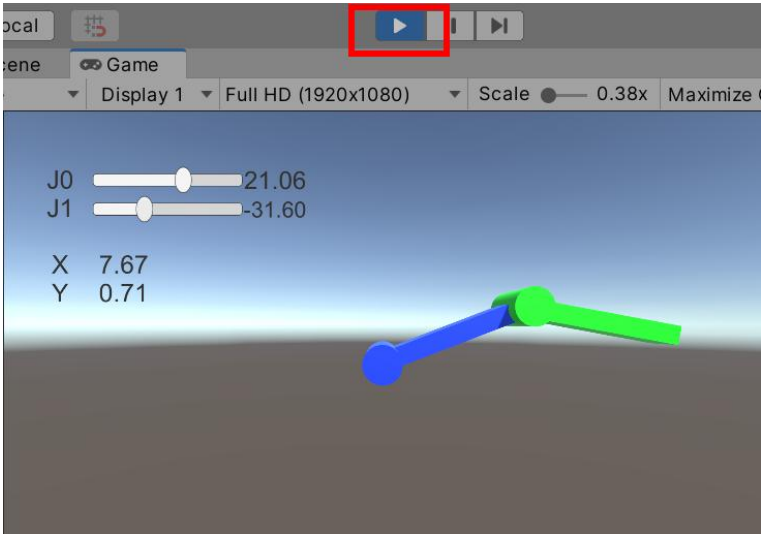

Figure 1-12

Save the project with a name such as IKProject1.

By the way, you can get the hand position without calculating with a script, by attaching an object to the tip of Joint_1 as shown in Fig. 1-13 and write a line like below.

```
Vector3 endPosition = Sphere.transform.position;
```

You can display the value in the Console view at the bottom of the screen during execution using **Debug.Log( )**.

```
Debug.Log(endPosition);
```

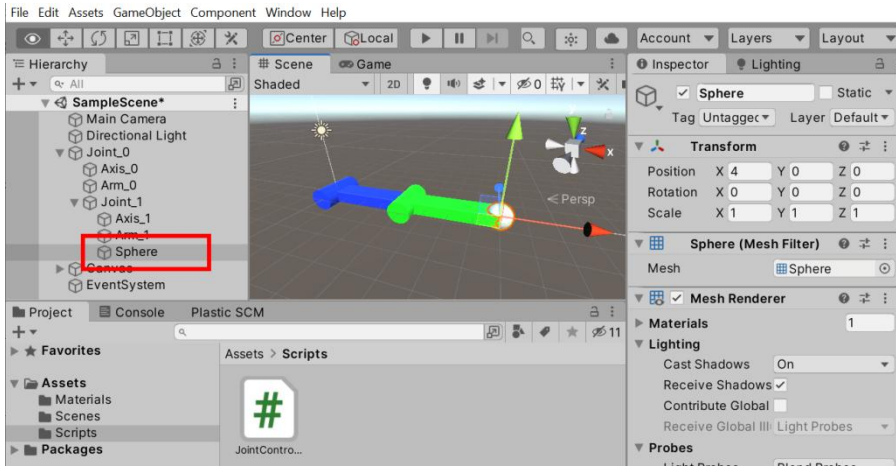
File Edit Assets GameObject Component Window Help
Center
Local
Account
Layers
Layout
Hierarchy
Scene
Game
Inspector
Lighting
Shaded
2D
SampleScene*
Main Camera
Directional Light
Joint_0
Axis_0
Arm_0
Joint_1
Axis_1
Sphere
EventSystem
Persp
Sphere
Static
Tag
Layer
Default
Transform
Position
Rotation
Scale
Sphere (Mesh Filter)
Mesh
Mesh Renderer
Materials
Lighting
Cast Shadows
On
Receive Shadows
Contribute Global
Probes
Project
Console
Plastic SCM
Favorites
Assets
Materials
Scenes
Scripts
Packages
Assets > Scripts
JointContro...

Figure 1-13

Chapter 2 Inverse Kinematics of 2 DOF Arm Robot

2-1 Getting Joint Angles from Hand Position

Inverse kinematics is to find the joint angles of the arm robot when the coordinates of the hand are known. Figure 1-1 is reposted. For two degrees of freedom, it is relatively simple and we use the trigonometric cosine theorem.

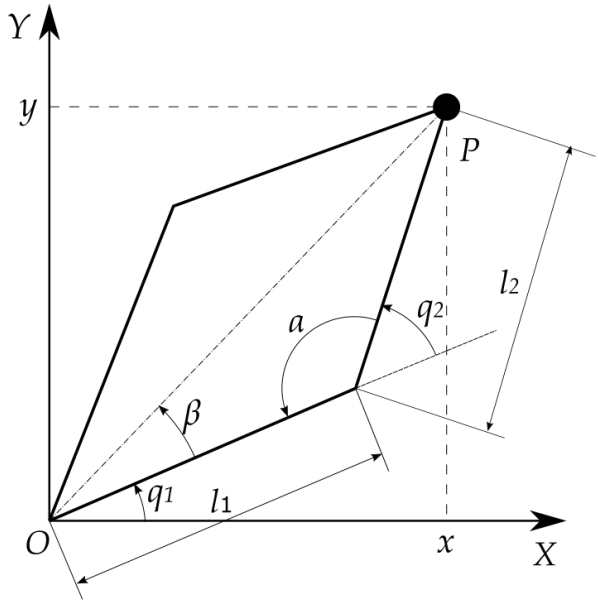

Figure 2-1

Equating the trigonometric cosine theorem

$$x^2 + y^2 = l_1^2 + l_2^2 - 2l_1 l_2 cos\alpha$$
$$l_2^2 = l_1^2 + x^2 + y^2 - 2l_1\sqrt{x^2 + y^2}cos\beta$$

(1.2)

Angles α and ß are given by

$$\alpha = cos^{-1}\left(\frac{l_1^2 + l_2^2 - x^2 - y^2}{2l_1 l_2}\right)$$

$$\beta = cos^{-1}\left(\frac{x^2 + y^2 + l_1^2 - l_2^2}{2l_1\sqrt{x^2 + y^2}}\right)$$

For mountain bend q_1 and q_2 are given by

$$q_1 = \tan^{-1}\left(\frac{y}{x}\right) + ß$$

$$q_2 = -\pi + \alpha$$

(1.3)

For valley bend q_1 and q_2 are given by

$$q_1 = \tan^{-1}\left(\frac{y}{x}\right) - ß$$

$$q_2 = \pi - \alpha$$

(1.4)

2-2 Coding Inverse Kinematics

Duplicate the project folder of previous chapter and rename it to IKProject2. Add the new folder in Unity Hub and open the project.

This time, we will reuse the robot model, but will change the UI for inverse kinematics.

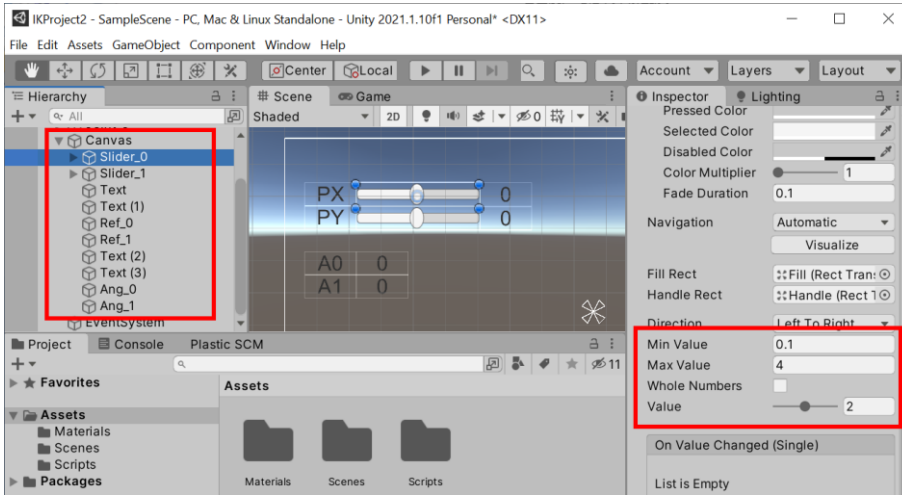

Figure 2-2

First, change the Texts displayed on the left side to PX, PY, A0, A1. Rename the Texts that displays the numbers to Ref_0, Ref_1, Ang_0, Ang_1 from the top.

Also, set both slider Min Value to 0.1, Max Value to 4, and Value to 2. These settings are the measure to prevent an error caused by specifying a position where the arm cannot reach.

Now let's modify the script. There is no change in the variable declaration part. Inside the **Start()** function we have to fix where the name has changed.

List 2-1

```
void Start()
{
    // robot
    for (int i = 0; i < joint.Length; i++)
```

```
    {
        joint[i] = GameObject.Find("Joint_" + i.ToString());
        arm[i] = GameObject.Find("Arm_" + i.ToString());
        armL[i] = arm[i].transform.localScale.x;
    }

    // UI settings
    for (int i = 0; i < joint.Length; i++)
    {
        slider[i] = GameObject.Find("Slider_" + i.ToString());
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
        posText[i] = GameObject.Find("Ref_" + i.ToString());
        angText[i] = GameObject.Find("Ang_" + i.ToString());
    }
}
```

In the **Update()** function the first for loop reads the slider value, and the next line is the part of the inverse kinematics calculation. Substituting **sliderVal[]** for x and y is simply to avoid making the expression too long. The joint angles in the mountain bend posture are calculated using equation (1.3).

List2-2

```
void Update()
{
    for (int i = 0; i < joint.Length; i++)
    {
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
    }
    float x = sliderVal[0];
    float y = sliderVal[1];
```

```
        float a = Mathf.Acos((armL[0] * armL[0] + armL[1] * armL[1] - x * x - y * y) / (2f * armL[0] * armL[1]));
        float b = Mathf.Acos((armL[0] * armL[0] + x * x + y * y - armL[1]* armL[1]) / (2f * armL[0] * Mathf.Pow((x * x + y * y), 0.5f)));
        angle[1].z = -Mathf.PI + a;
        angle[0].z = Mathf.Atan2(y, x) + b;

        for (int i = 0; i < joint.Length; i++)
        {
            joint[i].transform.localEulerAngles = angle[i] * Mathf.Rad2Deg;
            PosText[i].GetComponent<Text>().text = SliderVal[i].ToString("f2");
        }
        angText[0].GetComponent<Text>().text
            = (angle[0].z * Mathf.Rad2Deg).ToString("f2");
        angText[1].GetComponent<Text>().text = (angle[1].z * Mathf.Rad2Deg).ToString("f2");
    }
```

Do not break the lines of **float a**, **float b**, **angText[0]**,and **angText[1]**.

Save the script, run it, and make sure that the hand move to the correct position.

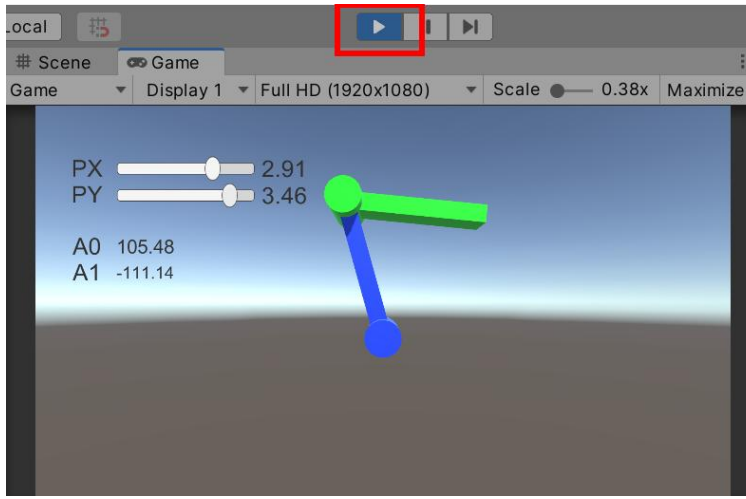
Local
Scene
Game
Game
Display 1
Full HD (1920x1080)
Scale
0.38x
Maximize O
PX
2.91
PY
3.46
A0
105.48
A1
-111.14

Figure 2-3

Chapter 3 Inverse Kinematics of 3 DOF Arm Robot

3-1 Getting Joint Angles from Hand Position

A robot arm with 3 degrees of freedom has a rotation axis around the vertical axis as the first axis compared to a robot arm with 2 degrees of freedom. You might afraid that it gets complicated suddenly when it comes to 3D, but it's not really that much. As you can see in Figure 3-1 the only difference is the x and y in the 2D case becomes a and b shown below.

$$a = \sqrt{x^2 + y^2}$$
$$b = z - l_1 \tag{3.1}$$

The rotation angle q_1 around the Z axis is determined only by the first axis.

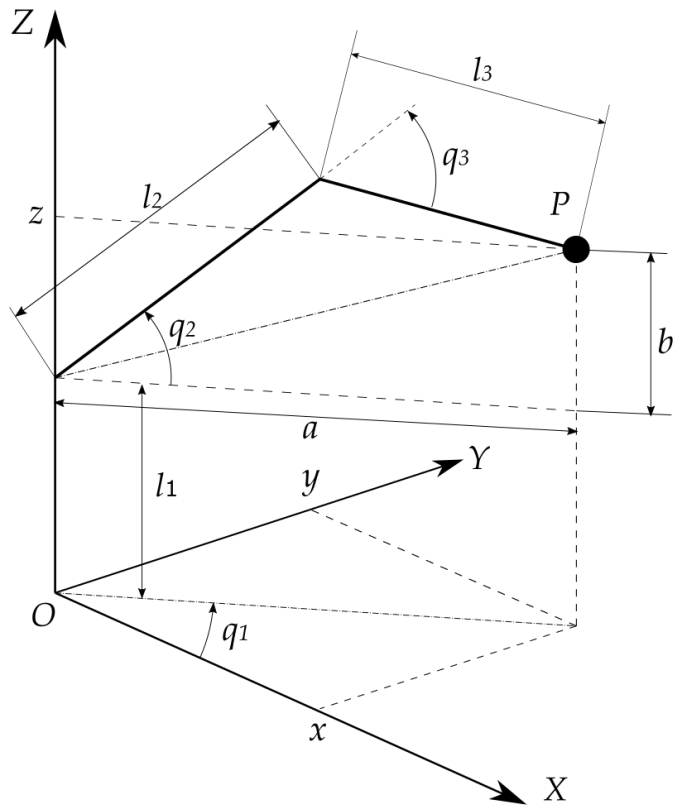

Figure 3-1

3-2 Making Unity 3DOF Model

Now let's get back into Unity and create a 3 degree of freedom model. Duplicate the "IKProject2" folder to make it "IKProject3" and so on.

First, in order to insert a new first axis Joint_0, rename the existing Joint_0 and Joint_1 to Joint_1 and Joint_2 including the objects inside.

Joint_1 → Joint_2　　Axis_1 → Axis_2　　Arm_1 → Arm_2

Joint_0 → Joint_1　　Axis_0 → Axis_1　　Arm_0 → Arm_1

Then evacuate the Joint_1 assembly upward using the green arrow.

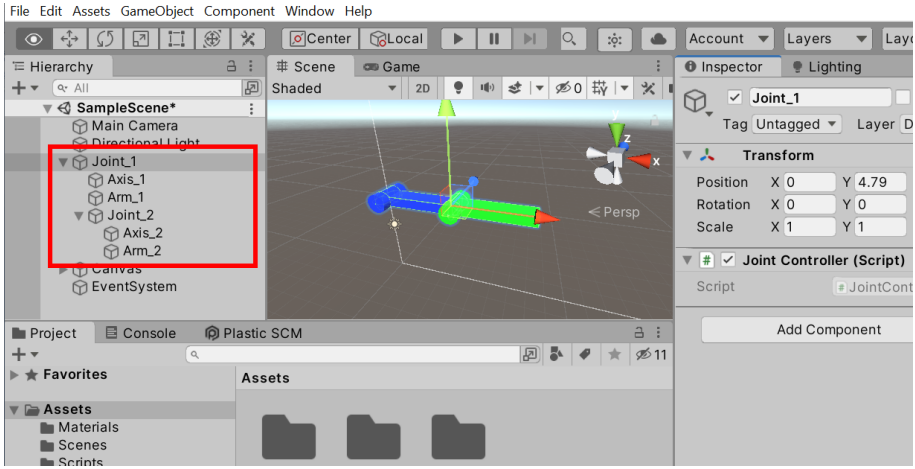

Figure 3-2

Joint_0 rotates around a vertical axis (Y axis in Unity), so create it as shown in Figure 3-3. Please refer to the values of Transform of each part shown below.

	PosX	PosY	PosZ	RotX	RotY	RotZ
Joint_0	0	0	0	0	0	0
Axis_0	0	0.5	0	0	0	0
Arm_0	0	1	0	0	0	0
Joint_1	0	2	0	0	0	0

	ScaleX	ScaleY	ScaleZ

Joint_0	1	1	1
Axis_0	1.5	0.5	1.5
Arm_0	0.5	2	1
Joint_1	1	1	1

The height offset of the 2nd axis (l_1 in Figure 3-1) is 2. In the Hierarchy view, make the Joint_1 assembly a child of Joint_0. Enter values in the Y-axis Rotation of Joint_0 and check that the whole body rotates horizontally.

Also Plane object (3D Object | Plane) is added for displaying the ground. Note that Plane is 10 meters on a side by default.

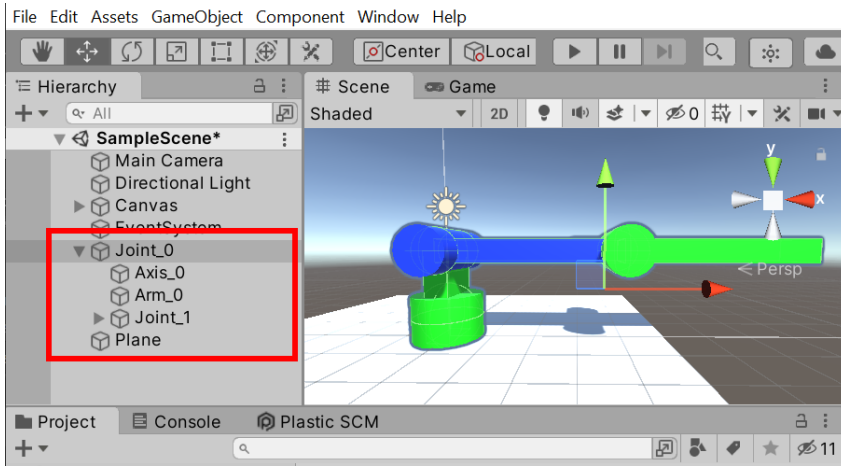

Figure 3-3

Next, add a Slider and Texts for in Canvas and modify the name accordingly. Set the PZ slider range to -4 to 4.

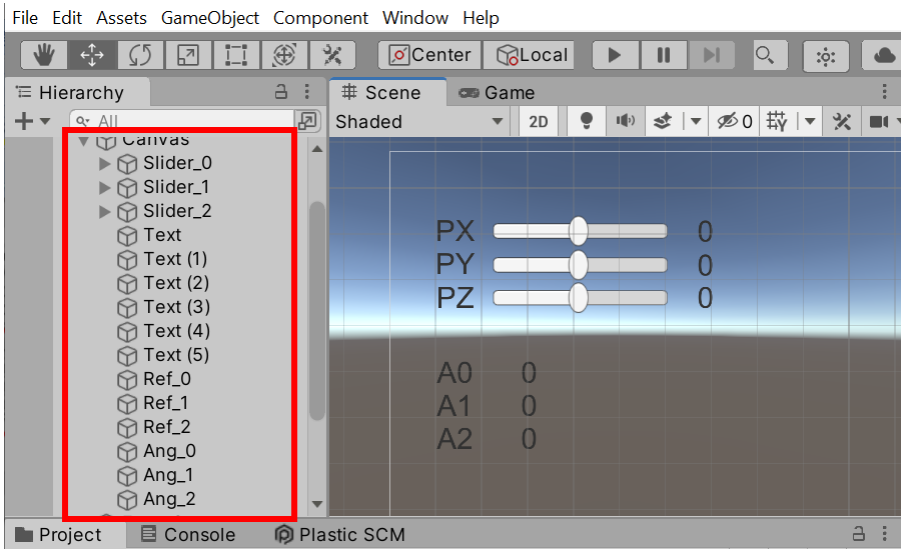

Figure 3-4

Now let's modify the script. First, change the number of array variables from 2 to 3 in the variable declaration part.

List 3-1

```
public class JointController : MonoBehaviour
{
    // robot
    private GameObject[] joint = new GameObject[3];
    private GameObject[] arm = new GameObject[3];
    private float[] armL = new float[3];
    private Vector3[] angle = new Vector3[3];

    // UI
    private GameObject[] slider = new GameObject[3];
    private float[] sliderVal = new float[3];
```

```
    private GameObject[] angText = new GameObject[3];
    private GameObject[] posText = new GameObject[3];
```

In the part that gets the length of the arm in the **Start()** function, since the first axis is in the Y-axis direction, we have to divide into cases using if else. Everything else is the same as before.

List 3-2

```
  void Start()
  {
    // robot
    for (int i = 0; i < joint.Length; i++)
    {
      joint[i] = GameObject.Find("Joint_" + i.ToString());
      arm[i] = GameObject.Find("Arm_" + i.ToString());
      if(i == 0) armL[i] = arm[i].transform.localScale.y;
      else armL[i] = arm[i].transform.localScale.x;
    }
    // UI settings
    for (int i = 0; i < joint.Length; i++)
    {     /*skipped*/          }
}
```

In the **Update()** function Z-axis components are added and the inverse kinematics calculation is performed applying the equation (3.1) .

リスト 3-3

```
  void Update()
  {
    for (int i = 0; i < joint.Length; i++)
```

```
    {
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
    }
    float x = sliderVal[0];
    float y = sliderVal[1];
    float z = sliderVal[2];
    angle[0].y = -Mathf.Atan2(z, x);
    float a = x / Mathf.Cos(angle[0].y);
    float b = y - armL[0];
    float alfa = Mathf.Acos((armL[1] * armL[1] + armL[2] * armL[2]
            - a * a - b * b) / (2f * armL[1] * armL[2]));
    angle[2].z = -Mathf.PI + alfa;
    float beta = Mathf.Acos((armL[1] * armL[1] + a * a + b * b - armL[2]
      * armL[2]) / (2f * armL[1] * Mathf.Pow((a * a + b * b), 0.5f)));
    angle[1].z = Mathf.Atan2(b, a) + beta;

    for (int i = 0; i < joint.Length; i++)
    {
        joint[i].transform.localEulerAngles = angle[i] * Mathf.Rad2Deg;
        posText[i].GetComponent<Text>().text
              = sliderVal[i].ToString("f2");
    }
    angText[0].GetComponent<Text>().text = (angle[0].y *
    Mathf.Rad2Deg).ToString("f2");
    angText[1].GetComponent<Text>().text = (angle[1].z *
    Mathf.Rad2Deg).ToString("f2");
    angText[2].GetComponent<Text>().text = (angle[1].z *
        Mathf.Rad2Deg).ToString("f2");
}
```

Let's save the script and run it.

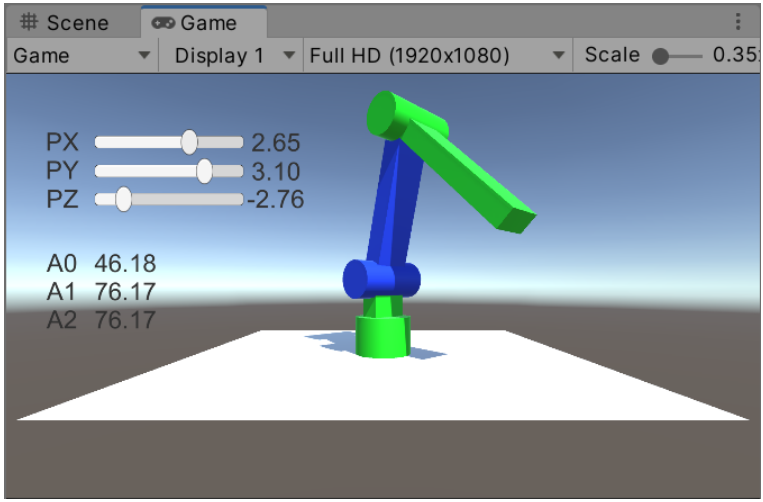

Figure 3-5

Adding error avoidance process

Now we add one feature. Until now we set the slider range small to prevent the program from collapsing caused by fully extended arm but this can be improved by avoiding the update the angles when the arm can't reach the distance.

The hand (tip of the arm) will not reach the target when length of the arms is shorter than the distance.

$$\sqrt{a^2 + b^2} > l_2 + l_3 \tag{3.2}$$

In this case, we should return the slider to the original position (previous **Update()**) and stop redrawing the model. First, in the variable declaration, describe the variable **prevSliderVal** for storing the previous

slider value.

List 3-4

```
// UI
private GameObject[] slider = new GameObject[3];
private float[] sliderVal = new float[3];
private float[] prevSliderVal = new float[3];
private GameObject[] angText = new GameObject[3];
private GameObject[] posText = new GameObject[3];
```

There is no change in **Start()**. After calculating the variables **a** and **b** in **Update()**, checking the equation (3.2), and if the posture becomes impossible, we return the slider to the original value and exit **Update()**.Otherwise, we continue the inverse kinematics calculation as before and substitute the new **sliderVal[]** for the **prevSliderVal[]** for the next calculation.

List 3-5

```
void Update()
{
    for (int i = 0; i < joint.Length; i++)
    {
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
    }
    float x = sliderVal[0];
    float y = sliderVal[1];
    float z = sliderVal[2];

    angle[0].y = -Mathf.Atan2(z, x);
    float a = x / Mathf.Cos(angle[0].y);
    float b = y - armL[0];
```

```
        if (Mathf.Pow(a * a + b * b, 0.5f) > (armL[1] + armL[2]))
        {
            for (int i = 0; i < joint.Length; i++)
            {
                sliderVal[i] = prevSliderVal[i];
                slider[i].GetComponent<Slider>().value = sliderVal[i];
            }
            return;
        }
        else
        {
            float alfa = Mathf.Acos((armL[1] * armL[1] + armL[2] * armL[2] - a
* a - b * b) / (2f * armL[1] * armL[2]));
            angle[2].z = -Mathf.PI + alfa;
            float beta = Mathf.Acos((armL[1] * armL[1] + a * a + b * b - armL[2]
* armL[2]) / (2f * armL[1] * Mathf.Pow((a * a + b * b), 0.5f)));
            angle[1].z = Mathf.Atan2(b, a) + beta;

            for (int i = 0; i < joint.Length; i++)
            {
                joint[i].transform.localEulerAngles = angle[i] * Mathf.Rad2Deg;
                posText[i].GetComponent<Text>().text                    =
sliderVal[i].ToString("f2");
                prevSliderVal[i] = sliderVal[i];
            }
            angText[0].GetComponent<Text>().text      =      (angle[0].y      *
Mathf.Rad2Deg).ToString("f2");
            angText[1].GetComponent<Text>().text      =      (angle[1].z      *
Mathf.Rad2Deg).ToString("f2");
            angText[2].GetComponent<Text>().text      =      (angle[1].z      *
```

```
Mathf.Rad2Deg).ToString("f2");
        }
    }
```

Save the script and run the app to make sure the slider doesn't update anymore when the arm is fully extended.

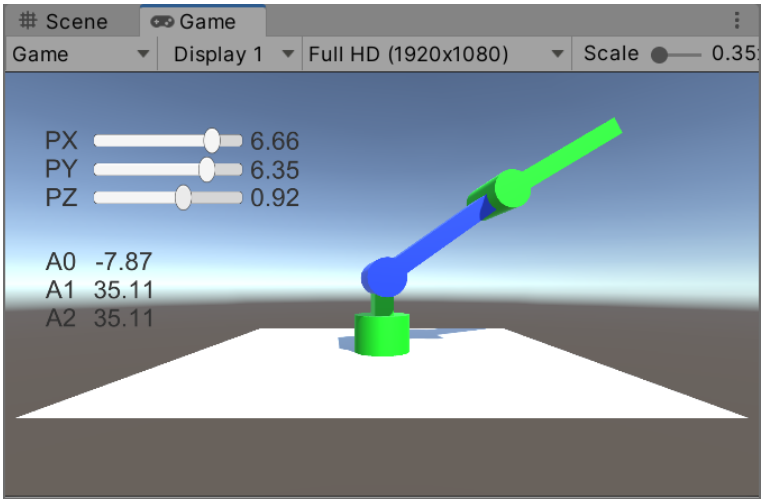

Figure 3-6

Chapter 4 Inverse Kinematics of 6 DOF Arm Robot

4-1 Extending the Model to 6 DOF

In this chapter, we will solve the inverse kinematics problem of the 6DOF arm robot analytically as before. When the number of joints increases to 6, you can specify not only the position of the hand but also the posture (orientation). You can easily expect the trigonometric formulas to become explosively complicated as the degrees of freedom increase. Therefore, what we are dealing with here is a robot of the type where the latter three axes intersect at one point (spherical wrist) to reduce the complexity of the formula. Many actual robots have this structure.

Getting back to Unity and duplicate the project from the previous chapter and start it as IK Project4. Add the three axes as shown in Figure 4-1. Shorten Arm_2 length. The parameters are shown for reference.

	PosX	PosY	PosZ	RotX	RotY	RotZ
Axis_2	0.5	0	0	0	0	0
Joint_3	1	0	0	0	0	0
Axis_3	0.5	0	0	0	0	-90
Arm_3	1	0	0	0	0	0
Joint_4	2	0	0	0	0	0
Axis_4	0	0	0	-90	0	0
Arm_4	0.5	0	0	0	0	0
Joint_5	1	0	0	0	0	0

Axis_5	0.5	0	0	0	0	-90
Arm_5	1	0	0	0	0	0

	ScaleX	ScaleY	ScaleZ
Axis_2	1.5	0.5	1.5
Joint_3	1	1	1
Axis_3	1.5	0.5	1.5
Arm_3	2	0.5	1
Joint_4	1	1	1
Axis_4	1	1	1
Arm_4	1	0.5	1
Joint_5	1	1	1
Axis_5	1.5	0.5	1.5
Arm_5	2	0.5	1

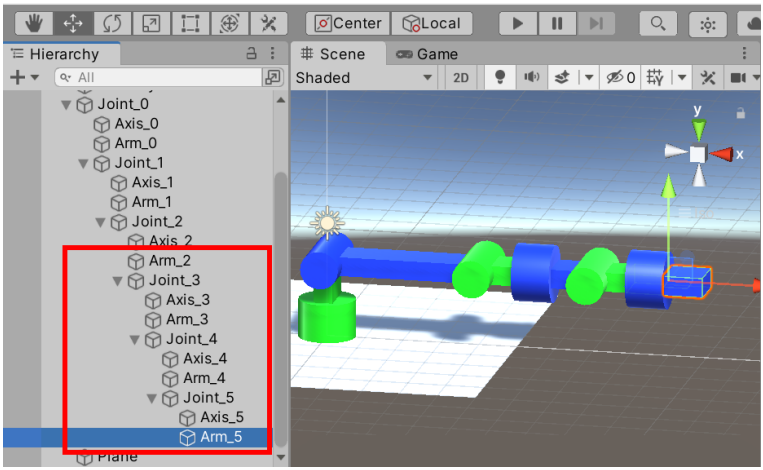

Figure 4-1

Make sure that the three axes (Joint_3, Joint_4, Joint_5) intersect at

one point (the center of Axis_4). And in this type of arm robot, the position of Axis_4 is determined by the 3 axes on the root side, and the posture of the hand is determined by the 3 axes on the tip side. Inverse kinematics uses this to solve. The outline of the solution is as follows.

1. Calculate the position of Axis_4 from the target hand position / posture(rotation).
2. Perform inverse kinematics calculations for the three root axes.
3. Estimate the rotation angles of the latter three axes using the rotation matrix.

Add UIs before we start inverse kinematics. Place three sliders to specify the rotation posture and related Texts. (Figure 4-2)
Set the slider value range to -90 to 90.

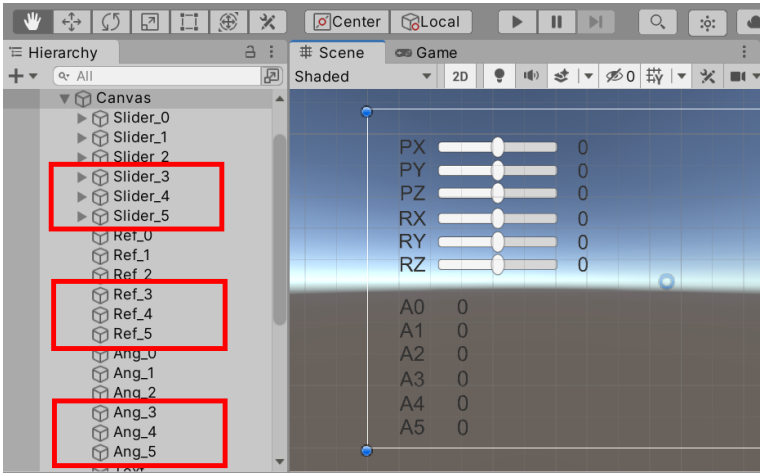

Figure 4-2

Finding the root coordinates of hand vector

We can find the position of Joint_4 (Axis_4) by returning to the upstream side from the position and posture of the hand. Joint_3 is upstream side of Joint_4, but make sure that Joint_3 does not affect the position of Joint_4 because it is the rotation of the axis along the arm.

To find the position of Axis_4, consider a vector with the length of (Arm_4 length + Arm_5 length), pointing in the direction of hand. A quaternion is used here to represent the rotation of this vector in three-dimensional space. Quaternion is a numerical value (a set of four numerical values) that expresses 3D rotation with the rotation axis and rotation angle, but in Unity it can be used without considering what quaternion exactly is.

Quaternions can be defined in Unity like

Quaternion.Euler (Rotation around X axis, Rotation around Y axis, Rotation around Z axis)

Assume that the vector V is the vector with the length of (**armL[4]** + **armL[5]**) lying along the X axis. Multiplying the quaternion Q and the vector V makes V rotate to the target posture. The coordinates of Axis_4 can be obtained by subtracting the X, Y, and Z components of V from the target coordinates.

Position.X = Ref.X – (Q * V).X
Position.Y = Ref.Y – (Q * V).Y
Position.Z = Ref.Z – (Q * V).Z

Inverse kinematics calculation of the first three axes

We can calculate inverse kinematics using the obtained Position.X,

Position.Y, and Position.Z. The method is exactly the same as in the previous chapter.

Introducing rotation matrix

Calculating the angles of the latter three axes is a little tricky. The first axis (in the second half) is already tilted, so we need to subtract that amount. First, let's examine how the coordinate system of each Joint accumulates rotations from the first axis (Joint_0) to the sixth axis (Joint_5).

For that we use a 3x3 rotation matrix. Consider a matrix that rotates a vector $(x_i,\ y_i,\ z_i)$ into a vector $(x_o,\ y_o,\ z_o)$ as in equation (4.1). For example, r_{00} is the amount of x_i projected onto x_o, and r_{21} is the amount of z_i projected onto z_o (Imagine sine and cosine).

$$\begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{4.1}$$

The rotation matrix for θ radians around Z axis is given by

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

The rotation matrix for θ radians around Y axis is given by

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 0 & 0 \\ -\sin\theta & 1 & \cos\theta \end{bmatrix} \tag{4.3}$$

The rotation matrix for θ radians around X axis is given by

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

(4.4)

Here, if we draw the coordinate system (Σi) in the figure to apply the rotation matrix to the arm robot model, it will be as shown in Figure 4-3. This is based on a technique called the Denavit-Hartenberg convention.The rules are

1) Align the Zi axis with the rotation axis of the joint.
2) The Xi axis should be perpendicular to the Zi axis and the Zi-1 axis.
3) Determine the Y axis according to the right-hand rule (when the thumb of the right hand is in the direction of Z axis and the fingers are in the X axis, the direction in which the palm faces is the Y axis)
4) The Xi axis intersects the Zi-1 axis
5) Right-handed screw rotation is the positive rotation

The coordinate system Σ6 (i = 6) is added to the position of the hand in the same direction as the coordinate system Σ5 (i = 5) for use in the calculation. And in the Figure 4-3, the X3 axis violates the rule 4. Originally, the coordinate system Σ3 should be placed at the position of the coordinate system Σ2 to obey the rule, but I put it there so that you can see it well.

Note that these axis settings are independent of Unity's X, Y, and Z axes.

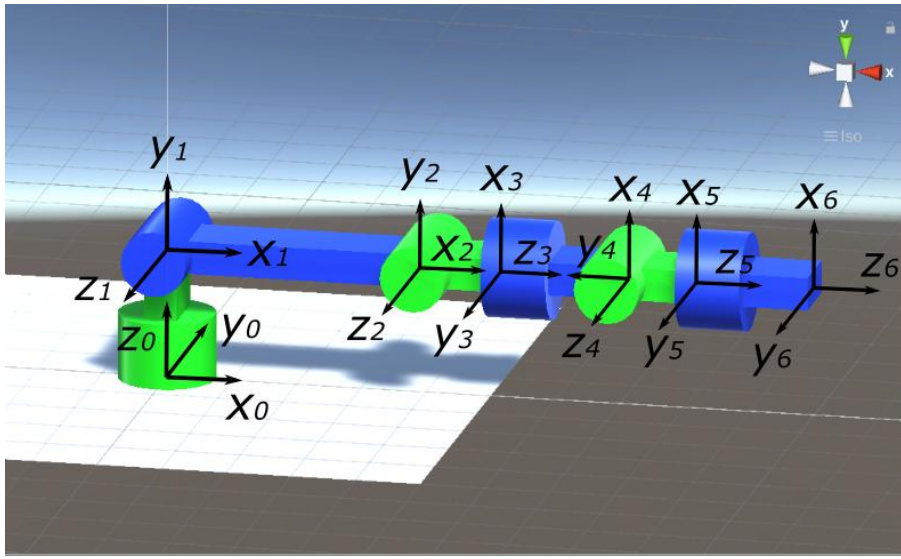

Figure 4—3

Here, we write the rotation matrix from the coordinate system Σi to the coordinate system Σj as R_j^i, and look at each rotation matrix concretely.

Combining the θ rotation around the Z0 axis with the projection of the coordinate system itself, R_1^0 is given by

$$R_1^0 = \begin{bmatrix} \cos\theta_0 & -\sin\theta_0 & 0 \\ \sin\theta_0 & \cos\theta_0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} c0 & 0 & s0 \\ s0 & 0 & -c0 \\ 0 & 1 & 1 \end{bmatrix} \tag{4.5}$$

The coordinate system projection from X0 to X1 is like

$$\begin{array}{c} \\ X0 \\ Y0 \\ Z0 \end{array} \begin{array}{c} \begin{array}{ccc} X1 & Y1 & Z1 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix} \end{array}$$

X0 projects on X1, Y0 projects on –Z1 and so on. Also, it is abbreviated as $\cos\theta_0$: c0, $\sin\theta_0$: s0.

In the same way, other rotation matrices are given by

$$R_2^1 = \begin{bmatrix} c1 & -s1 & 0 \\ s1 & c1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c1 & -s1 & 0 \\ s1 & c1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

$$R_3^2 = \begin{bmatrix} c2 & -s2 & 0 \\ s2 & c2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} -s2 & 0 & c2 \\ c2 & 0 & s2 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.7}$$

$$R_3^2 = \begin{bmatrix} c2 & -s2 & 0 \\ s2 & c2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} -s2 & 0 & c2 \\ c2 & 0 & s2 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.8}$$

$$R_4^3 = \begin{bmatrix} c3 & -s3 & 0 \\ s3 & c3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} c3 & 0 & -s3 \\ s3 & 0 & c3 \\ 0 & -1 & 0 \end{bmatrix} \tag{4.9}$$

$$R_5^4 = \begin{bmatrix} c4 & -s4 & 0 \\ s4 & c4 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} c4 & 0 & s4 \\ s4 & 0 & -c4 \\ 0 & -1 & 0 \end{bmatrix} \tag{4.10}$$

$$R_6^5 = \begin{bmatrix} c5 & -s5 & 0 \\ s5 & c5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

Using

$$R_6^3 = R_4^3 R_5^4 R_6^5$$

The latter half rotation of the robot R_6^3 is given by

$$R_6^3 = \begin{bmatrix} c3 & 0 & -s3 \\ s3 & 0 & c3 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} c4 & 0 & s4 \\ s4 & 0 & -c4 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} c5 & -s5 & 0 \\ s5 & c5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c3c4c5 - s3s5 & -c3c4s5 - s3c5 & c3s4 \\ s3c4c5 + c3s5 & -s3c4s5 + c3c5 & s3s4 \\ -s4c5 & s4s5 & c4 \end{bmatrix}$$

(4.12)

It seems that θ_4 can be found in the $c4$ part. So we need to get R_6^3 numerically.

From

$$R_6^0 = R_3^0 R_6^3$$

We get

$$R_6^3 = (R_3^0)^{-1} * R_6^0$$

(4.13)

-1 means the inverse matrix.

Since θ_0 to θ_2 are known from the inverse kinematics calculation for the first half joints, and R_6^0 can be obtained from the posture of the hand, the numerical solution of R_6^3 can be found using Equation (4.13).

After that, Equation (4.12) will be used to calculate $\theta 4 \rightarrow \theta 5 \rightarrow \theta 3$.

4-2 Coding Inverse Kinematics

We will code the contents of the previous section, but before that, since

the matrix calculation will be used this time, we have to import a library called "Math.Net". From

https://github.com/GlitchEnzo/NuGetForUnity/releases

Download

NuGetForUnity.3.0.2.unitypackage

and drop it in Assets in the Project view to install NuGet.

The NuGet menu will appear in the menu bar at the top. In the window that opens by NuGet |Manage NuGet Packages, enter "mathnet" in Search box and install MathNet.Numerics.

Open "JointController.cs" and add a line of List 4-1 in the top part of the code.

List 4-1

```
using MathNet.Numerics.LinearAlgebra.Single;
```

In the variable declaration part, change the number of elements for 6 degrees of freedom, and add Vector3 type variables **pos** and **rot** to store the target position and posture.

List 4-2

```
// robot
private GameObject[] joint = new GameObject[6];
private GameObject[] arm = new GameObject[6];
private float[] armL = new float[6];
private Vector3[] angle = new Vector3[6];
private Vector3 pos;
private Vector3 rot;
```

```
// UI
private GameObject[] slider = new GameObject[6];
private float[] sliderVal = new float[6];
private float[] prevSliderVal = new float[6];
private GameObject[] angText = new GameObject[6];
private GameObject[] posText = new GameObject[6];
```

First, we use the quaternion to find the position of Axis_4, the branch point between the positioning arms and the posture determination arms. **q** is a quaternion that represents the posture of the hand, **v** is a vector that represents the hand part, and the operation **(q * v)** aligns that vector with the target posture. **(x, y, z)** is the position of Axis_4.

List 4-3

```
void Update()
{
    for (int i = 0; i < joint.Length; i++)
    {
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
    }
    pos.x = sliderVal[0];
    pos.y = sliderVal[1];
    pos.z = sliderVal[2];
    rot.x = sliderVal[3];
    rot.y = sliderVal[4];
    rot.z = sliderVal[5];

    // calc Joint[4] position
    float endL = armL[4] + armL[5];
    Quaternion q = Quaternion.Euler(rot.x, rot.y, rot.z);
```

```
    Vector3 v = new Vector3(endL, 0f, 0f);
    float x = pos.x - (q * v).x;
    float y = pos.y - (q * v).y;
    float z = pos.z - (q * v).z;
    // continue to List 4-4
```

Next is the inverse kinematics part of the first half joints. Since the structure of the arm has been changed, rewrite **armL[2]** to **armL[2] + armL[3]**.

List 4-4

```
// calc angle[0],[1],[2]
angle[0].y = -Mathf.Atan2(z, x);
float a = x / Mathf.Cos(angle[0].y);
float b = y - armL[0];
if (Mathf.Pow(a * a + b * b, 0.5f) > (armL[1] + armL[2] + armL[3]))
{
   for (int i = 0; i < joint.Length; i++)
   {
      sliderVal[i] = prevSliderVal[i];
      slider[i].GetComponent<Slider>().value = sliderVal[i];
   }
   return;
}
else
{
   float alfa = Mathf.Acos((armL[1] * armL[1] + (armL[2] + armL[3])
      * (armL[2] + armL[3]) - a * a - b * b) / (2f * armL[1] *
      (armL[2] + armL[3])));
   angle[2].z = -Mathf.PI + alfa;
   float beta = Mathf.Acos((armL[1] * armL[1] + a * a + b * b –
```

```
        (armL[2] + armL[3]) * (armL[2] + armL[3])) / (2f * armL[1] *
            Mathf.Pow((a * a + b * b), 0.5f)));
    angle[1].z = Mathf.Atan2(b, a) + beta;
    // continue to List 4-5
```

Next is the part to find the angles of the latter three axes. The calculation of R_3^0 and R_6^0 are separate functions that returns 3 × 3 matrix. Using $c4$ $(\cos\theta_4)$ in Equation (4.12), we first find the **angle[4]**, then calculate the **angle[5]** from $-s4c5$ and the angle[3] from $s3s4$. However, the plus or minus sign may be reversed in the result of Acos, so it is necessary to use unused $s4s5$ and $c3s4$ to check if the sign is reversed.

Finally, invert the signs to match the rotation direction of Unity. However, the sign of **angle[4]** remains the same because the direction of the axis is also opposite.

List 4-5

```
// calc angle[3],[4],[5]
var R0_3 = CalcR0_3();
var R0_6 = CalcR0_6();
var R3_6 = R0_3.Inverse() * R0_6;
angle[4].z = Mathf.Acos(R3_6[2, 2]);
angle[5].x = Mathf.Acos(-R3_6[2, 0]
    / (Mathf.Sin(angle[4].z) + 1.0e-6f));
angle[3].x = Mathf.Asin(R3_6[1, 2]
    / (Mathf.Sin(angle[4].z) + 1.0e-6f));

// error check
if ((R3_6[2, 1] *
   (Mathf.Sin(angle[4].z) * Mathf.Sin(angle[5].x))) < 0)
{
```

```
    angle[5].x = -Mathf.Acos(-R3_6[2, 0]
        / (Mathf.Sin(angle[4].z) + 1.0e-6f));
}
if ((R3_6[0, 2] *
    (Mathf.Cos(angle[3].x) * Mathf.Sin(angle[4].z))) < 0)
{
    angle[3].x = -angle[3].x + Mathf.PI;
}

// correct angles
angle[3].x = -angle[3].x;
//angle[4].z = -angle[4].z;
angle[5].x = -angle[5].x;
for (int i = 0; i < joint.Length; i++)
{
    joint[i].transform.localEulerAngles = angle[i] * Mathf.Rad2Deg;
    posText[i].GetComponent<Text>().text
        = sliderVal[i].ToString("f2");
    prevSliderVal[i] = sliderVal[i];
}
angText[0].GetComponent<Text>().text
    = (angle[0].y * Mathf.Rad2Deg).ToString("f2");
angText[1].GetComponent<Text>().text
    = (angle[1].z * Mathf.Rad2Deg).ToString("f2");
angText[2].GetComponent<Text>().text
    = (angle[2].z * Mathf.Rad2Deg).ToString("f2");
angText[3].GetComponent<Text>().text
    = (angle[3].x * Mathf.Rad2Deg).ToString("f2");
angText[4].GetComponent<Text>().text
    = (angle[4].z * Mathf.Rad2Deg).ToString("f2");
angText[5].GetComponent<Text>().text
```

```
    = (angle[5].x * Mathf.Rad2Deg).ToString("f2");
}
}   // Update
```

The error check part is checking for inconsistencies by putting numbers in the unused elements of R_6^0. The reason why angle[3] and angle[5] are calculated with (+1.0e-6f) is to prevent the denominator from becoming 0.

List 4-6 shows the function that calculates R_3^0. It is the part of equations (4.5) to (4.7). MathNet matrix library is used.

List 4-6

```
DenseMatrix CalcR0_3()
{
    float[] C = new float[6];
    float[] S = new float[6];
    C[0] = Mathf.Cos(-angle[0].y);
    C[1] = Mathf.Cos(angle[1].z);
    C[2] = Mathf.Cos(angle[2].z);
    S[0] = Mathf.Sin(-angle[0].y);
    S[1] = Mathf.Sin(angle[1].z);
    S[2] = Mathf.Sin(angle[2].z);
    var R0_1 = DenseMatrix.OfArray(new float[,]
    {
        { C[0], 0f, S[0] },
        { S[0], 0f, -C[0] },
        { 0f, 1f, 0f }
    });
    var R1_2 = DenseMatrix.OfArray(new float[,]
    {
        { C[1], -S[1], 0f},
```

```
        { S[1], C[1], 0f },
        { 0f, 0f, 1f }
    });
    var R2_3 = DenseMatrix.OfArray(new float[,]
    {
        { -S[2], 0f, C[2] },
        { C[2], 0f, S[2] },
        { 0f, 1f, 0f }
    });
    return R0_1 * R1_2 * R2_3;
}
```

List 4-7 shows the function that calculates R_6^0. After rotating each axis using the angle specified by the slider, it's multiplied by the transpose matrix of the coordinate system.

List 4-7

```
DenseMatrix CalcR0_6()
{
    float rx = rot.x * Mathf.Deg2Rad;
    float ry = rot.y * Mathf.Deg2Rad;
    float rz = rot.z * Mathf.Deg2Rad;
    var R0_6z = DenseMatrix.OfArray(new float[,]
    {
        { Mathf.Cos(-ry), -Mathf.Sin(-ry), 0f },
        { Mathf.Sin(-ry), Mathf.Cos(-ry), 0f },
        { 0f, 0f, 1f }
    });
    var R0_6y = DenseMatrix.OfArray(new float[,]
    {
        { Mathf.Cos(-rz), 0f, Mathf.Sin(-rz) },
```

```
            { 0f, 1f, 0f },
            { -Mathf.Sin(-rz), 0f, Mathf.Cos(-rz) }
        });
        var R0_6x = DenseMatrix.OfArray(new float[,]
        {
            { 1f, 0f, 0f },
            { 0f, Mathf.Cos(-rx), -Mathf.Sin(-rx) },
            { 0f, Mathf.Sin(-rx), Mathf.Cos(-rx) }
        });
        var baseRot = DenseMatrix.OfArray(new float[,]
        {
            { 0f, 0f, 1f },
            { 0f, -1f, 0f },
            { 1f, 0f, 0f }
        });
        return R0_6z * R0_6x * R0_6y * baseRot;
    }
```

Let's save the script and run it.

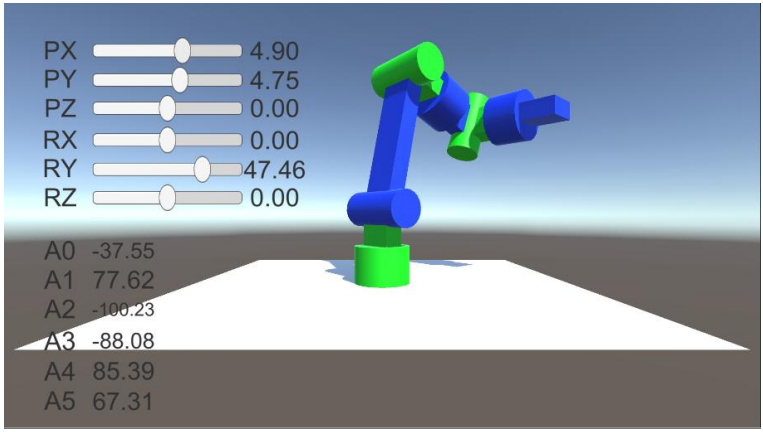

Figure 4－4

Chapter 5 Numerical Approach for Inverse Kinematics of 6 DOF Arm Robot

5-1 Numerical Approach

From this chapter, we will look at the numerical solution of inverse kinematics. The numerical solution method is different from the graphic solution method up to the previous chapter, it tries to bring the hand closer to the target position (posture) while gradually updating the joint angles by continuously using forward kinematics calculation.

The algorithm is as follows.

1. Calculate hand positions and postures by forward kinematics with the current set of joint angles
2. Check the distance from the target position and posture
3. Break the loop if the distance is small enough
4. If the distance is large, find a set of modified joint angles that will reduce the distance
5. Return to 1

A number of iterative methods have been proposed that expect to approach the target value while performing iterative calculations in this way. The question is how to find the corrected joint angles that reduces the distance, here we will take up the method of using the pseudo-inverse of the Jacobian matrix.

Consider the case of a 6DOF arm robot.
Let the vector

$$[\delta q_0 \quad \delta q_1 \quad \delta q_2 \quad \delta q_3 \quad \delta q_4 \quad \delta q_5]^{\mathrm{T}}$$

be the correction of joint angles.
And let the vector

$$[\delta p_x \quad \delta p_y \quad \delta p_z \quad \delta r_x \quad \delta r_y \quad \delta r_z]^{\mathrm{T}}$$

be the changes of position and posture of the joints.
By introducing a 6 × 6 matrix below, each element of the matrix can be regarded as the degree of influence on the degree of change in the hand position/posture when each joint angle is changed by a small amount.

$$\begin{bmatrix} \delta p_x \\ \delta p_y \\ \delta p_z \\ \delta r_x \\ \delta r_y \\ \delta r_z \end{bmatrix} = \begin{bmatrix} j_{00} & j_{01} & j_{02} & j_{03} & j_{04} & j_{05} \\ j_{10} & j_{11} & j_{11} & j_{13} & j_{14} & j_{15} \\ j_{20} & j_{21} & j_{22} & j_{23} & j_{24} & j_{25} \\ j_{30} & j_{31} & j_{32} & j_{33} & j_{34} & j_{35} \\ j_{40} & j_{41} & j_{41} & j_{43} & j_{44} & j_{45} \\ j_{50} & j_{51} & j_{52} & j_{53} & j_{54} & j_{55} \end{bmatrix} \begin{bmatrix} \delta q_0 \\ \delta q_1 \\ \delta q_2 \\ \delta q_3 \\ \delta q_4 \\ \delta q_5 \end{bmatrix}$$

(5.1)

This matrix is called the "Jacobian matrix".
The Jacobian matrix can be simply explained that it's the slope on a graph. Imagine an equation like (y = 2x) when (5.1) is expressed as

$$\Delta \mathrm{P} = \mathrm{J}\Delta \mathrm{Q}$$

(5.2)

Mathematically speaking, it is a differential coefficient that indicates how much the tip of the hand moves when joints are rotated a bit.
If you know the ΔP in which direction and how much you want to move your hand, the correction amount ΔQ of the joint angle can be calculated using

$$\mathrm{J}^{-1}\Delta \mathrm{P} = \Delta \mathrm{Q}$$

(5.3)

The Jacobian matrix changes depending on the posture of the entire robot arm and needs to be updated every frame.

Calculating Jacobian Matrix

As said above that each element of the Jacobian matrix is a differential coefficient, but since there are many variables (joints), it is a partial differential. For example, j_{00} is the amount of change in the x-coordinate of the hand position (δp_x) when the first joint angle is moved by a small amount (δq_0), while the other joints are fixed at the current angle.

Since the amount of change in the hand position means the peripheral speed in rotation around the joint axis, δp can be written by

$$\delta p = wa_i \times (p_6 - p_i) * \delta q \tag{5.4}$$

wa_i is the axis of rotation in the world coordinate system, and "×" is the outer product of the vectors. Please refer to Figure 5-1.

For δr the equation is as follows.

$$\delta r = wa_i * \delta q \tag{5.5}$$

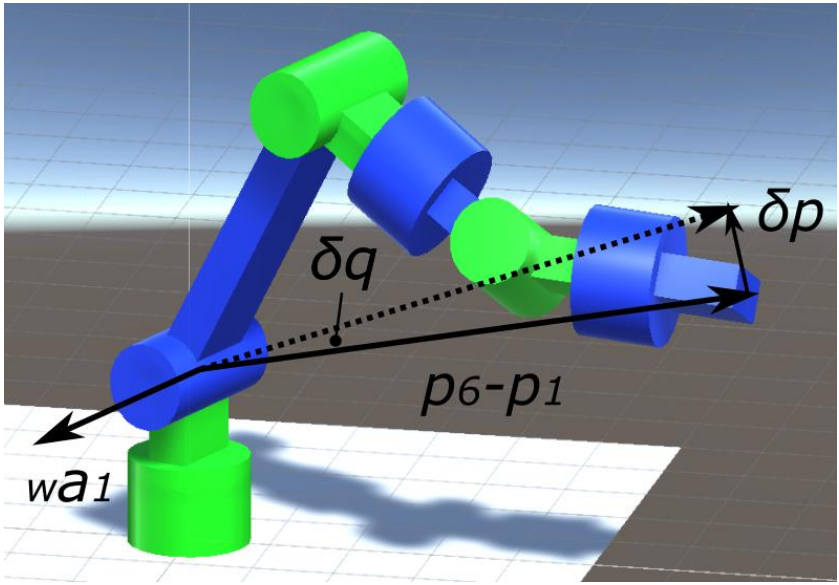

Figure 5－1

5-2 Coding Numerical Inverse Kinematics

We will implement inverse kinematics with 6 degrees of freedom robot, so duplicate IKProject4 and rename it IKProject5.

Open the Joint Controller.cs script to modify it. First we add variables.

List 5-1

```
// robot
private GameObject[] joint = new GameObject[6];
//private GameObject[] arm = new GameObject[6];
//private float[] armL = new float[6];
//private Vector3[] angle = new Vector3[6];
private float[] angle = new float[6];
// arm dimensions
```

```
private Vector3[] dim = new Vector3[6];
// position of joint in the world coordinate
private Vector3[] point = new Vector3[7];
// direction of axis
private Vector3[] axis = new Vector3[6];
// local quaternion on its parent
private Quaternion[] rotation = new Quaternion[6];
// quaternion in the world coordinate
private Quaternion[] wRotation = new Quaternion[6];
private Vector3 pos;   // target position
private Vector3 rot;   // target posture

private float lambda = 0.1f;   // used in iteration
```

Delete **arm** and **armL**. We declare **angle** as float type this time. It's just to avoid cluttering the code.

The dimensions of the arms are stored in the Vector3 type **dim**. Since the calculation of the arm rotation in the forward kinematics is performed by using the quaternion, if the dimension is set to Vector3, the 3D rotation can be expressed by a simple multiplication (Quaternion * Vector3).

point is the position of each local coordinate system for joints in the world coordinate system, and the number of elements is 7 because it includes the tip of the hand. (See Figure 4-3).

axis is the rotation axis of each coordinate system, and **rotation** and **wRotation** are quaternions that represent the rotation of each coordinate system locally (to its parent) and in the world. **wRotation** holds wa_1 in Figure 4-3 for example.

lambda is a coefficient for adjusting the convergence calculation. For the time being, it is set to 0.1.

Next inside **Start()** function, the X, Y, and Z axes are all along Unity's

coordinate system. Vector3 **axis[]** is added since we changed angle to float.

List 5-2

```
void Start()
{
    // robot
    for (int i = 0; i < joint.Length; i++)
    {
        joint[i] = GameObject.Find("Joint_" + i.ToString());
        //arm[i] = GameObject.Find("Arm_" + i.ToString());
        //if(i == 0) armL[i] = arm[i].transform.localScale.y;
        //else armL[i] = arm[i].transform.localScale.x;
    }

    // UI settings
    for (int i = 0; i < joint.Length; i++)
    {
        slider[i] = GameObject.Find("Slider_" + i.ToString());
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
        prevSliderVal[i] = sliderVal[i];
        posText[i] = GameObject.Find("Ref_" + i.ToString());
        angText[i] = GameObject.Find("Ang_" + i.ToString());
    }

    // arm dimensions
    dim[0] = new Vector3(0f, 2f, 0f);
    dim[1] = new Vector3(4f, 0f, 0f);
    dim[2] = new Vector3(1f, 0f, 0f);
    dim[3] = new Vector3(2f, 0f, 0f);
    dim[4] = new Vector3(1f, 0f, 0f);
    dim[5] = new Vector3(2f, 0f, 0f);
```

```
    // direction of rotation
    axis[0] = new Vector3(0f, 1f, 0f);
    axis[1] = new Vector3(0f, 0f, 1f);
    axis[2] = new Vector3(0f, 0f, 1f);
    axis[3] = new Vector3(1f, 0f, 0f);
    axis[4] = new Vector3(0f, 0f, 1f);
    axis[5] = new Vector3(1f, 0f, 0f);

    // rotation angle at start
    angle[0] = 0f;
    angle[1] = 90f;
    angle[2] = -90f;
    angle[3] = 0f;
    angle[4] = 0f;
    angle[5] = 0f;
}
```

In the **Update()** function, delete the part below "**// calc Joint [4] position**" and add a line **calcIK()** that performs inverse kinematics calculation.

List 5-3

```
void Update()
{
    for (int i = 0; i < joint.Length; i++)
    {
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
    }
    pos.x = sliderVal[0];
    pos.y = sliderVal[1];
    pos.z = sliderVal[2];
```

```
    rot.x = sliderVal[3];
    rot.y = sliderVal[4];
    rot.z = sliderVal[5];

    // IK
    CalcIK();
}
```

It is the main part of the inverse kinematics calculations. It basically follows the algorithm shown at the beginning, but the processing when the iteration does not converge is not implemented.

List 5-4

```
void CalcIK()
{
    for (int i = 0; i < 100; i++)    // iteration
    {
        ForwardKinematics();    // get position/posture of hand
        var err = CalcErr();        // distance from target
        float err_norm = (float)err.L2Norm();    // Absolute value of err
        if (err_norm < 1E-3)    // small enough and break
        {
            break;
        }
        var J = CalcJacobian(); // calculate Jacobian
        // correct angles
        var dAngle = lambda * J.PseudoInverse() * err;
        for (int ii = 0; ii < joint.Length; ii++)
        {
            angle[ii] += dAngle[ii, 0] * Mathf.Rad2Deg;
        }
```

```
        }
        // update joint angles
        for (int i = 0; i < joint.Length; i++)
        {
            rotation[i] = Quaternion.AngleAxis(angle[i], axis[i]);
            joint[i].transform.localRotation = rotation[i];
            prevSliderVal[i] = sliderVal[i];
            posText[i].GetComponent<Text>().text
                = sliderVal[i].ToString("f2");
            angText[i].GetComponent<Text>().text
                = angle[i].ToString("f2");
        }
    }
```

```
        var dAngle = lambda * J.PseudoInverse() * err;
```

This line does the equation (5.3).

PseudoInverse() function is used because it works well and returns matrix even when the robot has a peculiar posture (such as when the arm is fully extended) and the inverse matrix does not exist. Since it can be used as a function of Math.Net, we will use it without knowing what exactly is.

For the rotation of **joint[]**, we get the quaternion **rotation[]** from the rotation angle **angle[]** and the axis of rotation **axis[]** and set it to **joint[]**. Unity's transform.rotation is originally a quaternion.

Next, we see functions in **calcIK()**. **ForwardKinematics()** performs forward kinematic calculations.

List 5-5

```
void ForwardKinematics()
{
    point[0] = new Vector3(0f, 0f, 0f);
    wRotation[0] = Quaternion.AngleAxis(angle[0], axis[0]);
    for (int i = 1; i < joint.Length; i++)
    {
        point[i] = wRotation[i - 1] * dim[i - 1] + point[i - 1];
        rotation[i] = Quaternion.AngleAxis(angle[i], axis[i]);
        wRotation[i] = wRotation[i - 1] * rotation[i];
    }
    point[joint.Length] = wRotation[joint.Length - 1]
        * dim[joint.Length - 1] + point[joint.Length - 1];
}
```

wRotation[] is the rotation based on the world coordinate system, and **rotation[]** is the rotation based on the local (to the parent joint) coordinate system. Using the arm dimension **dim[]**, **wRotation[]** rotates the local joints in order from the root to finally get the position of the hand (**point[6]**). The posture (rotations) of the hand is stored in **wRotation[5]**.

List 5-6 shows the function **CalcErr()** that calculates the distance from the target. It returns a 6 × 1 matrix of position / posture error.

List 5-6

```
DenseMatrix CalcErr()
{
    // position error
    Vector3 perr = pos - point[6];
    // posture(rotation) error
    Quaternion rerr
```

```
        = Quaternion.Euler(rot) * Quaternion.Inverse(wRotation[5]);
        // quaternion to xyz angles
        Vector3 rerrVal
            = new Vector3(rerr.eulerAngles.x,
            rerr.eulerAngles.y,
            rerr.eulerAngles.z);
        if (rerrVal.x > 180f) rerrVal.x -= 360f;
        if (rerrVal.y > 180f) rerrVal.y -= 360f;
        if (rerrVal.z > 180f) rerrVal.z -= 360f;
        var err = DenseMatrix.OfArray(new float[,]
        {
            { perr.x },
            { perr.y },
            { perr.z },
            { rerrVal.x * Mathf.Deg2Rad},
            { rerrVal.y * Mathf.Deg2Rad},
            { rerrVal.z * Mathf.Deg2Rad}
        });
        return err;
    }
```

The position error **perr** is calculated subtracting the hand position **point[6]** obtained by ForwardKinematics from the target position **pos**. The posture error **rerr** is a quaternion, so it's a little unfamiliar, but this gives us the difference between **rot** and **wRotation[5]**. We have to convert it to rotation around the X, Y, Z axes, also have to adjust them to -180 to 180 range from the range of 0-360 °. Finally, it returns the err as a 6x1 matrix.

L2Norm() in List 5-4 is a Math.Net built-in function that finds the absolute square value of a 6x1 matrix to determine if the convergence calculation should be finished.

CalcJacobian() is a coded version of equations (5.4) and (5.5).

List 5-7

```
DenseMatrix CalcJacobian()
{
    Vector3 w0 = wRotation[0] * axis[0];
    Vector3 w1 = wRotation[1] * axis[1];
    Vector3 w2 = wRotation[2] * axis[2];
    Vector3 w3 = wRotation[3] * axis[3];
    Vector3 w4 = wRotation[4] * axis[4];
    Vector3 w5 = wRotation[5] * axis[5];
    Vector3 p0 = Vector3.Cross(w0, point[6] - point[0]);
    Vector3 p1 = Vector3.Cross(w1, point[6] - point[1]);
    Vector3 p2 = Vector3.Cross(w2, point[6] - point[2]);
    Vector3 p3 = Vector3.Cross(w3, point[6] - point[3]);
    Vector3 p4 = Vector3.Cross(w4, point[6] - point[4]);
    Vector3 p5 = Vector3.Cross(w5, point[6] - point[5]);

    var J = DenseMatrix.OfArray(new float[,]
    {
        { p0.x, p1.x, p2.x, p3.x, p4.x, p5.x },
        { p0.y, p1.y, p2.y, p3.y, p4.y, p5.y },
        { p0.z, p1.z, p2.z, p3.z, p4.z, p5.z },
        { w0.x, w1.x, w2.x, w3.x, w4.x, w5.x },
        { w0.y, w1.y, w2.y, w3.y, w4.y, w5.y },
        { w0.z, w1.z, w2.z, w3.z, w4.z, w5.z }
    });
    return J;
}
```

Let's save the script and run it. It should work, but when you slide the PZ slider too far, the robot gets unstable. The cause is that the arm is fully extended, so let's try to limit the joint angles in the next section.

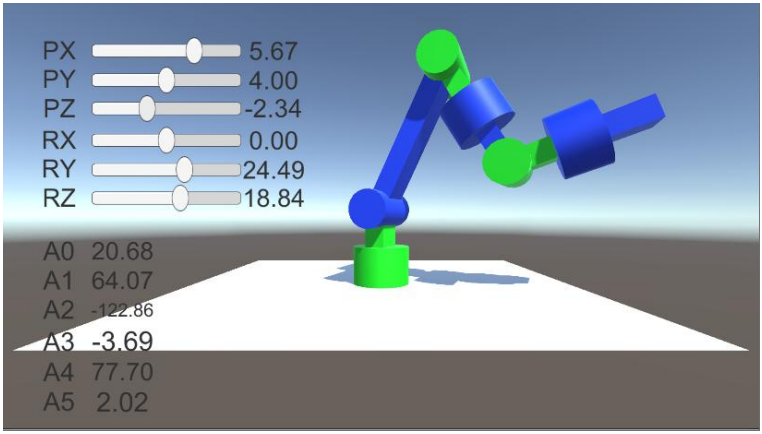

Figure 5－2

Limiting Joint Angles

We add three new variables. **prevAngle** is for recalling the previous angle when an error occurs and the joint angles are not updated. **minAngle[]** and **maxAngle[]** are variables for storing the minimum and maximum joint angles. It is set to -150/150 in **Start()** function. They are supposed to be set to separate values for each joint.

List 5-8

```
private float[] prevAngle = new float[6];
private float[] minAngle = new float[6];
private float[] maxAngle = new float[6];

void Start()
```

```
{
    /*
      skipped
    */
    for (int i = 0; i < Joint.Length; i++)
    {
        minAngle[i] = -150f;
        maxAngle[i] = 150f;
    }
}
```

Next we add two variables in the **calcIK()** function. **count** counts the convergence iteration and when it reaches 100 times (count = 99) the error processing will be executed. Boolean variable **outOfLimit** will be set to true when the angle is out of range after convergence.

In the error handling part, the slider position and joint angles (**angle[]**) are restored to their original values. If no error occurs, we update the joint angles as before and set new **angle[]** to **prevAngle[]**.

List 5-9

```
void CalcIK()
{
    int count = 0;
    bool outOfLimit = false;
    for (int i = 0; i < 100; i++)
    {
        count = i;
        ForwardKinematics();    // get position/posture of hand
        var err = CalcErr();        // distance from target
        float err_norm = (float)err.L2Norm();    // Absolute value of err
        if (err_norm < 1E-3)    // small enough and break
```

```
        {
            for (int ii = 0; ii < joint.Length; ii++)
            {
                if (angle[ii] < minAngle[ii] || angle[ii] > maxAngle[ii])
                {
                    outOfLimit = true;
                    break;
                }
            }
            break;
        }
        var J = CalcJacobian();// calculate Jacobian
        // correct angles
        var dAngle = lambda * J.PseudoInverse() * err;
        for (int ii = 0; ii < joint.Length; ii++)
        {
            angle[ii] += dAngle[ii, 0] * Mathf.Rad2Deg;
        }
    }
    // convergence fail or angle out of range
    if (count == 99 || outOfLimit)
    {
        for (int i = 0; i < joint.Length; i++)
        {
            slider[i].GetComponent<Slider>().value = prevSliderVal[i];
            angle[i] = prevAngle[i];
        }
    }
    else // update angles
    {
        for (int i = 0; i < joint.Length; i++)
```

```
        {
            rotation[i] = Quaternion.AngleAxis(angle[i], axis[i]);
            joint[i].transform.localRotation = rotation[i];
            prevSliderVal[i] = sliderVal[i];
            prevAngle[i] = angle[i];
            posText[i].GetComponent<Text>().text
              = sliderVal[i].ToString("f2");
            angText[i].GetComponent<Text>().text
              = angle[i].ToString("f2");
        }
    }
}
```

Modifying Robot Structure

The good thing about the numerical calculation method is that even if the structure of the robot changes a little, there is no need to reconsider the solution. Let's try changing a part of the robot model.

Shift Joint_3 slightly upward by setting Transform | Position | Y of Joint_3 to 0.5 .

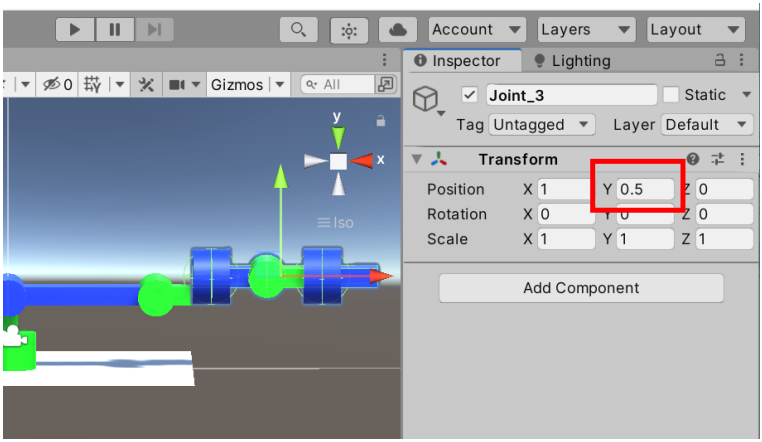

Figure 5－3

In the script change the **dim[2]** setting in **Start()** function like this.

```
dim[2] = new Vector3(1f, 0.5f, 0f);
```

This is the only script change, so let's save and run it.

Chapter 6 Another Type of Robot

6-1 UR3 Arm Robot

UR3 (Universal Robot) is a small arm robot with 6 axes of rotary drive. The joint configuration is very different from the robot models we have dealt with so far, but Unity provides a 3D model of this robot, so let's apply a numerical approach to it.

https://github.com/Unity-Technologies/articulations-robot-demo

Download the Unity project from here and Add from Unity Hub to open it.

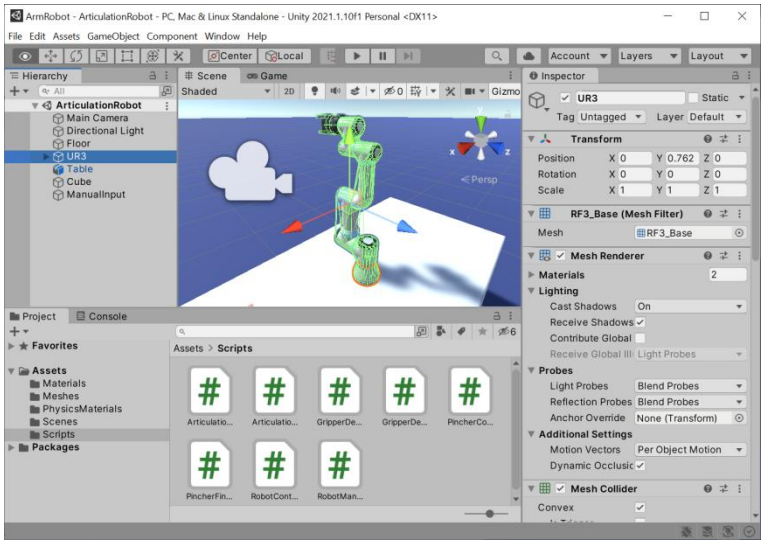

Figure 6－1

When you run the app, you can control the robot from the keyboard. Please refer to the github page for key assignment. If you expand UR3 in the Hierarchy view, arm parts will appear, but please note that we don't use Wrist1 this time.

We will modify this project to support inverse kinematics. First of all, you will need the Canvas that you have been using so far. But you don't want to remake that again, so we will bring it from IK Project 5.

・Open IK Project 5 and drag Canvas in the Hierarchy view to Assets and drop it there.
・A Prefab of Canvas is created in Assets. A Prefab is a reusable object, which is basically a copy.
・Drag and drop this Prefab of Canvas into the Assets of the Arm Robot project, and then drag and drop it into the Hierarchy view.
・Canvas has been copied to the Arm Robot project.

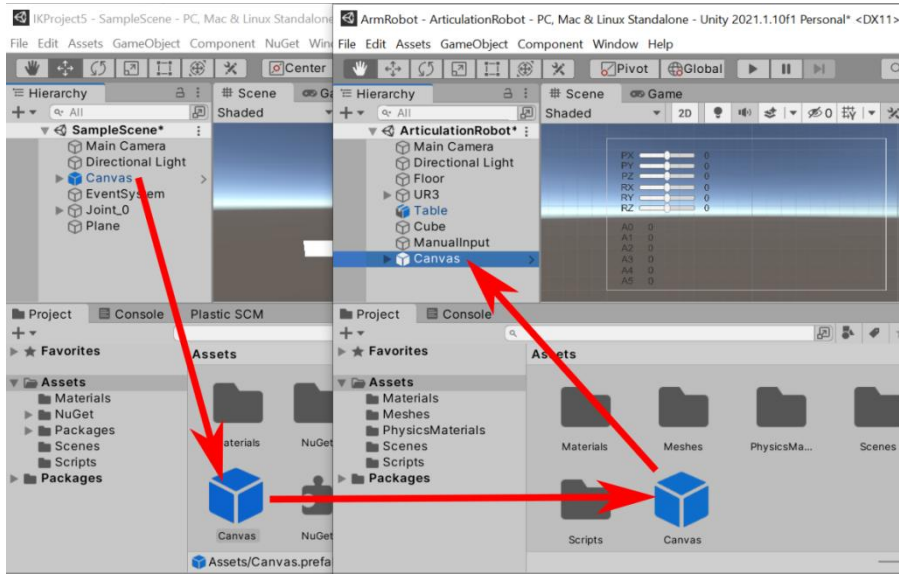

Figure 6－2

After adding Canvas, right-click in the Hierarchy view and select UI | Event System to add Event System. Without this, the slider will not respond.
Next, install MathNet.Numerics as in section 4-2.
Drag and drop the JointController.cs script from the Script folder of IKProject5 to the Script folder of ArmRobot project. Attach the brought script to the Inspector view with UR3 selected in the Hierarchy view.

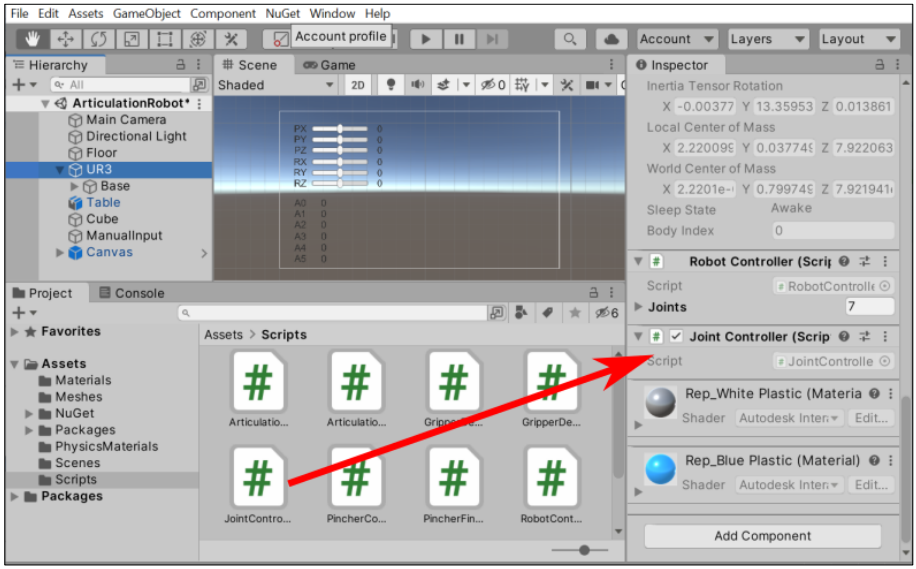

Figure 6－3

We will modify the script. Redefine the GameObject **joint[]** as an ArticulationBody. Also, make it public.

List 6-1

```
// robot
```

```
//private GameObject[] joint = new GameObject[6];
public ArticulationBody[] joint = new ArticulationBody[6];
```

In **Start()** function change the dimensions and axis / angle settings. Remove the **loop** part where finding joints. We will associate them to the objects in Unity editor.

For the dimension **dim[]** of each arm, expand UR3 in the Hierarchy view in the Unity screen, select each part, and copy the value of Transform | Position in the Inspector view. Since Base is a pedestal part, for **dim[0]** use the dimensions of Shoulder and so on. Also, we won't use Wrist1 this time, so add the dimensions of Wrist1 and Wrist2 for **dim[2]** dimensions.

Set the initial posture as it's bending elbow slightly.

List 6-2

```
void Start()
{
    // robot
    /*for (int i = 0; i < joint.Length; i++)
    {
        joint[i] = GameObject.Find("Joint_" + i.ToString());
    }*/

    // UI settings
    for (int i = 0; i < joint.Length; i++)
    {
        slider[i] = GameObject.Find("Slider_" + i.ToString());
        sliderVal[i] = slider[i].GetComponent<Slider>().value;
        prevSliderVal[i] = sliderVal[i];
        posText[i] = GameObject.Find("Ref_" + i.ToString());
        angText[i] = GameObject.Find("Ang_" + i.ToString());
```

```
    }

    // arm dimensions
    dim[0] = new Vector3(0f, 0.06584513f, -0.0643453f);
    dim[1] = new Vector3(-3.109574e-05f, 0.2422461f, -0.005073354f);
    dim[2] = new Vector3(3.109574e-05f, 0.1733774f + 0.04468203f,
          0.04262526f - 0.04507105f);
    dim[3] = new Vector3(0f, 0.03749204f, -0.03853976f);
    dim[4] = new Vector3(0f, 0.045f, -0.047f);
    dim[5] = new Vector3(0f, 0f, -0.13f);

    // direction of axis
    axis[0] = new Vector3(0f, 1f, 0f);
    axis[1] = new Vector3(0f, 0f, 1f);
    axis[2] = new Vector3(0f, 0f, 1f);
    axis[3] = new Vector3(0f, 0f, 1f);
    axis[4] = new Vector3(0f, 1f, 0f);
    axis[5] = new Vector3(0f, 0f, 1f);

    // angle at initial posture
    angle[0] = 0f;
    angle[1] = 30f;
    angle[2] = -60f;
    angle[3] = 30f;
    angle[4] = 0f;
    angle[5] = 0f;

    for (int i = 0; i < joint.Length; i++)
    {
       minAngle[i] = -160f;
       maxAngle[i] = 160f;
```

```
    }
}
```

For **CalcIK()** change the part that sets the joint angle of the robot. This way of setting angles is unique to this project.

List 6-3

```
for (int i = 0; i < joint.Length; i++)
{
    var drive = joint[i].xDrive;
    drive.target = angle[i];
    joint[i].xDrive = drive;
    prevSliderVal[i] = sliderVal[i];
    prevAngle[i] = angle[i];
    posText[i].GetComponent<Text>().text
        = sliderVal[i].ToString("f2");
    angText[i].GetComponent<Text>().text = angle[i].ToString("f2");
}
```

Change the first line of **ForwardKinematics()**. Since the first vertical rotation axis is installed on the pedestal, we have to raise the bottom at **point[0]** by that amount.

List 6-4

```
void ForwardKinematics()
{
    point[0] = new Vector3(0f, 0.08605486f, 0f);
    /* skipped */
}
```

That's all for modifying the script. Save the script and then select UR3 in the Hierarchy view of the Unity screen to attach the script to the Inspector view.

There is an item called Joint in the component of Joint Controller, and you can set objects there in Element0 to Element5. Drag and drop Base…HandE from Hierarchy view. Again Wrist1 will not be used.

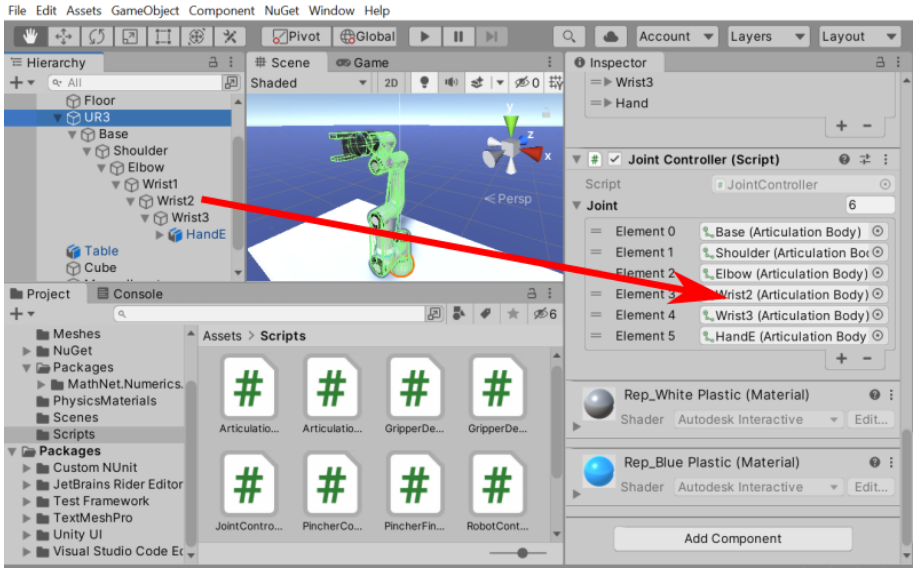

Figure 6－4

One more thing. In the Inspector view, open the Articulation Body components of Shoulder and HandE and set Anchor Rotation as follows.

Shoulder： X = 90　Y = 270　Z = 0

HandE： X = 0　Y = 0　Z = 270

Try running it now. You may get warnings in red texts, but it's possible to play. Using numerical method it is possible to apply the inverse kinematics of robots with different structures in the same way as in the

previous chapter. If you find it difficult to see the details, try pressing Maximize On Play in the Game tab.

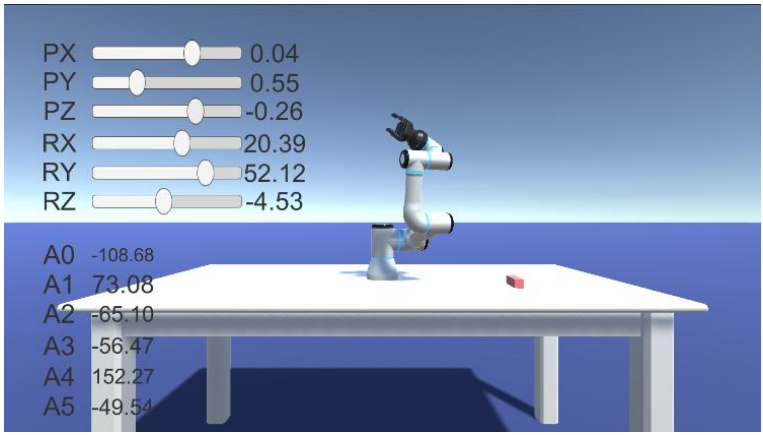

Figure 6—5