Yinhong Qin
yq2021@nyu.edu

Project 1

ROB 6003
November 16, 2021

Question 1

1. Computing the skew symmetric form of a given vector.
   Consider a vector $\mathbf{v} \in \mathbb{R}^3$, its skew symmetric form is:

$$[\mathbf{v}] = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

```python
# vec_to_skew
def vec_to_skew(vec):
    result = np.array([[0, -vec[2, 0], vec[1, 0]], [vec[2, 0], 0, -vec[0, 0]], [-vec[1, 0], vec[0, 0], 0]])
    return result
```

Figure 1: Screenshot of function "vec_to_skew"

2. Computing the skew symmetric form of a given twist vector.
   Consider a vector $\mathcal{V} = \begin{bmatrix} \omega \\ v \end{bmatrix} \in \mathbb{R}^6$, its skew symmetric form is:

$$[\mathcal{V}] = \begin{bmatrix} [w] & v \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{4\times 4}$$

```python
def get_angular_velocity_and_linear_velocity_from_twist_vector(twist_vector):
    angular_velocity = twist_vector[0: 3]
    linear_velocity = twist_vector[3: 6]

    angular_velocity = np.resize(angular_velocity, (3, 1))
    linear_velocity = np.resize(linear_velocity, (3, 1))
    return [angular_velocity, linear_velocity]


# twist_to_skew
def twist_to_skew(twist_vector):
    [angular_w, linear_v] = get_angular_velocity_and_linear_velocity_from_twist_vector(twist_vector)
    bracket_w = vec_to_skew(angular_w)

    result = np.append(bracket_w, linear_v, axis=1)
    const = np.array([[0, 0, 0, 0]])
    result = np.append(result, const, axis=0)

    return result
```

Figure 2: Screenshot of function "twist_to_skew"

3. Computing the exponential of a twist $\mathcal{V}$.

   The screw axis of $\mathcal{V}$ is $\mathcal{S} = \begin{bmatrix} \omega_s \\ v_s \end{bmatrix}$.

   If $\|\omega_s\| = 1$, then for any $\theta \in \mathbb{R}$, its exponential is:

$$\exp([\mathcal{S}]\theta) = \begin{bmatrix} \exp([\omega_s]\theta) & I\theta + (1 - \cos\theta)[\omega] + (\theta - \sin\theta)[\omega]^2 \\ 0 & 1 \end{bmatrix}$$

   If $\omega = 0$ and $\|v\| = 1$, then we have:

$$\exp([\mathcal{S}]\theta) = \begin{bmatrix} I & v\theta \\ 0 & 1 \end{bmatrix}$$

   Note: The function parameter "twist_vector" here is actually $[\mathcal{S}]\theta$.

```python
def get_angular_velocity_and_linear_velocity_from_twist_vector(twist_vector):
    angular_velocity = twist_vector[0: 3]
    linear_velocity = twist_vector[3: 6]

    angular_velocity = np.resize(angular_velocity, (3, 1))
    linear_velocity = np.resize(linear_velocity, (3, 1))
    return [angular_velocity, linear_velocity]


# exp_twist_bracket
def exp_twist_bracket(twist_vector):
    [angular_w, linear_v] = get_angular_velocity_and_linear_velocity_from_twist_vector(twist_vector)
    zero_vec = np.array([[0], [0], [0]])
    const_vec = np.array([[0, 0, 0, 1]])
    if np.array_equal(zero_vec, angular_w):
        result = np.append(np.append(np.identity(3), linear_v, axis=1), const_vec, axis=0)
    else:
        rotation_theta = np.linalg.norm(angular_w)
        rotation_axis = angular_w / rotation_theta

        rotation_part = rodrigues_form(rotation_axis, rotation_theta)
        bracket_w = vec_to_skew(rotation_axis)
        J_mat = np.identity(3) + 1 / rotation_theta * ((1 - np.cos(rotation_theta)) * bracket_w
                                                       + (rotation_theta - np.sin(rotation_theta)) * np.matmul(bracket_w, bracket_w))

        translation_part = np.matmul(J_mat, linear_v)
        result = np.append(np.append(rotation_part, translation_part, axis=1), const_vec, axis=0)

    return result
```

Figure 3: Screenshot of function "exp_twist_bracket"

4. Computing the inverse of a homogeneous transformation T.

   The inverse of a homogeneous transformation $T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \in SE(3)$ is:

$$\begin{aligned} T^{-1} &= \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}^{-1} \\ &= \begin{bmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{bmatrix} \end{aligned}$$

```python
# inverseT
def inverseT(transformation):
    [rotation, translation] = get_rotation_and_translation_from_transform_matrix(transformation)
    rotation = rotation.transpose()

    const = np.array([[0, 0, 0, 1]])

    result = np.append(rotation, np.matmul(-rotation, translation), axis=1)
    result = np.append(result, const, axis=0)
    return result
```

Figure 4: Screenshot of function "inverseT"

5. Computing the adjoint of a homogeneous transformation T.

   The adjoint of a homogeneous transformation $T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \in SE(3)$ is:

$$[Adj_T] = \begin{bmatrix} R^{-1} & 0 \\ [p]R & R \end{bmatrix} \in \mathbb{R}^{6\times 6}$$

```python
def get_rotation_and_translation_from_transform_matrix(transformation):
    rotation = transformation[0: 3, 0: 3]

    translation = transformation[0:3, 3]
    translation = np.resize(translation, (3, 1))

    return [rotation, translation]


# getAdjoint
def getAdjoint(transformation):
    [rotation, translation] = get_rotation_and_translation_from_transform_matrix(transformation)
    zero_mat = np.zeros((3, 3))

    adjoint_mat = np.append(np.append(rotation, zero_mat, axis=1),
                            np.append(np.matmul(vec_to_skew(translation), rotation), rotation, axis=1), axis=0)
    return adjoint_mat
```

Figure 5: Screenshot of function "getAdjoint"

Question 2 & 3

Given a set of joint configuration θ, the corresponding forward kinematics $T(\theta)$ can be computed by this formula:

$$T(\theta) = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_n]\theta_n}$$

The spatial Jacobian of a given set of configuration θ is:

$$J_s(\theta) = [J_{s1} \quad J_{s2}(\theta) \quad \cdots \quad J_{sn}(\theta)]$$

where

$$J_{s1} = \mathcal{S}_1$$

and

$$J_{sn}(\theta) = Ad_{e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \ldots e^{[\mathcal{S}_{n-1}]\theta_{n-1}}} \mathcal{S}_n$$

The spatial screw axes of this "Kuka iiwa 14" robot is listed in this matrix below. The i-th column represents the spatial screw axis of the i-th joint.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & -0.36 & 0 & 0.78 & 0 & -1.18 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To solved question 2 and 3, I designed a class "RoboticArm". Below is the definition and initialization method of it.

```python
class RoboticArm:
    name = ""
    degree_Of_FreeDom = 0
    initial_position = []  # Position of end effector at zero-configuration
    configurations = []
    spatial_screw_axis = []

    endEffector_pose = np.identity(4)
    spatial_Jacobian = np.array([[], [], [], [], [], []])

    def __init__(self, name_string, dof, starting_position):
        self.name = name_string
        self.degree_Of_FreeDom = dof
        self.configurations = np.zeros(dof)
        self.initial_position = starting_position

    def get_end_effector_pose(self):
        return self.endEffector_pose

    def initialize_screw_axis(self, axis: np.array):
        if axis.shape[0] != 6:
            print("The dimension of single screw axis should be 6!!!")

        if axis.shape[1] != self.degree_Of_FreeDom:
            print("The number of input Screw axes does not match the DOF of the Robot!!!")

        for i in range(self.degree_Of_FreeDom):
            self.spatial_screw_axis.append(axis[:, [i]])

    def forward_kinematics(self, theta: np.array):
        axes = self.spatial_screw_axis.copy()
        initial_translation = np.resize(self.initial_position, (3, 1))
        self.endEffector_pose = get_transformation_matrix_from_rotation_and_translation(np.identity(3), initial_translation)

        for i in range(self.degree_Of_FreeDom):
            screw_axis = axes.pop()
            self.endEffector_pose = np.matmul(exp_twist_bracket(screw_axis * theta[self.degree_Of_FreeDom - i - 1]),
                                              self.endEffector_pose)

    def get_space_Jacobian(self, theta: np.array):
        self.spatial_Jacobian = np.array([[], [], [], [], [], []])
        axes = self.spatial_screw_axis.copy()
        multi_mat = np.identity(4)

        for i in range(self.degree_Of_FreeDom):
            screw_axis = axes[i]
            Jacobian_i = np.matmul(getAdjoint(multi_mat), screw_axis)
            self.spatial_Jacobian = np.append(self.spatial_Jacobian, Jacobian_i, axis=1)

            if i != self.degree_Of_FreeDom - 1:
                multi_mat = np.matmul(multi_mat, exp_twist_bracket(screw_axis * theta[i]))

        return self.spatial_Jacobian

# Robot initialization
robot = RoboticArm("Kuka_iiwa_14", 7, [0, 0, 1.301])
Spatial_Screw_Axises = np.array([[0, 0, 0, 0, 0, 0, 0], [0, 1, 0, -1, 0, 1, 0], [1, 0, 1, 0, 1, 0, 1],
                                 [0, -0.36, 0, 0.78, 0, -1.18, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]])
robot.initialize_screw_axis(Spatial_Screw_Axises)
```

Figure 6: Screenshot of the class "RoboticArm".

To compute the forward kinematics and spatial Jacobian with a given configuration of this Robot, we can use the code listed below:

```
theta_configurations = np.array([-1.2, 0.7, 2.8, 0.7, 1.2, 0.2, 0.3])
robot.forward_kinematics(theta_configurations)
print("End effctor pose is: \n", robot.get_end_effector_pose())
print("\nCurrent Spatial Jacobian is: \n", robot.get_space_Jacobian(theta_configurations))
```

```
End effctor pose is:
 [[-0.96690733 -0.2540491  -0.02343634  0.1534601 ]
 [ 0.0976152  -0.28351568 -0.95398644 -0.75567256]
 [ 0.23571483 -0.92470423  0.29893244  0.79498287]
 [ 0.          0.          0.          1.        ]]

Current Spatial Jacobian is:
 [[ 0.00000000e+00  9.32039086e-01  2.33437275e-01  9.71028842e-01
   1.45630607e-01 -5.28443029e-01 -2.34363366e-02]
 [ 0.00000000e+00  3.62357754e-01 -6.00436064e-01  1.02620976e-01
  -9.70142642e-01 -2.42005608e-01 -9.53986444e-01]
 [ 1.00000000e+00  0.00000000e+00  7.64842187e-01 -2.15805291e-01
   1.93945821e-01 -8.13745200e-01  2.98932440e-01]
 [ 0.00000000e+00 -1.30448792e-01  2.16156983e-01 -1.54864119e-02
   6.11984012e-01  7.04629163e-01  5.32507840e-01]
 [ 0.00000000e+00  3.35534071e-01  8.40374188e-02  6.82655928e-01
   8.01933227e-02 -2.73803900e-01 -6.45056887e-02]
 [ 0.00000000e+00  0.00000000e+00 -6.93889390e-18  2.54938445e-01
  -5.83907460e-02 -3.76154957e-01 -1.64109053e-01]]
```

Figure 7: Screenshot of the solution to Question 2 & 3.

Question 4

The position of end-effector for each configuration θ is:

$$T(\theta) = forward\_kinematics(\theta) = \exp([\mathcal{S}_1]\theta_1) \cdots \exp([\mathcal{S}_n]\theta_n)M$$

where M is the pose of end-effector at zero-configuration.

The code to compute forward kinematics of each configuration and plotting the results is listed below:

```
num_of_configurations = joint_trajectory.shape[1]
endEffector_position_list = []
for i in range(num_of_configurations):
    theta = joint_trajectory[:, i]
    robot.forward_kinematics(theta)
    [rot, trans] = get_rotation_and_translation_from_transform_matrix(robot.get_end_effector_pose())
    endEffector_position_list.append(trans)

x_pos_list = []
y_pos_list = []
z_pos_list = []


for i in range(num_of_configurations):
    x_pos_list.append(endEffector_position_list[i][0, 0])
    y_pos_list.append(endEffector_position_list[i][1, 0])
    z_pos_list.append(endEffector_position_list[i][2, 0])

plt.figure(figsize=(15, 15))

plt.subplot(3, 1, 1)
plt.plot(x_pos_list, y_pos_list)
plt.xlabel("x-axis position", fontsize=20)
plt.xticks(fontsize=15)
plt.ylabel("y-axis position", fontsize=20)
plt.yticks(fontsize=15)
plt.title("X vs Y")
plt.grid()

plt.subplot(3, 1, 2)
plt.plot(x_pos_list, z_pos_list)
plt.xlabel("x-axis position", fontsize=20)
plt.xticks(fontsize=15)
plt.ylabel("z-axis position", fontsize=20)
plt.yticks(fontsize=15)
plt.title("X vs Z")
plt.grid()

plt.subplot(3, 1, 3)
plt.plot(y_pos_list, z_pos_list)
plt.xlabel("y-axis position", fontsize=20)
plt.xticks(fontsize=15)
plt.ylabel("z-axis position", fontsize=20)
plt.yticks(fontsize=15)
plt.title("Y vs Z")
plt.grid()

plt.subplots_adjust(hspace=0.5)
```

Figure 8: Screenshot of the solution to Question 4.

The result is:

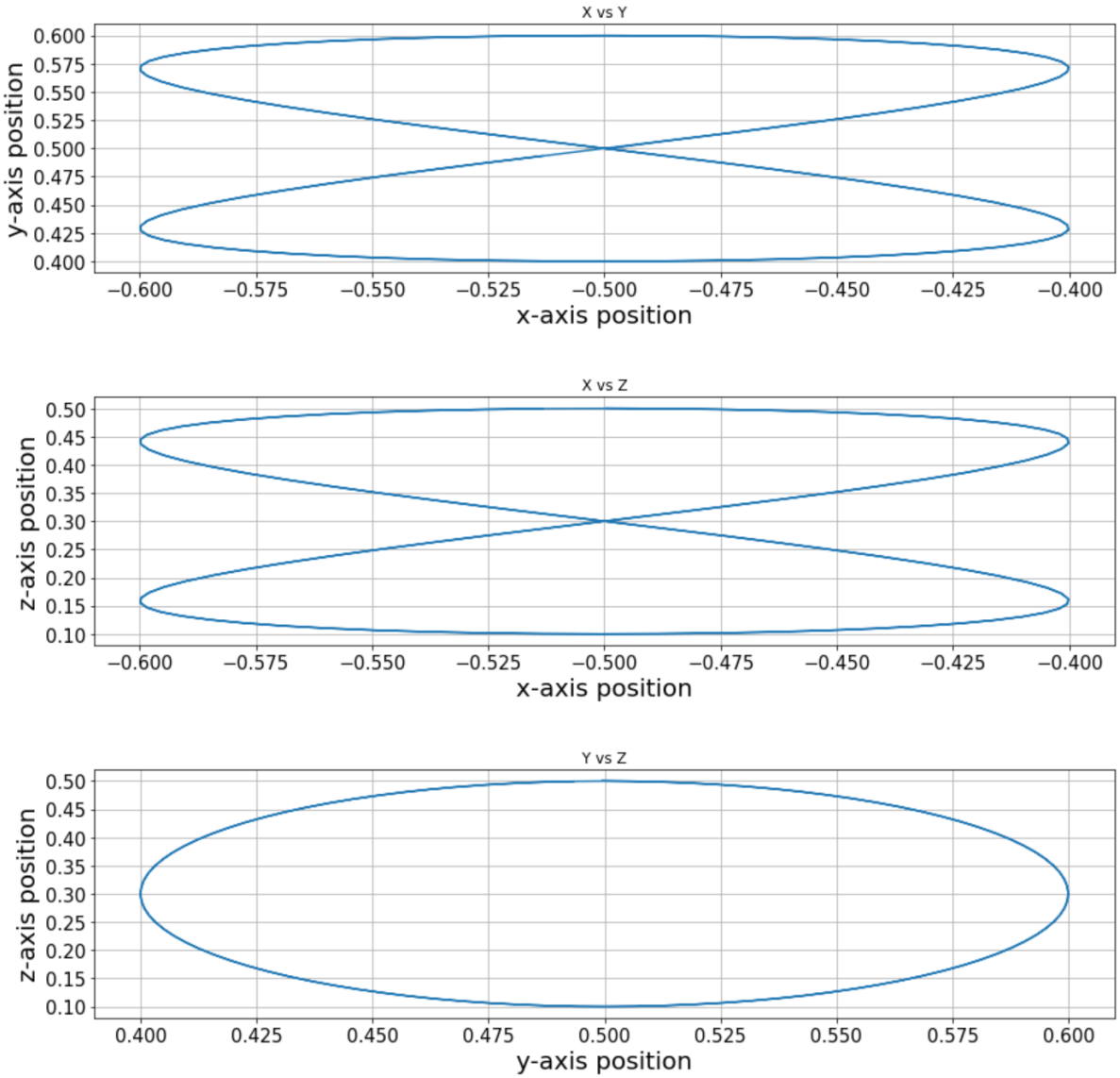

Figure 9: Analysis Result.

The robot hand is drawing like the number “8”.

Question 5

Let the frame with same origin as the end-effector frame but oriented like the spatial frame be denoted as T_{new}.
The pose of end-effector for a given configuration θ is

$$T(\theta) = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$$

Thus, we can know that,

$$T_{new}(\theta) = \begin{bmatrix} I & p \\ 0 & 1 \end{bmatrix}$$

The code to compute the spatial twist of the end-effector with each configuration in different reference frame is listed below:

```
twist_in_spatial_frame = []
twist_in_end_effector_frame = []
twist_in_thrid_frame = []

for i in range(num_of_configurations):
    theta = joint_trajectory[:, i]
    dtheta = joint_velocities[:, [i]]
    current_spatial_jacobian = robot.get_space_Jacobian(theta)
    current_spatial_twist_of_endEffector = np.matmul(current_spatial_jacobian, dtheta)
    twist_in_spatial_frame.append(current_spatial_twist_of_endEffector)

    robot.forward_kinematics(theta)
    T_sb = robot.get_end_effector_pose()
    T_bs = inverseT(T_sb)
    body_twist = np.matmul(getAdjoint(T_bs), current_spatial_twist_of_endEffector)
    twist_in_end_effector_frame.append(body_twist)

    [rot, trans] = get_rotation_and_translation_from_transform_matrix(T_sb)
    T_s_new = get_transformation_matrix_from_rotation_and_translation(np.identity(3), trans)
    T_s_new_inverse = inverseT(T_s_new)
    new_twist = np.matmul(getAdjoint(T_s_new_inverse), current_spatial_twist_of_endEffector)
    twist_in_thrid_frame.append(new_twist)
```

Figure 10: Screenshot of the code of computing the spatial twist of the end-effector with each configuration in different reference frame.

The x,y,z components of linear velocity in spatial frame is listed below:

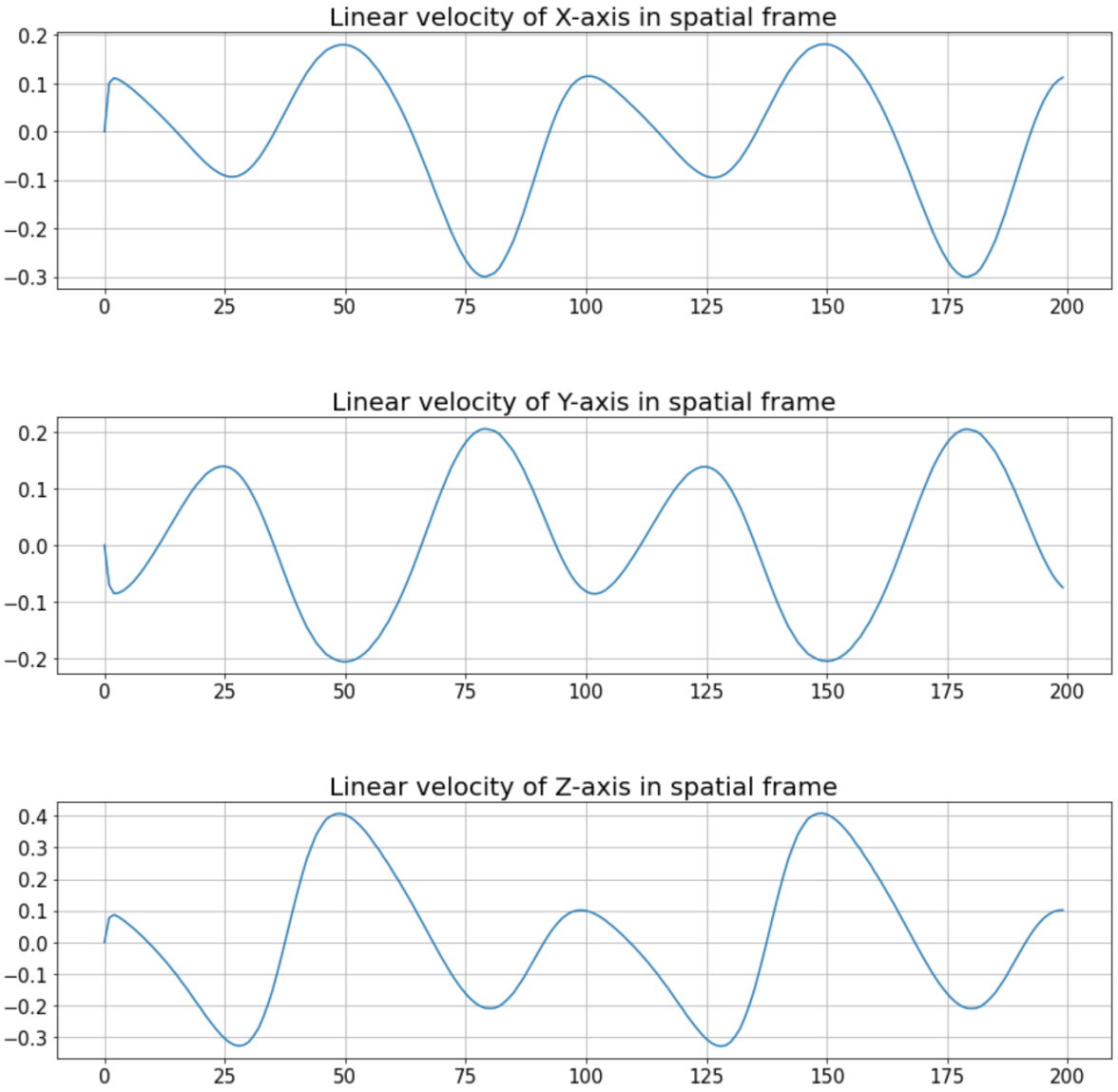

Figure 11: Analysis Result.

The x,y,z components of linear velocity in end-effector frame is listed below:

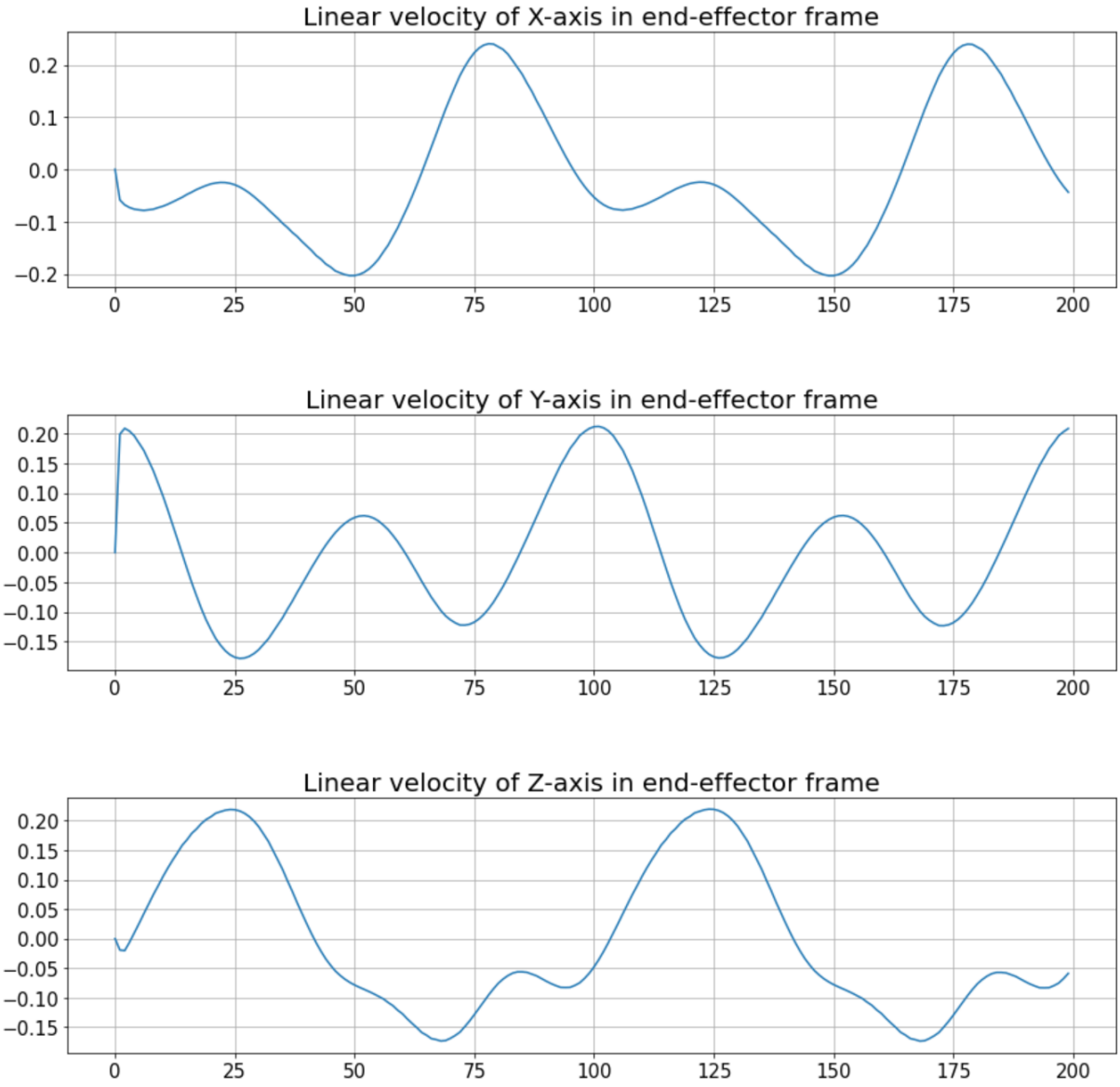

Figure 12: Analysis Result.

The x,y,z components of linear velocity in third frame (the frame with same origin as the end-effector frame but oriented like the spatial frame) is listed below:

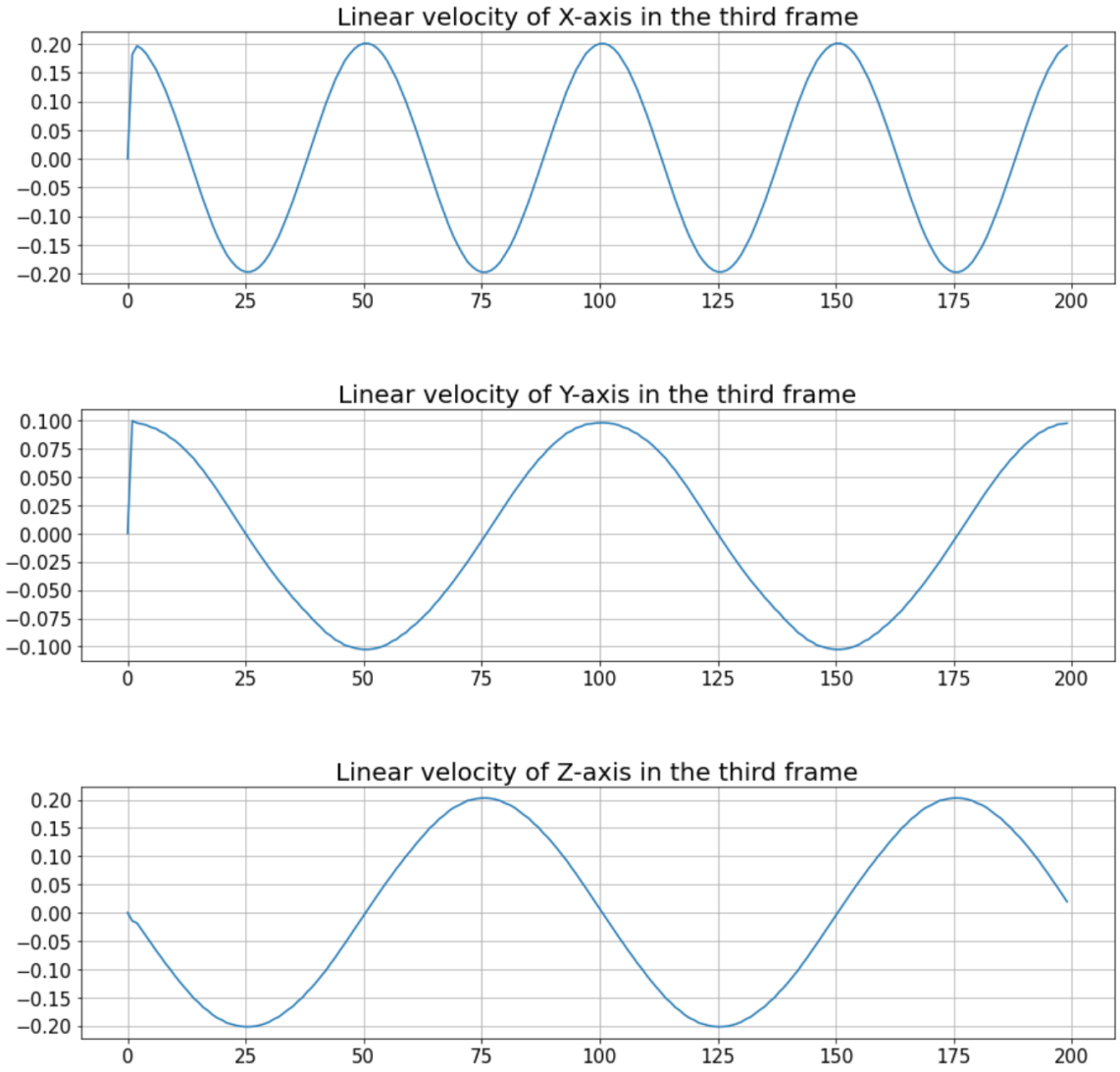

Figure 13: Analysis Result.

The comparison of each x,y,z components of linear velocity in different frame is listed below.

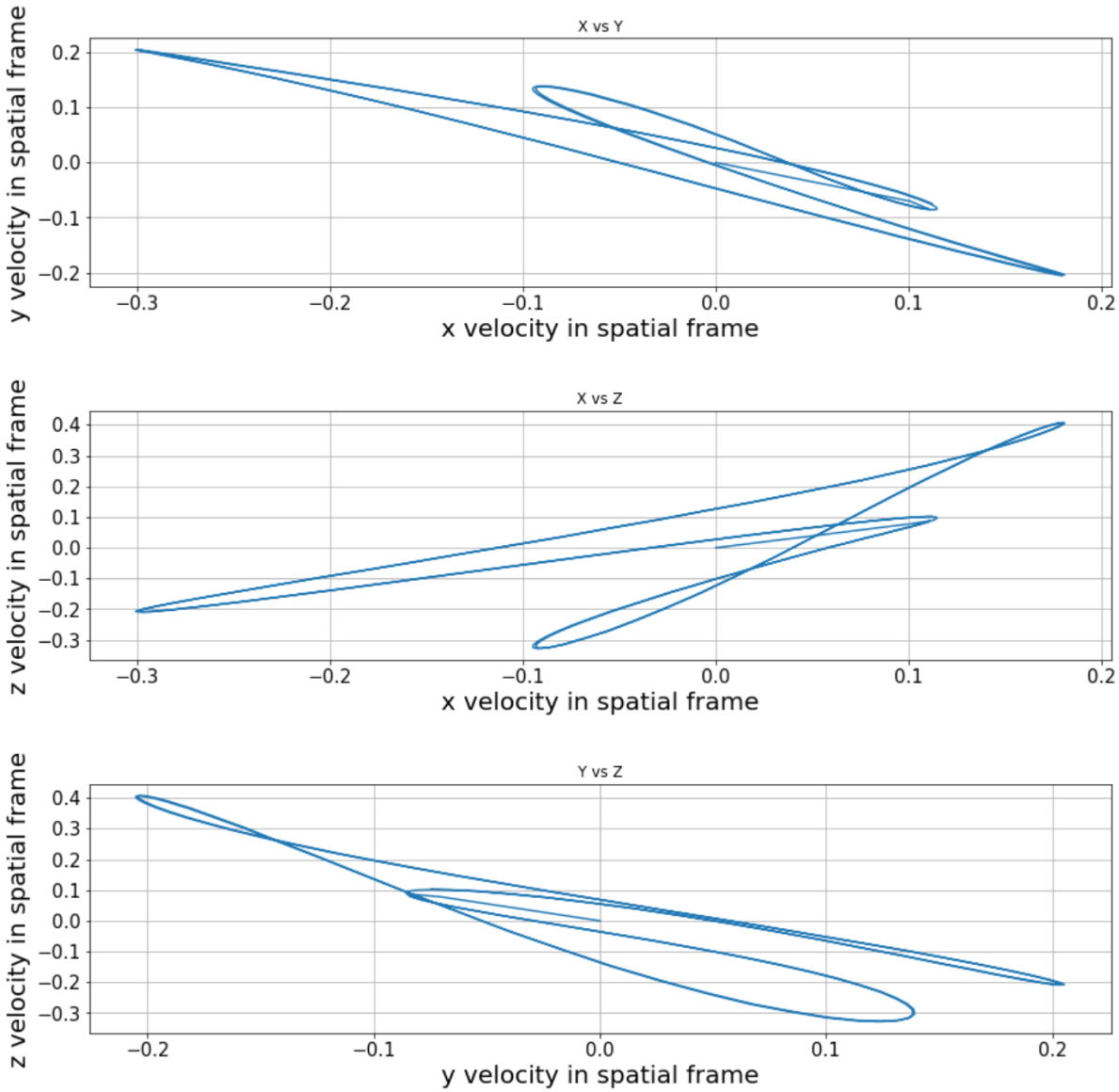

Figure 14: Analysis Result.

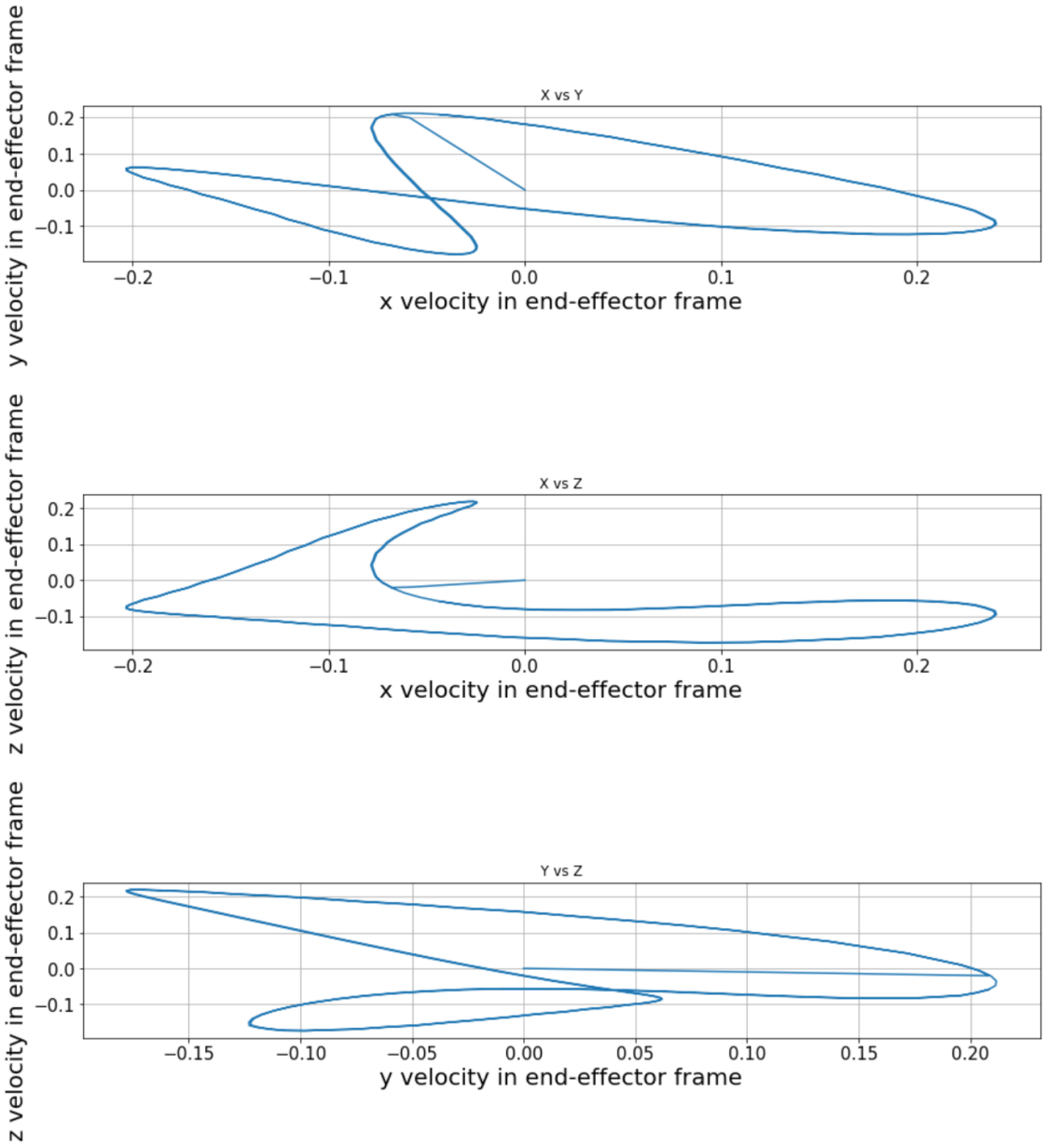

Figure 15: Analysis Result.

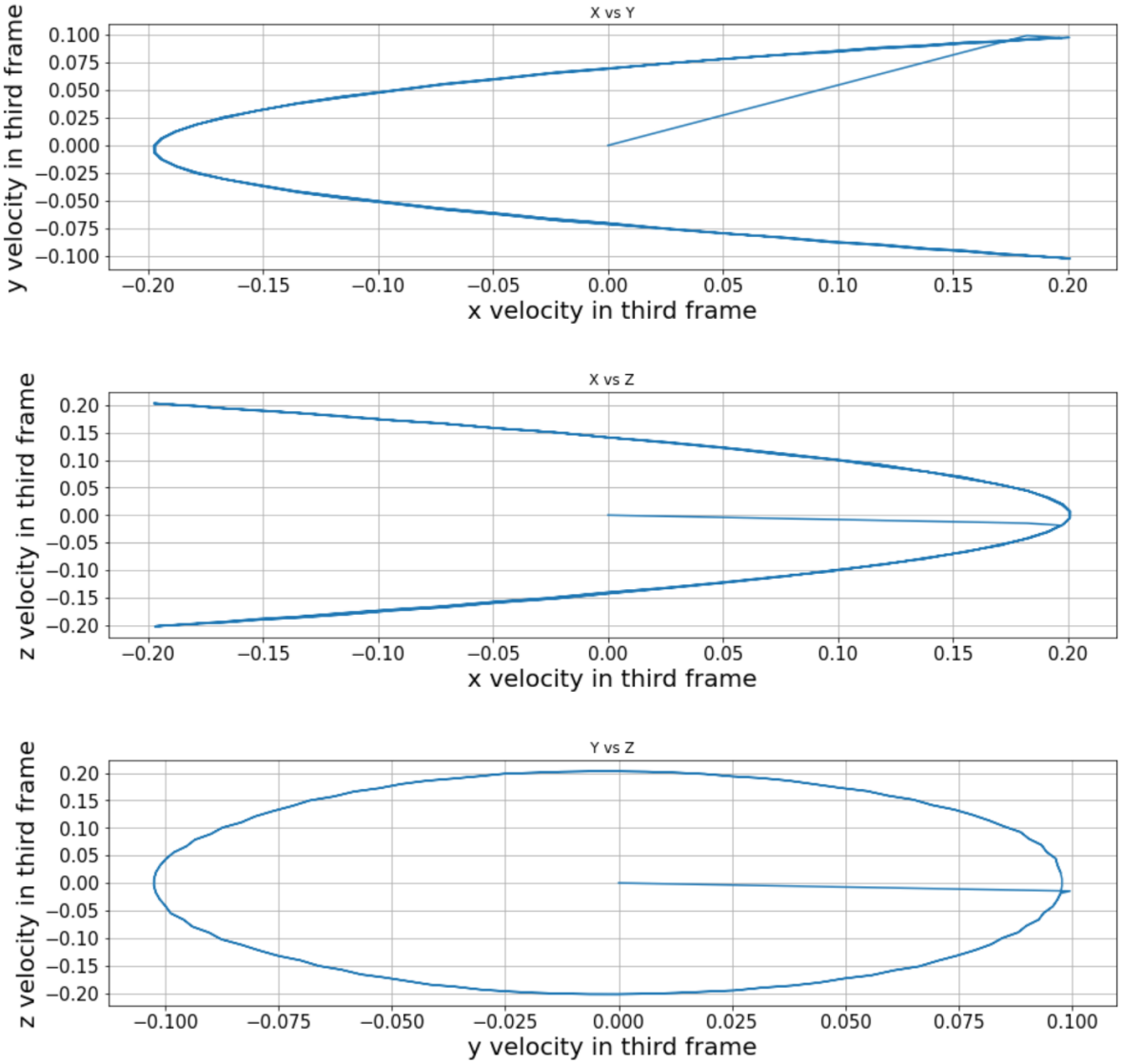

Figure 16: Analysis Result.

It seems the result in the third frame is most intuitive. Because the first two comparison results is parabola type and the last one is ellipse type. The result in spatial frame and end-effector frame are in odd type.