Chapter 5

Robot Kinematics

Robot kinematics is the study of motion in multi-degree of freedom robot links. For a robot manipulator or arm, it focuses on the geometric relationships between a robot's end-effectors and links and how they move to achieve a desired position and orientation for the end effector (e.g., gripper or tool).

The robot should be able to derive current end-effector position information from all joints' angles - a process known as forward kinematics. Conversely, for motion planning tasks that require the end-effector to reach desired new positions or follow trajectory, the robot must determine the joint angles (or other variables) needed to achieve that pose. This calculation is referred as inverse kinematics.

In addition to static positioning problems, when the end-effector is in motion, the relationship between end-effector velocity and joint rates must be analyzed too. A matrix called `Jacobian` is used to map velocities from joint space to Cartesian space. The Jacobian also relates joint torques to the resultant force and torque applied by the end-effector.

In this chapter, we only talk about forward and inverse kinematics of `6R` serial manipulator since most industrial robot manipulators fall into this category. `Hex7Bot` robot arm can sometimes be considered as a 6R manipulator if we treat its coupled links(a.k.a `parallelogram`) between link 2&3 separately. Particularly, two geometric inverse kinematic algorithms are introduced and have been implemented in our workspace. One is the popular `Pieper` solution, which applies for robot when last axes intersect. The second inverse kinematic is a modified Pieper solution especially for local close loop links robot manipulator such as `ABB120` robot. At last, for robot velocity control, we also drive the robot Cartesian in modified D-H frames for robot velocity control and use it to detect potential singular workspace.

5.1 Forward Kinematics

Forward kinematics uses specified joint parameters to compute the resulting position of the end-effector. To handle the complex geometry of a manipulator, we affix frames to various parts of the mechanism - as described in the previous chapter - and then describe the relationship between these frames. Once the link frames are defined and the corressponding link parameters are known, developing the kinematic equation is straightforward for serial manipulators: using the values of the link parameters, the individual link transformation matrices are computed. These matrices are then multiplied together to obtain the single transformation that relates frame N to the robot base frame 0:

$$ {}^{0}_{n}T = {}^{1}_{0}T\,{}^{2}_{1}T\,{}^{3}_{2}T \ldots {}^{n-1}_{n}T \tag{5.1.1} $$

If modified D-H frames are adopted, transformation matrix between adjacent frames

$$ {}^{i-1}_{i}T = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i \sin\alpha_{i-1} \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i \cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1.2} $$

For 6R robot arm, the forward kinematics will be

$$ {}^{0}_{6}T = {}^{1}_{0}T\,{}^{2}_{1}T\,{}^{3}_{2}T\,{}^{4}_{3}T\,{}^{5}_{4}T\,{}^{6}_{5}T \tag{5.1.3} $$

To get the robot TCP with respect to the base, according to the robot frame assignment, the TCP is simply a transition along the z-axis of frame 6 by d_6. Therefore the final position of the end-effector with respect to the robot base frame can be expressed as:

$$ {}^{0}_{tcp}T = {}^{0}_{6}T\,{}^{6}_{tcp}T $$

$$ = \begin{bmatrix} r11 & r12 & r13 & x \\ r21 & r22 & r23 & y \\ r31 & r32 & 333 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ = \begin{bmatrix} r11 & r12 & r13 & d_6 r13 + x \\ r21 & r22 & r23 & d_6 r23 + y \\ r31 & r32 & r33 & d_6 r33 + z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1.4} $$

Sample code for generic Forward kinematics:

```
#ifndef M_PI
    #define M_PI            3.14159265358979
#endif

#ifndef ROW
    #define ROW               4
#endif
#ifndef COLUMN
    #define COLUMN            4
#endif
typedef struct
{
    double a_i_1;
    double alpha_i_1;
    double d_i;
}  Mod_DH_PARAM;  //Modified DH Parameters


class BaseModel
{
public:
    BaseModel(int dof, const std::string& robot_name);
    ~BaseModel();
    virtual bool GetDHParameters() = 0;
    virtual bool GetJointLimits() = 0;

    virtual void FKine (const double jtdisp[], double pose[][4]);
    vector<vector<double>>
      FKineDH(const vector<vector<double>>& dh_parameters,
              const vector<double>& joint_angles);
    vector<vector<double>>
      FKineModDH(const vector<double>& joint_angles);
protected:
    vector<JOINTLIMIT> JointLimit_; //maximum 9 DOF
    vector<Mod_DH_PARAM> modDH_Param_;
   //6x3 : [0]: a; [1]:alpha; [2]:d
    vector<vector<double>> DH_Param_;
 };
```

```
// Function to create a homogeneous transformation matrix
// for a single joint
vector<vector<double>> vecmat::createHomoMatrixModDH(double a_i_1,
    double alpha_i_1, double d_i, double theta_i)
{
    vector<vector<double>> T(4, vector<double>(4, 0.0));

    T[0][0] = cos(theta_i);
    T[0][1] = -sin(theta_i);
    T[0][2] = 0;
    T[0][3] = a_i_1;

    T[1][0] = sin(theta_i) * cos(alpha_i_1);
    T[1][1] = cos(theta_i) * cos(alpha_i_1);
    T[1][2] = -sin(alpha_i_1);
    T[1][3] = -d_i * sin(alpha_i_1);

    T[2][0] = sin(theta_i) * sin(alpha_i_1);
    T[2][1] = cos(theta_i) * cos(alpha_i_1);
    T[2][2] = cos(alpha_i_1);
    T[2][3] = d_i * cos(alpha_i_1);

    T[3][0] = 0.0;
    T[3][1] = 0.0;
    T[3][2] = 0.0;
    T[3][3] = 1.0;

    return T;
}

 // Function to calculate forward kinematics
vector<vector<double>> BaseModel::FKineModDH
    (const vector<double>& joint_angles) {
    // Identity matrix
    vector<vector<double>> T_total = {{1,0,0,0},
    {0,1,0,0}, {0,0,1,0}, {0,0,0,1}};

    for (size_t i = 0; i < modDH_Param_.size(); ++i) {
        double a_i_1 = modDH_Param_[i].a_i_1;
        double alpha_i_1 = modDH_Param_[i].alpha_i_1;
        double d_i = modDH_Param_[i].d_i;
        double theta_i = joint_angles[i];

        vector<vector<double>> T_joint =
            vecmat::createHomoMatrixModDH(a_i_1, alpha_i_1, d_i, theta_i);
         T_total = vecmat::matrixMultiply(T_total, T_joint);
    }
    return T_total;
}
```

If we have already known all D-H parameters of a robot arm, such as for `Hex7bot` and its D-H frames arrangements are shown in figure 5.1:

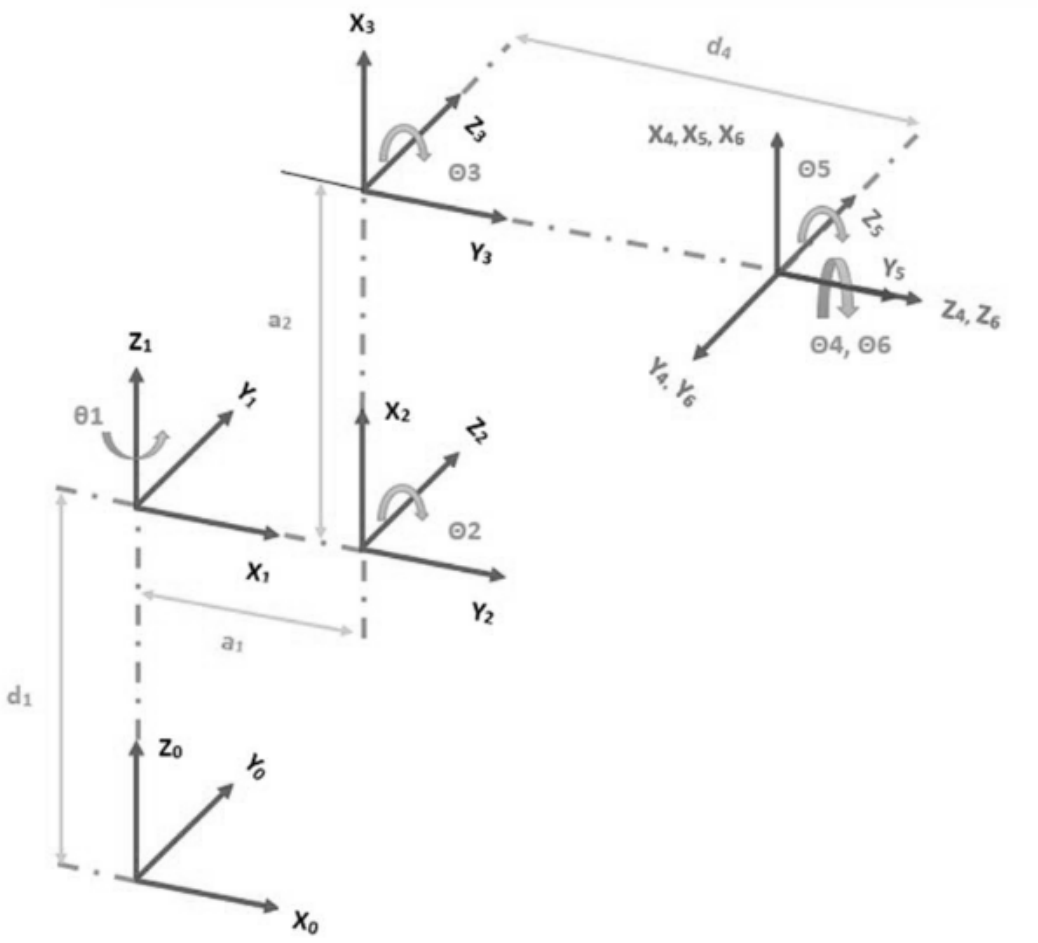

Figure 5.1: Hex7bot frames arrangement

And D-H parameters:

i	θ_i	α_{i-1}	$a_{i-1}(mm)$	$d_i(mm)$
1	θ_1	0	0	10.4
2	θ_2	-90	20.0	0
3	θ_3	0	120	0
4	θ_4	-90	30.0	160
5	θ_5	90	0	0
6	θ_6	-90	0	120

we can write its Forward Kinematics in a function:

```
class Hex7BotModel : public BaseModel;
void Hex7BotModel::FastFKine (const double theta[], double Pos[][4])
{
   double a[6], d_off[6];
   int base = 0;

  for (i = 0; i < MAX_7BOT_JNTS ; i++){
       a[i] = modDH_Param_[i].a_i_1;
       alpha[i] = modDH_Param_[i].alpha_i_1 ; //rad
       d_off[i] = modDH_Param_[i].d_i;
       theta[i] = DH_Para_[base + 3];
       base += COLUMN;
   }

   double c1 = cos(theta[0]);
   double c2 = cos(theta[1]);
   double c3 = cos(theta[2]);
   double c4 = cos(theta[3]);
   double c5 = cos(theta[4]);
   double c6 = cos(theta[5]);

   double s1 = sin(theta[0]);
   double s2 = sin(theta[1]);
   double s3 = sin(theta[2]) ;
   double s4 = sin(theta[3]);
   double s5 = sin(theta[4]);
   double s6 = sin(theta[5]);
```

```
double s23 = s2 * c3 + c2 * s3;
double c23 = c2 * c3 - s2 * s3;

double s4s5 = s4 * s5;
double s4c5 = s4 * c5;
double c4s5 = c4 * s5;
double c4c5 = c4 * c5;

double s5s6 = s5 * s6;
double c5c6 = c5 * c6;
double s5c6 = s5 * c6;

double s4s6 = s4 * s6;
double s4c6 = s4 * c6;
double c4s6 = c4 * s6;
double c4c6 = c4 * c6;

double c4c5c6 = c4c5 * c6;
double c4c5s6 = c4c5 * s6;
double s4c5c6 = s4c5 * c6;
double s4s5s6 = s4s5 * s6;
double s4c5s6 = s4c5 * s6;

double nx = c1 * c23 * (c4c5c6 - s4s6) - c1 *s23 * s5 * c6
        + s1 * (s4c5c6 + c4 * s6);
double ny = s1 * c23 * (c4c5c6 - s4s6) - s1 * s23 * s5 * c6
        - c1 * (s4c5c6 + c4 * s6);
double nz = -s23 * (c4c5c6 - s4s6) - c23 * s5c6 ;

double ox = - c1 * c23 * (c4c5s6 + s4c6) + c1 * s23 * s5s6
        - s1 * (s4c5s6 - c4 * c6);
double oy = - s1 * c23 * (c4c5s6 + s4c6) + s1 * s23 * s5s6
        + c1 * (s4c5s6 - c4 * c6);
double oz = s23 * (c4c5s6 + s4c6) + c23 * s5s6;

double ax = - c1 * c23 * c4s5 - c1 * s23 * c5 - s1 * s4s5;
double ay = - s1 * c23 * c4s5 - s1 * s23 * c5 + c1 * s4s5;
double az = s23 * c4s5 - c23 * c5;
```

```
    // Please note: Although modified D-H frames is implemented, link offset
    // axis 6 measured along Z6 d6is not in T65 homogeneous transform.
    // That is for last three frames (three consecutive axes of the robot
    // robot intersect at a common point(Pieper solution). This arrangement
    // can decouple joint 1 and 4 during Inverse kinematics computation:
    // i.e., Py /Px = tangent (joint 1 angle).
    double Px = -d_off[3] * c1 * s23 + c1 * (c2 * a[2] + a[1])
                + a[3] * c1 * c23;
    double Py = -d_off[3] * s1 * s23 + s1 * (c2 * a[2] + a[1])
                + a[3] * s1 * c23;
    double Pz = -s2 * a[2] + d_off[0] - d_off[3] * c23 - a[3] * s23;

    cout << "\t" << "Wrist center (Px, Py, Pz)" <<setw(12) <<
            Px << setw(12) << Py << setw(12) << Pz << endl;

    // To get the pose of  the tip (TCP) with respect to he robot base,
    // it is simply a transition along the z-axis of frame{6} by d6.
    //The final position  can be expressed as:
    // p_tip = T60 * P6 = [  nx ox ax Px
    //                       ny oy ax Py
    //                       nz oz ax Pz
    //                       0  0   0  1]   * [ 0 0 d6 1]'
    //     = [ Px + ax * d6, Py + ay * d6 , Pz + az * d6 1]'
    double px = Px + ax * d_off[5];
    double py = Py + ay * d_off[5];
    double pz = Pz + az * d_off[5] ;

    cout << "\t" << "End Effector (px, py, pz)" <<setw(12)
            << px << setw(12) << py << setw(12) << pz << std::endl;

    Pos[0][0] = nx; Pos[1][0] = ny; Pos[2][0] = nz;
    Pos[0][1] = ox; Pos[1][1] = oy; Pos[2][1] = oz;
    Pos[0][2] = ax; Pos[1][2] = ay; Pos[2][2] = az;
    Pos[0][3] = px; Pos[1][3] = py; Pos[2][3] = pz;
    Pos[3][0] = 0;  Pos[3][1] = 0;  Pos[3][2] = 0; Pos[3][3] = 1;
}
```

5.2 Inverse Kinematics

Inverse kinematics takes a desired end-effector location and computes the associated joint angles. For serial manipulators this requires the solution of a set of polynomials derived from the kinematics equations, resulting in multiple possible configurations for the chain. In he case of a general 6R serial manipulator, there are sixteen different inverse kinematics solutions, corresponding to the roots of a sixteenth-degree polynomial.

If the 6R robot manipulator's last three axes intersect at a common point, i.e., the origins of link frames {4}, {5}, and {6} are all located at this point of intersection, we can get inverse kinematics algebraic solution (Pieper method).

5.2.1 Inverse Kinematics In Classic DH Frames

If we use classic DH to establish robot frames:

$$ {}^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2.1} $$

For last three axes intersect, the position and orientation of the wrist expressed in terms of the classic homogeneous transformation is defined as following:

$$ {}^0_wT_{ORG} = {}^0_3T = {}^0_1T^1_2T^2_3T \tag{5.2.2} $$

The last three axes (Z_3, Z_4,Z_5) of frame 3, 4, 5 intersect at wrist center.

$$ {}^0_5T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{22} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^0_6T \cdot {}^6_5T = {}^0_6T^5_6T^{-1} \tag{5.2.3} $$

Where

$$ {}^0_6T = \begin{bmatrix} l_x & m_x & n_x & q_x \\ l_y & m_y & n_y & q_y \\ l_z & m_z & n_z & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

is the goal position and orientation that to be accomplished. And ${}^5_6T^{-1}$ is inverse homogeneous transformation matrix, i.e.,

$$ {}^{i-1}_iT^{-1} = \begin{bmatrix} \cos\theta_i & \sin\theta_i & 0 & -a_i \\ -\sin\theta_i\cos\alpha_i & \cos\theta_i\cos_i\alpha_i & \sin\alpha_i & -d_i\sin\alpha_i \\ \sin\theta_i\sin\alpha_i & -\cos\theta_i\sin\alpha_i & \cos\alpha_i & -d_i\cos\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2.4} $$

The wrist center position $[p_x, p_y, p_z]$ the fourth column of left transformation matrix can be derive from the end-effector, i.e.,

$$ p_x = -l_xa_6 - ud_6 + q_x $$

$$ p_y = -l_ya_6 - vd_6 + q_y $$

$$ p_z = -l_za_6 - wd_6 + q_z $$

Where

$$ u = m_x\sin\alpha_6 + n_x\cos\alpha_6 $$

$$ v = m_y\sin\alpha_6 + n_y\cos\alpha_6 $$

$$ w = m_z\sin\alpha_6 + n_z\cos\alpha_6 $$

We can see that the wrist position is a function of θ_1, θ_2 and θ_3. There are algebraic solutions to most 6R serial robot with the parameter $a_2 = 0$, or $\alpha_2 = 0$. For example, for $a_2 = 0$ we have

$$p_x = (c_1c_2 - c\alpha_1 s_1 s_2)a_3c_3 + [-c_1c\alpha_2 s_2 - c\alpha_1 s_1 s\alpha_2 s_2 + s\alpha_1 s_1 s\alpha_2]a_3 s_3$$
$$+[c_1 s\alpha_2 s_2 + c\alpha_1 s_1 s\alpha_2 c_2 + s\alpha_1 s_1 c\alpha_2]d_3$$

$$+(s\alpha_1 s_1 d_2 + a_1 c_1) \tag{5.2.5}$$

$$p_y = (s_1c_2 + c\alpha_1 c_1 s_2)a_3c_3 + [-s_1c\alpha_2 s_2 + c\alpha_1 c_1 c\alpha_2 c_2 - s\alpha_1 c_1 s\alpha_2]a_3 s_3$$
$$+[s_1 s\alpha_2 s_2 - c\alpha_1 c_1 s\alpha_2 c_2 - s\alpha_1 c_1 c\alpha_2]d_3$$
$$+(-s\alpha_1 c_1 d_2 + a_1 s_1) \tag{5.2.6}$$

$$p_z = s\alpha_1 s2a_3c_3 + [s\alpha_1 c\alpha_2 c_2 + c\alpha_1 s\alpha_2]a_3 s_3$$
$$+[-s\alpha_1 s\alpha_2 c_2 + c\alpha_1 c\alpha_2]d_3$$
$$+(c\alpha_1 d_2 + d_1) \tag{5.2.7}$$

we have

$$p_x c_1 + p_y s_1 = c_2 a_3 c_3 - c\alpha_2 s_2 a_3 s_3 + s\alpha_2 s_2 d_3 + s\alpha_2 s_2 d_3 + a_1; \tag{5.2.8}$$

To the subcase: $\alpha_2 = 0$, using $\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$. the above equation becomes:

$$p_x c_1 + p_y s_1 = c_{23} a_3 + a_1 \tag{5.2.9}$$

$$p_z = s_{23} a_3 s\alpha_1 + c\alpha_1 c\alpha_2 d_3 + c\alpha_1 d_2 + d_1 \tag{5.2.10}$$

using the equation

$$c_{23} = \pm\sqrt{1 - {s_{23}}^2} = \pm\sqrt{1 - (k/s\alpha_1)^2}$$

where

$$k = p_z - c\alpha_1 c\alpha_2 d_3 - c\alpha_1 d_2 - d_1 \tag{5.2.11}$$

Solving trigonometric equation of the form

$$a\sin x + b\cos x = c$$

as long as $a^2 + b^2 = c^2$

$$a\sin x + b\cos x = \sqrt{a^2+b^2}(\cos\beta sinx + \sin\beta\cos x) = \sqrt{a^2+b^2}\sin(x+\beta)$$

Have resolving θ_1, we may resolve it for θ_2 and θ_3. Notice that 2 possible θ_1 values, we may have 4 solutions for θ_2 and θ_3. Similarly, we can solve θ_4 θ_5, θ_6 by

$${}^3_4T = {}^0_3T^{-1} \cdot {}^0_6T\,{}^4_6T^{-1}$$

Next, we need implement it with computer programming language for real-time control, such as:

```
/**********************************************************************
function description:  solve equation:  k1*sin(theta)+k2*cos(theta)=const;

   parameters: sincoeff: pointer to k1;
      coscoeff: pointer to k2;
          cons: pointer to const;
      solution: pointer to array recording the solutions,
                their values are limited within [-PI, PI];
   return value: 0: no solution exists;
              1: one solution, this happens only when the sum of theta
                 and atan(k2,k1) is equal to PI;
              2: two solutions;
**********************************************************************/
int BaseModel::solve(double *sincoef, double *coscoef,
            double *cons, double *solution)
{
    double temp1, temp2, angletemp, sumtemp;
    temp1 = (*sincoef) * (*sincoef) + (*coscoef) * (*coscoef);
    temp1 = sqrt(temp1);
    temp1 = (*cons) / temp1;
    temp2 = fabs(temp1) - 1.0;

    if (temp2 > CALC_TOLERANCE) {
            return(0); /* no solution */
    }
    else {
        angletemp = atan2(*coscoef, *sincoef);
        if (temp2 > (-CALC_TOLERANCE)) {
            if (temp1 > 0.0){
                sumtemp = M_PI / 2;
            }
            else{
                sumtemp = -M_PI / 2;
            }

            solution[0] = sumtemp - angletemp;
            if (solution[0] > M_PI){
                solution[0] = solution[0] - 2 * M_PI;
            }
            else{
                if (solution[0] < -M_PI){
                    solution[0] = solution[0] + 2 * M_PI;
                }
            }
            return(1);
        }
```

```
        else {
            sumtemp = asin(temp1);
            solution[0] = sumtemp - angletemp;
            solution[1] = M_PI - sumtemp - angletemp;
        }

        if (solution[0] > M_PI){
            solution[0] = solution[0] - 2 * M_PI;
        }
        else{
            if (solution[0] < -M_PI){
                solution[0] = solution[0] + 2 * M_PI;
            }
        }

        if (solution[1] > M_PI){
            solution[1] = solution[1] - 2 * M_PI;
        }
        else{
            if (solution[1] < -M_PI){
                solution[1] = solution[1] + 2 * M_PI;
            }
        }

        if (fabs(solution[0] - solution[1]) < CALC_TOLERANCE){
            return(1);
        }

        if ((fabs(solution[0] - M_PI) < CALC_TOLERANCE) &&
            (fabs(solution[1] - M_PI) < CALC_TOLERANCE)) {
            return(1);
        }
        else {
            return(2);
        }
    }
    return 0; // no solution find
}
```

Since it is a generic analytic solution, we will integrate it into robot model base class. The follow code snippet shows how to solve for joint 1 and joint2 first.

```
//The auxiliary structure recording the intermediate results
typedef struct
{
    double angle;
    double sine;
    double cosine;
    double h1;
    double h2;
    double h3;
} AUX_RECORD;

typedef struct
{
    double angle;
    int    pointer;
} JT_ANGLE_LINK;

/*************************************************************************
 Function: solve inverse kinematics for robots whose
 last three joints intersect.

  Parameter: dhpara: pointer to the matrix of D-H parameters;
          pose: pointer to the pose matrix;
      jtcurrnt: pointer to the array recording the current joint angle;
         jtnew: pointer to the array recording the required joint
                    relative displacement;
       jtlimit: pointer to the array recording joint limits;

   return value: 0: no solution;
                 1: solution exist;
*************************************************************************/
int BaseModel::Pieper_IFK(const double dhpara[], const double pos[][4],
                          const double jtcurrnt[], double jtnew[],
                             const JOINTLIMIT jtlimit[])
```

- Pierper_IFK function body:

```
{
 int     i, j, k, jt_num1, jt_num2, jt_num3, jt_num4, jt_num5, jt_num6;
 int     base, itemp, ptr;
 int     solution, solunum;

 double a1, a2, a3, a4, a5, a6, d1, d2, d3, d4, d5, d6;
 double miu1, miu2, miu3, miu4, miu5, miu6;
 double lamda1, lamda2, lamda3, lamda4, lamda5, lamda6;
 double tht1, tht2, tht3, tht4, tht5, tht6;
 double alpha, atemp, angletemp, temp0, temp1, temp2, temp3,
        temp4, temp5, temp6;
 double sumtemp, diffsum;

 double lx, ly, lz, mx, my, mz, nx, ny, nz, qx, qy, qz;
 double f1, f2, f3, f4, n1, n2, n3, h1, h2, h3, m1, m2, m3, det;
 double u, v, w, p, q, r;
 double c1, s1, c2, s2, c3, s3, c4, s4, c5, s5, c6, s6;
 double deno, num;
 double aux[2], final_val;

 double cos_coef, sin_coef, cons, cons1, cons2;
 double rtemp[2];
 double mtemp1[3][4], mtemp2[3][4], ma1[3][4], ma2[3][4], ma3[3][4],
            ma4[3][4], ma5[3][4], ma6[3][4];

 AUX_RECORD theta1[8];
 JT_ANGLE_LINK theta2[8], theta3[8], theta5[16], theta4[32];
```

- D-H parameters initialization:

```
/* Step 0:  parameter initialization: parameter input &
        auxiliary value setup */
base = 0;   a1 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu1 = sin(alpha); lamda1 = cos(alpha);
d1 = dhpara[base + 2];   tht1 = dhpara[base + 3];

base += COLUMN;  a2 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu2 = sin(alpha);  lamda2 = cos(alpha);
d2 = dhpara[base + 2];  tht2 = dhpara[base + 3];

base += COLUMN;  a3 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu3 = sin(alpha); lamda3 = cos(alpha);
d3 = dhpara[base + 2];  tht3 = dhpara[base + 3];

base += COLUMN;  a4 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu4 = sin(alpha); lamda4 = cos(alpha);
d4 = dhpara[base + 2]; tht4 = dhpara[base + 3];

base += COLUMN;  a5 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu5 = sin(alpha); lamda5 = cos(alpha);
d5 = dhpara[base + 2]; tht5 = dhpara[base + 3];

base += COLUMN; a6 = dhpara[base];
alpha = dhpara[base + 1] * M_PI / 180;
miu6 = sin(alpha); lamda6 = cos(alpha);
d6 = dhpara[base + 2]; tht6 = dhpara[base + 3];
```

- Solve first joint:

```
lx = pos[0][0];    mx = pos[0][1];    nx = pos[0][2];    qx = pos[0][3];
ly = pos[1][0];    my = pos[1][1];    ny = pos[1][2];    qy = pos[1][3];
lz = pos[2][0];    mz = pos[2][1];    nz = pos[2][2];    qz = pos[2][3];

temp0 = lamda2 * d2 + d3 + lamda3 * d4;
temp1 = -lamda3 * d3 * d4 + (d1 * d1 - a1 * a1 + a2 * a2 + d2 * d2 -
            a3 * a3 - d3 * d3 - d4 * d4) / 2;
temp2 = temp1 + a1 * a1 + lamda1 * d1 * d2;
temp3 = temp0 + lamda1 * lamda2 * d1;
temp4 = -d4 * d4 * miu3 * miu3 - a3 * a3;
temp5 = miu4 * lamda5;
temp6 = lamda4 * lamda5;

u = mx * miu6 + nx * lamda6;
v = my * miu6 + ny * lamda6;
w = mz * miu6 + nz * lamda6;

p = -lx * a6 - u * d6 + qx;
q = -ly * a6 - v * d6 + qy;
r = -lz * a6 - w * d6 + qz;

/* Step 1:  solve the joint displacement for the first and second joint */
/* Step 1.1: solve for the case miu2!=0  a2==0 */
/* Step 1.1.1: solve the displacement for joint 1 & 2 */
if ((fabs(miu2) > CALC_TOLERANCE) && (fabs(a2) > CALC_TOLERANCE)) {
    fprintf(stderr,"Unsolvable situation for this method.\n");
    return 0;
}

if ((miu2 != 0.0) && (fabs(a2) <= CALC_TOLERANCE)) {
   /* start if miu2!=0 and a2=0 */
   std::cout << "\t"<< "miu2 not zero, a2 zero " << std::endl;
   cos_coef = d2 * q * miu1 - a1 * p;
   sin_coef = -a1 * q - d2 * miu1 * p;

   cons = temp2 + (p * p + q * q + r * (r - 2 * d1))/2 - d2 * lamda1 * r;
   cons = -cons;
```

```
itemp=solve(&(sin_coef), &(cos_coef), &(cons), rtemp);
i=0;   jt_num1=0;
while (i <= itemp - 1){
    solunum = bound_check(rtemp[i], jtlimit[0].upperlmt,
            jtlimit[0].lowerlmt, aux);
    if (solunum != 0){
       j=0;
       while (j<solunum) {
            theta1[jt_num1].angle=aux[j];
            jt_num1++;
            j++;
       }
   }
   i++;
 }

if (jt_num1 == 0) return 0;
```

- Solve second joint:

```
for(i = 0; i <= jt_num1-1; i++) {
   cout << "\t" << "the joint 1 displacement:(deg)"
             << theta1[i].angle / M_PI * 180  << std::endl;
}

i=0; jt_num2=0;
while (i <= (jt_num1-1)){
   /* start while 2 */
   c1 = cos(theta1[i].angle);
   s1 = sin(theta1[i].angle);

   h1 = c1*p+s1*q-a1;
   atemp = s1*p-c1*q;
   h2 = -lamda1*atemp;
   h3 = miu1*atemp;
   atemp = r-d1;
   h2 += miu1*atemp;
   h2 = -h2;
   h3 += lamda1*atemp;

   f1 = -lamda2*h3+temp0;
   f1 = f1/miu2;

   itemp = solve(&(h1), &(h2), &(f1), rtemp);

   if (itemp != 0) {
      solunum = bound_check(rtemp[0], jtlimit[1].upperlmt,
                jtlimit[1].lowerlmt, aux);
      if (solunum != 0) {
         j=0;
         while (j < solunum){
             theta2[jt_num2].angle = aux[j];
             theta2[jt_num2].pointer = i;
             jt_num2 ++;
             j ++;
         }
         theta1[i].cosine = c1;
         theta1[i].sine = s1;
         theta1[i].h1 = h1;
         theta1[i].h2 = -h2;
         theta1[i].h3 = h3;
      }
   }
```

```
        if (itemp>1) {
           solunum = bound_check(rtemp[1], jtlimit[1].upperlmt,
                                 jtlimit[1].lowerlmt, aux);
           if (solunum != 0) {
              j=0;
              while (j < solunum)
              {
                  theta2[jt_num2].angle = aux[j];
                  theta2[jt_num2].pointer = i;
                  jt_num2++;
                  j++;
              }
              theta1[i].cosine = c1;
              theta1[i].sine = s1;
              theta1[i].h1 = h1;
              theta1[i].h2 = -h2;
              theta1[i].h3 = h3;
           }
        }
        i++;
     }/* end while 2 */
     if (jt_num2==0) return 0;

  }/* end if miu2!=0 and a2=0 */
  ...
}
```

Once all possible joint solutins have been computed, the solution with the least joint displacement is typically selected for real-time control. Additionally, to manage accumulated calculation errors, it is necessary to verify each candidate solution by substituting it into the forward kinematics equations. This verification ensures that the resulting end-effector pose(position and orientaion) falls within the specified controlled rangs, such as

```
// Step 5: solve the joint displacement for the sixth joint
// it is possible to insert the joint angle selection function here as the following
//   calculation is rather computation intensive.
i = 0;
jt_num6 = 0;
solution = 0;
diffsum = 1000;
while (i <= jt_num4 - 1){
 /* start while for step 5 */
 tht4 = theta4[i].angle;
 c4 = cos(tht4);
 s4 = sin(tht4);

 ma4[0][0] = c4;  ma4[0][1] = -s4 * lamda4;  ma4[0][2] = s4 * miu4;
 ma4[1][0] = s4;  ma4[1][1] = c4 * lamda4;   ma4[1][2] = -c4 * miu4;
 ma4[2][0] = 0;   ma4[2][1] = miu4;          ma4[2][2] = lamda4;
 ma4[0][3] = a4 * c4; ma4[1][3] = a4 * s4;   ma4[2][3] = d4;

 base = theta4[i].pointer;
 tht5 = theta5[base].angle;
 c5 = cos(tht5);
 s5 = sin(tht5);

 ma5[0][0] = c5;  ma5[0][1] = -s5 * lamda5;  ma5[0][2] = s5 * miu5;
 ma5[1][0] = s5;  ma5[1][1] = c5 * lamda5;   ma5[1][2] = -c5 * miu5;
 ma5[2][0] = 0;   ma5[2][1] = miu5;          ma5[2][2] = lamda5;
 ma5[0][3] = a5 * c5;  ma5[1][3] = a5 * s5;  ma5[2][3] = d5;

 base = theta5[base].pointer;
 tht3 = theta3[base].angle;
 c3 = cos(tht3);
 s3 = sin(tht3);

 ma3[0][0] = c3;  ma3[0][1] = -s3 * lamda3;  ma3[0][2] = s3 * miu3;
 ma3[1][0] = s3;  ma3[1][1] = c3 * lamda3;   ma3[1][2] = -c3 * miu3;
 ma3[2][0] = 0;   ma3[2][1] = miu3;          ma3[2][2] = lamda3;
 ma3[0][3] = a3 * c3;  ma3[1][3] = a3 * s3;  ma3[2][3] = d3;

 base = theta3[base].pointer;
 tht2 = theta2[base].angle;
 c2 = cos(tht2);
 s2 = sin(tht2);
```

```
ma2[0][0] = c2;  ma2[0][1] = -s2 * lamda2;  ma2[0][2] = s2 * miu2;
ma2[1][0] = s2;  ma2[1][1]= c2 * lamda2;    ma2[1][2] = -c2 * miu2;
ma2[2][0] = 0;   ma2[2][1]= miu2;           ma2[2][2] = lamda2;
ma2[0][3] = a2*c2;   ma2[1][3] = a2*s2;     ma2[2][3] = d2;

base = theta2[base].pointer;
tht1 = theta1[base].angle;
c1 = cos(tht1);
s1 = sin(tht1);

ma1[0][0] = c1;  ma1[0][1] = -s1 * lamda1; ma1[0][2] = s1 * miu1;
ma1[1][0] = s1;  ma1[1][1] = c1 * lamda1;  ma1[1][2] = -c1 * miu1;
ma1[2][0] = 0;   ma1[2][1] = miu1;         ma1[2][2] = lamda1;
ma1[0][3] = a1 * c1;  ma1[1][3] = a1 * s1; ma1[2][3] = d1;
```

- Calculate ${}^5_6T = (T_1T_2T_3T_4T_5)^{-1} \cdot {}^0_6T_{pos}$, And solve joint 6 with all possible solutions.

```
HMul(ma1, ma2, mtemp1);
HMul(mtemp1, ma3, mtemp2);
HMul(mtemp2, ma4, mtemp1);
HMul(mtemp1, ma5, mtemp2);
HInv(mtemp2, mtemp1);
HMul(mtemp1, pos, mtemp2);

c6 = mtemp2[0][0];
s6 = mtemp2[1][0];

if (fabs(c6 * c6 + s6 * s6 - 1.0) <= CALC_TOLERANCE)
{
  jt_num6++;
  tht6 = atan2(s6, c6);

  ma6[0][0] = c6; ma6[0][1] = -s6*lamda6; ma6[0][2] = s6*miu6;
  ma6[1][0] = s6; ma6[1][1] = c6*lamda6;  ma6[1][2] = -c6*miu6;
  ma6[2][0] = 0;  ma6[2][1]=  miu6;       ma6[2][2] = lamda6;
  ma6[0][3] = a6 * c6;  ma6[1][3] = a6 * s6;  ma6[2][3] = d6;
  if(HComp(ma6, mtemp2, 10 * CALC_TOLERANCE) == 1){
    solunum = bound_check(tht6, jtlimit[5].upperlmt, jtlimit[5].lowerlmt, aux);
    if (solunum != 0) {
      j=0;
      while (j < solunum) {
        sumtemp = fabs(tht1 - jtcurrnt[0]) + fabs(tht2 - jtcurrnt[1]) +
                  fabs(tht3 - jtcurrnt[2]) + fabs(tht4 - jtcurrnt[3]) +
                  fabs(tht5 - jtcurrnt[4]) + fabs(aux[j] - jtcurrnt[5]);
        if (sumtemp <= diffsum)
        {
          solution = i;
          diffsum = sumtemp;
          final_val = aux[j] - jtcurrnt[5];
        }
        j++;
      }
    }
  }
}
i++;
}/* end while for step 5 */
```

- Verify all possible solution based on joint displacement

```
    //print out all joint displacement since last calculation.
    if (diffsum<1000){
        cout << "\n" << "________solution exists_______="
                  << solution <<"__________" << std::endl;
        jtnew[3] = theta4[solution].angle - jtcurrnt[3];
        base = theta4[solution].pointer;
        jtnew[4] = theta5[base].angle - jtcurrnt[4];
        base = theta5[base].pointer;
        jtnew[2] = theta3[base].angle - jtcurrnt[2];
        base = theta3[base].pointer;
        jtnew[1] = theta2[base].angle - jtcurrnt[1];
        base = theta2[base].pointer;
        jtnew[0] = theta1[base].angle - jtcurrnt[0];
        jtnew[5] = final_val;

        for (int tempi = 0; tempi < 6; tempi ++)
        {
            std::cout << "Joint[" << tempi << "]=" << jtnew[tempi] / M_PI * 180
            <<" degree" << std::endl;
        }
        return(1);
    }
    else
    {
        return(-1);
    }
    } // endof Piper_IFK
```

5.2.2 Inverse Kinematics In Modified DH Frames

A lot of books and popular resources are talking about the `Pieper` method in modified DH frames. Since in modified DH frames, the coordinates of frames O_{i-1} is put on joint $i-1$, not the axis i in the classic DH convention. The coordinates of O_i are put on the axis i, not the axis $i+1$ in the classic DH convention. The wrist center become:

$$^0P_{4ORG} = {}^0_1T\,{}^1_2T\,{}^2_3T\,^3P_{4ORG} \tag{5.2.12}$$

and

$$^0P_{4ORG} = {}^0_1T\,{}^1_2T\,{}^2_3T \begin{bmatrix} a_3 \\ -d_4 \sin\alpha_3 \\ d_4 \cos\alpha_3 \\ 1 \end{bmatrix} \tag{5.2.13}$$

Using Modified DH transformation matrix ${}^{i-1}_iT$ on ${}^2_3T$ first, we have:

$$^0P_{4ORG} = {}^0_1T\,{}^1_2T \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ 1 \end{bmatrix} \tag{5.2.14}$$

where

$$f_1(\theta_3) = a_3c_3 + d_4s\alpha_3s_3 + a_2,$$
$$f_2(\theta_3) = a_3c\alpha_2s_3 - d_4s\alpha_3c\alpha_2c_3 - d_4s\alpha_2c\alpha_3 - d_3s\alpha_2,$$
$$f_3(\theta_3) = a_3s\alpha_2s_3 - d_4s\alpha_3s\alpha_2c_3 + d_4c\alpha_2c\alpha_3 + d_3c\alpha_2,$$

Using Modified DH transformation matrix on ${}^0_1T$ and ${}^1_2T$, we obtained

$${}^0P_{4ORG} = \begin{bmatrix} c_1g_1 - s_1g_2 \\ s_1g_1 + c_1g_2 \\ g_3 \\ 1 \end{bmatrix} \tag{5.2.15}$$

where

$$g_1 = c_2f_1 - s_2f_2 + a_1,$$
$$g_2 = s_2c\alpha_1f_1 + c_2c\alpha_1f_2 - s\alpha_1f_3 - d_2s\alpha_1,$$
$$g_3 = s_2s\alpha_1f_1 + c_2s\alpha_1f_2 + c\alpha_1f_3 + d_2c\alpha_1,$$

We can easily derive that magnitude squared of ${}^0P_{4ORG}$ is:

$$r = g_1^2 + g_2^2 + g_3^2 \tag{5.2.16}$$

We now rewrite this equation, along with the Z component equation as a system of two equations in the form

$$r = (k_1c_2 + k_2s_2)2a_1 + k_3, \tag{5.2.17}$$

$$z = (k_1s_2 - k_2c_2)s\alpha_1 + k_4, \tag{5.2.18}$$

where

$$k_1 = f_1 \tag{5.2.19}$$
$$k_2 = -f_2, \tag{5.2.20}$$
$$k_3 = f_1^2 + f_2^2 + f_3^2 + a_1^2 + d_2^2 + 2d_2f_3, \tag{5.2.21}$$
$$k_4 = f_3c\alpha_1 + d_2c\alpha_1 \tag{5.2.22}$$

Now, let us consider the solution of (5.2.12) for θ_3. We distinguish three cases:

1. If $a_1 = 0$ then we have $r = k_3$ where r is known. The right-hand side $k3$ is a function of θ_3 only. We can solve quadratic equation: such as $a\sin\theta_3 + b\cos\theta_3 = c$ and satisfy $a^2 + b^2 = c^2$.
2. If the manipulator has the link structure: $s\alpha_1 = 0$, then we have $z = k_4$, where z is known, we can solve θ_3 with quadratic equation
3. Otherwise, eliminate s_2 and c_2 to obtain

$$\frac{(r - k_3)^2}{4a_i^2} + \frac{(z - k_4)^2}{s^2\alpha_1} = k_1^2 + k_2^2. \tag{5.2.23}$$

having solved for θ_3 first, then solve for θ_2 and θ_1 etc.

5.3 Hex7Bot New DH Frames

Hex7bot was evolved from the 7bot robot arm, which mimics ABB IRB2400 industry robot-arm. The joint 2 and joint 3 of the Hex7bot have coaxial motors but opposite direction. Meanwhile the servo on joint3 drives parallelogram link with a pivot under the forearm, which is considered to be origin of frame 3.

In addition, because all servo motors of the Hex7Bot arm are rotated clockwise for positive positions, and the joint 2 and joint 3 rotate along opposite direction. We have to consider real joint control direction and their zero positions.

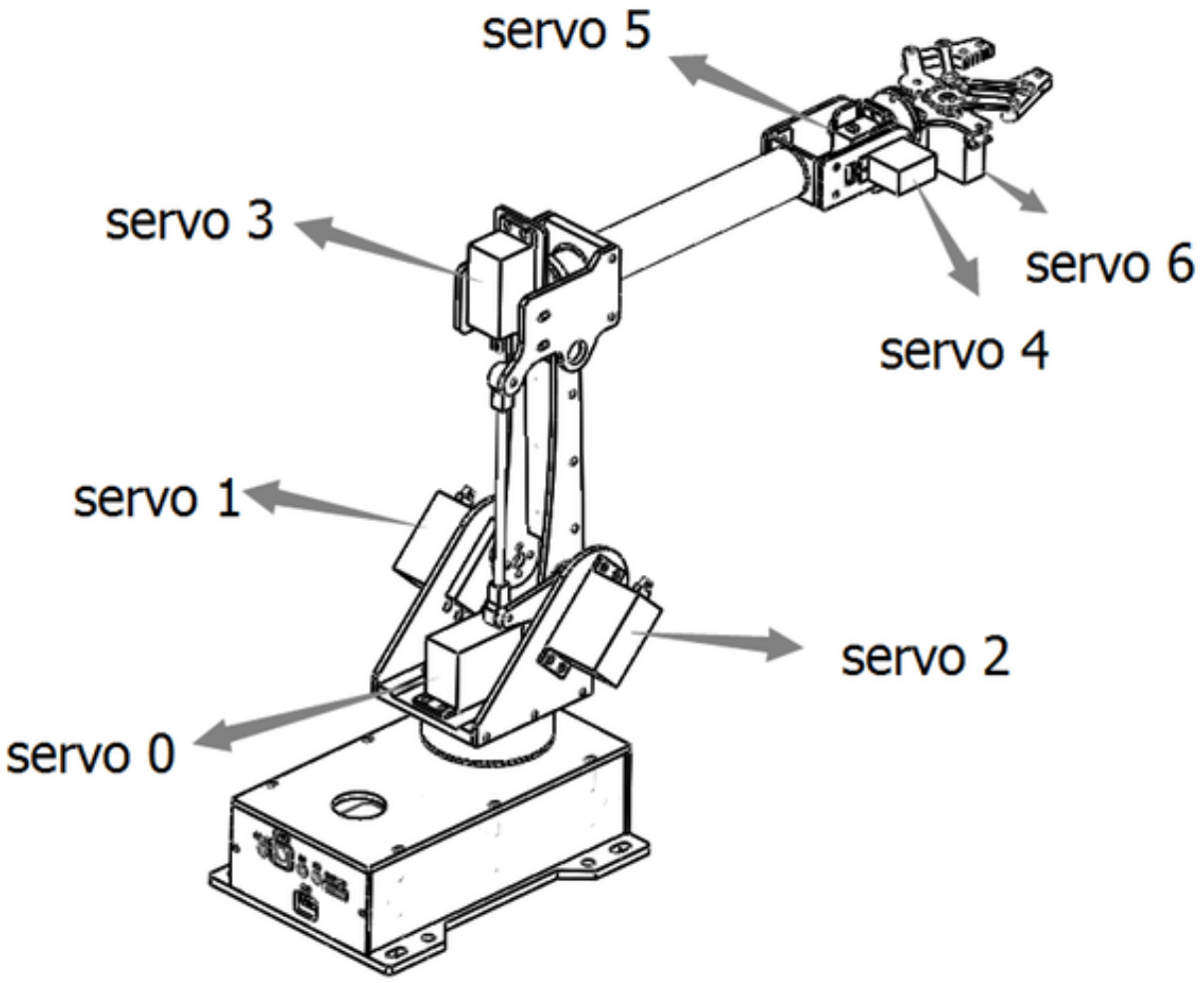

Figure 5.2: 7 DOF Hex7bot

If we attach base and all joint frames shown as Figure 5.2 and 5.3, we can get all links' D-H parameters.

Note:

- d_4 is the distance from actually origin of frame 4, 5, 6 to virtual joint3(pivot) distance along z_4.

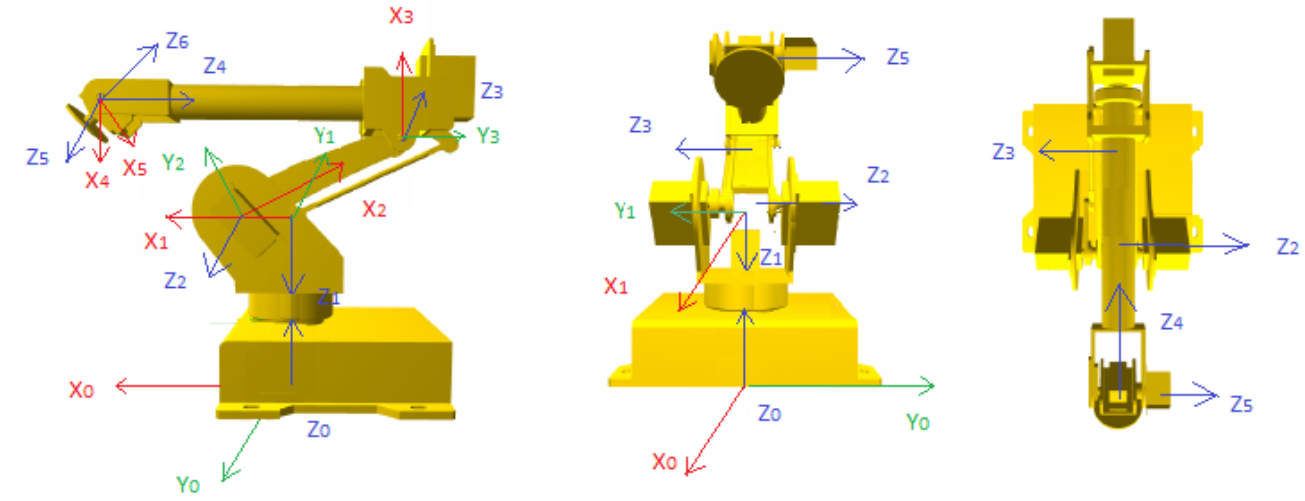

Figure 5.3: Hex7bot Base and joint 1-3 frames

- d_6 is the distance from original of frame 4, 5, 6 to end-effector (TCP).

If d_6 is not zero, that is also to say that if the frame 4, 5 and 6 are not intersecting, there are no any algebraic solutions even with the geometric method. So it is not practical using generic Pieper method. Next we will explain how to derive this type robot inverse kinematics.

The new D-H parameters can be:

i	θ_i	α_{i-1}	$a_{i-1}(mm)$	$d_i(mm)$
1	θ_1	180	0	10.4
2	$\theta_2 - 90^0$	90	20.0	0
3	$\theta_3 - 30^0$	180	120	0
4	θ_4	-90	30.0	-160
5	θ_5	90	0	0
6	θ_6	-90	0	-120

Note α_2 is changed from 0 to 180. θ_2 and θ_3 are real joint control variables. -90^0 and -30^0 are initial values (offset) expressed in DH frames when Joint2 and Joint3 are at 'zero' positions.

5.4 Hex7Bot Inverse Kinematics

It is also a little bit challenging and computation intensive to derive algebraic inverse kinematics of hex7bot. However, if combining with the geometric method, especially your customized robot arm characteristics, you may find the solution efficiently.

Let's first derive a forward kinematics with the new D-H frames arrangement in the above table by substituting the link parameters into the general homogeneous transform:

$$ {}^0_6T = {}^0_1T {}^1_2T {}^2_3T {}^3_4T {}^4_5T {}^5_6T \tag{5.4.1} $$

$$ {}^{0}_{6}T = {}^{0}_{3}T\,{}^{3}_{6}T $$

where

$$ {}^{0}_{3}T = {}^{0}_{3}T\,{}^{1}_{2}T\,{}^{2}_{3}T $$

$$ {}^{0}_{3}T = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ -s_1 & -c_1 & 0 & 0 \\ 0 & 0 & -1 & -d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_2 & -s_2 & 0 & a_1 \\ 0 & 0 & -1 & 0 \\ s_2 & c_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_3 & -s_3 & 0 & a_2 \\ -s_3 & -c_3 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{0}_{3}T = \begin{bmatrix} c_1c_2 & -c_1s_2 & s_1 & c_1a_1 \\ s_1c_2 & s_1s_2 & c_1 & -s_1a_1 \\ -s_2 & -c_2 & 0 & -d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_3 & -s_3 & 0 & a_2 \\ -s_3 & -c_3 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{0}_{3}T = \begin{bmatrix} c_1c_2c_3 + c_1s_2s_3 & -(c_1c_2s_3 + c_1s_2c_3) & -s_1 & c_1c_2a_2 + c_1a_1 \\ -s_1c_2c_3 - s_1s_2s_3 & s_1c_2s_3 - s_1s_2c_3 & -c_1 & -s_1c_2a_2 - s_1a_1 \\ -s_2c_3 + c_2s_3 & s_2s_3 + c_2c_3 & 0 & -s_2a_2 - d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{0}_{3}T = \begin{bmatrix} c_1c_{32} & -c_1s_{32} & -s_1 & c_1(c_2a_2 + a_1) \\ -s_1c_{32} & s_1s_{32} & -c_1 & -s_1(c_2a_2 + a_1) \\ -s_{32} & c_{32} & 0 & -s_2a_2 - d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4.2} $$

Where

$$ c_{32} = cos(\theta_3 - \theta_2) = c_3c_2 + s_3s_2 $$

, and

$$ s_{32} = sin(\theta_3 - \theta_2) = s_3c_2 - c_3s_2 $$

Similarly, for ${}^{3}_{6}T$

$$ {}^{4}_{6}T = {}^{4}_{5}T\,{}^{5}_{6}T = \begin{bmatrix} c_5 & -s_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s_5 & c_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s_6 & -c_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{3}_{6}T = {}^{3}_{4}T\,{}^{4}_{6}T = \begin{bmatrix} c_4 & -s_4 & 0 & a_3 \\ 0 & 0 & 1 & 0 \\ -s_4 & -c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_5c_6 & -c_5s_6 & -s_5 & 0 \\ s_6 & c_6 & 0 & 0 \\ s_5s_6 & -s_5c_6 & c_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4.3} $$

$$ {}^{3}_{6}T = \begin{bmatrix} c_4c_5c_6 + s_4s_5 & -c_4c_5s_6 + s_4c_6 & -c_4s_5 & a_3 \\ s_5c_6 & -s_5s_6 & c_5 & d_4 \\ -s_4c_5c_6 - c_4s_6 & s_4c_5s_6 - c_4c_6 & s_4s_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4.4} $$

At last, we get

$$ {}^{0}_{6}T = {}^{0}_{3}T \cdot {}^{3}_{6}T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{33} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4.5} $$

where

$$ r_{11} = c_1c_{32}(c_4c_5c_6 - s_4s_6) - c_1s_{32}s_5c_6 + s_1(s_4c_5c_6 + c_4s_6) $$

$$ r_{12} = c_1c_{32}(-c_4c_5s_6 + s_4c_6) + c_1s_{32}s_5s_6 - s_1(s_4c_5s_6 - c_4c_6) $$

$$ r_{13} = -c_1c_{32}c_4s_5 - c1s_{32}c_5 - s_1s_4s_5 $$

$$ r_{21} = -s_1c_{32}(c_4c_5c_6 - s_4s_6) + s_1s_{32}s_5c_6 + c_1(s_4c_5c_6 + c_4s_6) $$

$$ r_{22} = -s_1c_{32}(-c_4c_5s_6 + s_4c_6) - s_1s_{32}s_5s_6 - c_1(s_4c_5s_6 - c_4c_6) $$

$$ r_{23} = -s_1c_{32}c_4s_5 + s_1s_{32}c_5 - c_1s_4s_5 $$

$$ r_{31} = -s_{32}(c_4c_5c_6 + s_4s_6) + c_{32}s_5c_6 $$

$$ r_{32} = -s_{32}(-c_4c_5s_6 + s_4c_6) - c_{32}s_5s_6 $$

$$ r_{33} = -s_{32}c_4s_5 + c_{32}c_5 $$

$$ x = -d_4c_1s_{32} + c_1(c_2a_2 + a_1) + c_1c_{32}a_3 $$

$$ y = d_4s_1s_{32} - s_1(c_2a_2 + a_1) - s_1c_{32}a_3 $$

$$ z = -s_2a_2 - d_1 + d_4c_{32} - s_{32}a_3 $$

Note for here $[x, y, z]$ is the wrist origin. According to the robot frame assignment, to find the position for the TCP with respect to robot base, it is simply a transition along the z axis of the frame{6} by d_6 (120 mm). Therefore, the final position of the end-effector with respect to the robot reference frame can be expressed as:

$$ P_{tcp} = {}^{0}_{6}T \cdot P^{6} $$

$$ P_{tcp} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x_6 \\ r_{21} & r_{22} & r_{23} & y_6 \\ r_{31} & r_{33} & r_{33} & z_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ d_6 \\ 1 \end{bmatrix} $$

Will get end-effector (TCP) position as following:

$$ p^0_x = x_6 + (-c_1c_{32}c_4s_5 - c_1s_{32}c_5 - s_1s_4s_5)d_6 $$

$$ p^0_y = y_6 + (-s_1c_{32}c_4s_5 + s_1s_{32}c_5 - c_1s_4s_5)d_6 $$

$$ p^0_z = z_6 + (-s_{32}c_4s_5 + c_{32}c_5)d_6 $$

You can see from the above equations, if we did not set up the last frame 3, 4, 5 in a common origin ($d_6 \neq 0$) the end-effector position will be a function of all joint positions. If d_6= 0, we know the wrist origin is only determined by θ_1, θ_2, θ_3 and allowing the wrist position to be solved independently of the wrist orientation, which simplifies the overall inverse kinematics problem into a more manageable analytical solution.

So having a Hex7bot end-effector pose (position and orientation), i,e., its homogeneous Transformation matrix is known, we can first get wrist position with respect to base frame

$$x_4 = x_5 = x_6 = p_x - r_{13} \cdot d_6 \tag{5.4.6}$$

$$y_4 = y_5 = y_6 = p_y - r_{23} \cdot d_6 \tag{5.4.7}$$

$$z_4 = z_5 = z_6 = p_z - r_{33} \cdot d_6 \tag{5.4.8}$$

and then call Pieper `IK` .

In this section, we will give another approach (i.e., the mixed geometrical method) for faster solver.

From the wrist position equation:

$$x = -d_4 c_1 s_{32} + c_1(c_2 a_2 + a_1) + c_1 c_{32} a_3 \tag{5.4.9}$$

$$y = -d_4 s_1 s_{32} - s_1(c_2 a_2 + a_1) - s_1 c_{32} a_3 \tag{5.4.10}$$

$$z = -s_2 a_2 - d_1 + d_4 c_{32} - s_{32} a_3 \tag{5.4.11}$$

The projection of the wrist components on x-y planes of frame {0} has the same components on frame {1}. In addition, since both link two and three are planar, the x and y components of the wrist position vector in the frame {1} changes with respect to θ_1 only. Thus, two possible solutions for θ_1 can be achieved by simply applying the arc-tangent function

$$\theta_1 = atan2(x, y) \tag{5.4.12}$$

or

$$\theta_{11} = \pi + \theta_1 \tag{5.4.13}$$

The solutions θ_2 and θ_3 are obtained by considering a plane, shown in Figure 5.4, formed by the second and third planar links, which is also coincident with link {1}'s $x - z$ plane.

The wrist distance ${}^1_4L$ can be expressed as:

$${}^1_4L^2 = {}^1_4x^2 + {}^1_4y^2 + {}^1_4z^2 \tag{5.4.14}$$

and further as:

$${}^1_4L^2 = ({}^0_4x - a_1 \cos\theta_1)^2 + ({}^0_4y - a_1 \sin\theta_1)^2 + ({}^0_4z - d_1)^2 \tag{5.4.15}$$

Note: $({}^0_4x, {}^0_4y)$ is wrist position in robot base (frame{0}).

L_3 is forearm's length and is also hypotenuse of a right triangle. The other two sides are a_3 and d_4 respectively.

$$L_3 = \sqrt{d_4^2 + a_3^2}$$

L_2 is upper arm's length:

$$L_2 = a_2$$

The cosine law is used to solve ζ for θ_3 as follow:

$$\cos\zeta = \frac{(L_2)^2 + (L_3)^2 - {}^1_4L^2}{2L_2L_3} \tag{5.4.16}$$

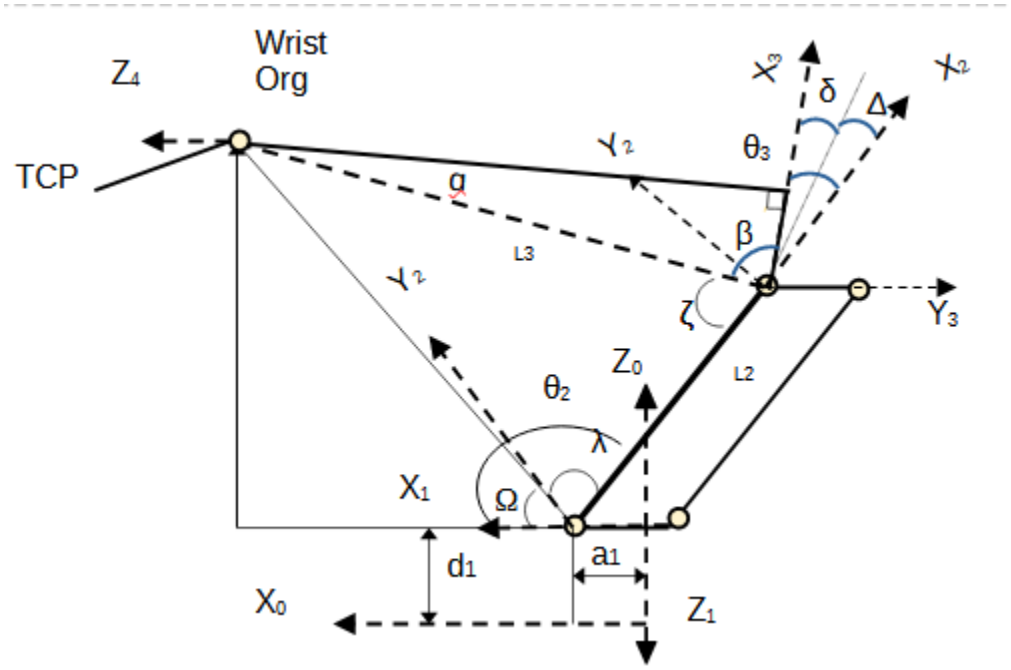

Figure 5.4: Link2&3 or link 1 X-Z plane

From the figure 5.4, we can also see:

$$\theta_3 = \Delta + \delta = \pi - (\beta + \zeta) = \pi - (\frac{\pi}{2} - \alpha) - \zeta = \frac{\pi}{2} + \alpha - \zeta \tag{5.4.17}$$

Where δ is the virtual joint displacement of joint 3 and Δ is the initial offset angle due to the virtual pivot. In particular, Hex7bot's upper arm, forearm and two parallel connecting rods form a parallelogram (4-bar parallel linkage mechanism), and δ is the summation of joint displacements of actuators 2 and 3. The angle α is a constant offset from upper arm to forearm; its value can be solved from

$$\tan(\alpha) = \frac{a_3}{d_4}$$

If ζ is known and θ_3 can also be derived.

In addition, we should notice that the θ_3 rotation direction is opposite to θ_2. The real value of θ_3 in the new D-H frames should be:

$$\theta_3 = -\frac{\pi}{2} - \alpha + \zeta \tag{5.4.18}$$

The negative sign in θ_3 indicates that the rotation occurred in the opposite direction.

Likewise, we can follow the same procedure to solve λ for θ_2 using similar trigonometric relationships.

$$\theta_2 = -(\Omega \pm \lambda) \tag{5.4.19}$$

where

$$\cos(\lambda) = \frac{(L_2)^2 + \frac{1}{4}L^2 - (L_3)^2}{2L_2\frac{1}{4}L}$$

$$\Omega = atan2({}^0_4z - d_1, \sqrt{{}^0_4x^2 + {}^0_4y^2})$$

There are two possible θ_2 depending on elbow positions: For elbow down:

$$\theta_2 = -(\Omega - \lambda)$$

For elbow up:

$$\theta_2 = -(\Omega + \lambda)$$

The Hex7Bot only has elbow up position as shown in Figure 5.4: i.e.,

$$\theta_2 = -(\Omega + \lambda)$$

Please note that θ_2 is derived based on the second modified DH frames in the last section. i.e., both Joint 2 and 3 motors rotate clockwise as positive directions but opposite from the world frame. When the Hex7Bot is in the "Zero" (also called Home) position upon being powered on, the link2 is almost vertical. At this position, the controlled joint (encoder) is 0 and θ_2 will be -90^0. If you want to find the end-effector position using D-H based forward kinematics you need to use $\theta_2 = -90^0$. While if your want to use calculated θ_2 and θ_3 for robot control, you should consider the initial offset and signs. such as $\theta_2 + 90^0$ And $\theta_3 + 30^0$ to get real joint displacements for hex7bot position control. Depending on your DH frames or URDF, you may have different θ_2 and θ_3 signs or expressions.

Again, the rotation occurred in the opposite direction of the z-axis, as well as there being an initial rotation offset for real joint control. The initial offset is calculated/compensated on Joint 2 so that it can fall into real joint limits. In addition, please note that due to the Hex7bot mechanic constraints, only elbow-up robot configuration exists. So there is usually only one θ_2 solution is within the joint limits.

After solving the first 3 angles with a geometrical solution, the next step of the inverse kinematic solution will deal with the procedure of solving the orientation sub-problem to find the joint angles θ_4, θ_5, and θ_6. Since the last 3 axis intersection, we can solve θ_4, θ_5 within the orientation matrix. , such as:

$${}^3_6R = ({}^0_3R)^{-1} \cdot {}^0_6R$$

$${}^3_6R = ({}^0_3R)^T {}^0_6R = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{33} & g_{33} \end{bmatrix}$$

where

$$g_{11} = c_1 c_{32} n_x + s_1 c_{32} n_y + s_{32} n_z$$

$$g_{12} = c_1 c_{32} o_x + s_1 c_{32} o_y + s_{32} o_z$$

$$g_{13} = c_1 c_{32} a_x + s_1 c_{32} a_y + s_{32} a_z$$

$$g_{21} = -c_1 s_{32} n_x - s_1 s_{32} n_y + c_{32} n_z$$

$$g_{22} = -c_1 s_{32} o_x - s_1 s_{32} o_y + c_{32} o_z$$

$$g_{23} = -c_1 s_{32} a_x - s_1 s_{32} a_y + c_{32} a_z$$

$$g_{31} = s_1 n_x - c_1 n_y$$

$$g_{32} = s_1 o_x - c_1 o_y$$

$$g_{33} = s_1 a_x - c_1 a_y$$

It is possible to obtain the solutions for θ_4, θ_5 and θ_6.

$$\theta_5 = atan2(-\sqrt{{g_{13}}^2 + {g_{33}}^2}, g_{22}) \tag{5.4.20}$$

$$\theta_4 = atan2(\frac{g_{33}}{s_5}, \frac{-g_{13}}{s_5}) \tag{5.4.21}$$

and

$$\theta_6 = atan2(-\frac{g_{22}}{s_5}, \frac{g_{21}}{s_5}) \tag{5.4.22}$$

For a singular configuration where the joint axes 4 and 6 are parallel, this results in a similar motion of the last three intersection links of the robot manipulator. $\theta_5 = 0$:

$$\theta_4 + \theta_6 = atan2(-g_{12}, g_{11})$$

There are infinite solutions, and we can choose an optimal solution with the least motion moving distance, such as δ = total distance on joints 4 and 6, and control joints 4 & 6 to move half distance displacement. i.e. as $\theta_5 + \delta/2$ and $\theta_4 + \delta/2$

Base on this solution, we can implement it with C++ code in ROS as:

```
static const double  elbow_ext_ang_limit_deg   155.0; //degree
static const double  elbow_bend_ang_limite_deg  30.0; //degree
int Hex7BotModel::InvKine(const double pos[][4],
        const double jtcurrnt[], double theta[])
{
   int i, j,  base;
   // DH Pole length . The distance measured from Z_i_1 to Z_i along X_i_1;
   double a[MAX_7BOT_JNTS];
   //DH Twist angle. The angle of rotation from Z_i_1 to Z_i along x_i_1;
   double alpha[MAX_7BOT_JNTS];
   // DH offset . The distance measured from X_i_1 to X_i along Z_i;
   double d_off[MAX_7BOT_JNTS];
   double aux[2], rtemp[2];
   int     solution, solunum;
   AUX_RECORD theta1_aux[2], theta5_aux[2];
   AUX_RECORD theta2_aux[2];
   int    jt1_num, jt2_num,jt5_num;

   /* Step 0:  parameter initialization: parameter input & auxiliary value setup */
   base = 0;
   jt1_num = jt2_num  = jt5_num = 0;

   for (i = 0; i < MAX_7BOT_JNTS ; i++)
   {
       a[i] = modDH_Param_[i].a_i_1;
       alpha[i] = modDH_Param_[i].alpha_i_1 ; //rad
       d_off[i] = modDH_Param_[i].d_i;
       theta[i] = DH_Para_[base + 3];
       base += COLUMN;
   }
   double nx = pos[0][0];
   double ny = pos[1][0];
   double nz = pos[2][0];
   double ox = pos[0][1];
   double oy = pos[1][1];
   double oz = pos[2][1];
   double ax = pos[0][2];
   double ay = pos[1][2];
   double az = pos[2][2];
   double px = pos[0][3];  //where px , py , pz is (TCP position)
   double py = pos[1][3];
   double pz = pos[2][3];
```

```
//Hex7bot modified DH table i{1,6} : modified (proximal)
//   DH parameters
// Link_i  a_i_1   alpha_i_1   d_i   theta_i  Joint range
//  1      0       180           10.4    theta[0]  -180~180
//  2      20.0    90          0    theta[1]  -110~100
//  3      120     180           0    theta[2]  -65~65
//  4      30.0    -90         160   theta[3]  -200~200
//  5      0       90           0    theta[4]  -120~120
//  6      0       -90          120   theta[5]  -400~400
//Solve joint 1 by computing the position of the wrist.
//  Wrist position is frame 4,5,6 intersection position
double Px =  px - d_off[5] * ax;
double Py =  py - d_off[5] * ay;
double Pz =  pz - d_off[5] * az;

//There are two possible solutions between 0 to 2*PI because of tangent
rtemp[0] = atan2 (Py, Px);
rtemp[1] = rtemp[0] + M_PI;
```

- Check whether solution θ_1 is in the joint limits

```
// check whether the solution is in the joint limits
solunum = bound_check(rtemp[0], JointLimit_[0].upperlmt,
            JointLimit_[0].lowerlmt, aux);
if (solunum != 0){
    theta1_aux[jt1_num++].angle = aux[0];
}
solunum = bound_check(rtemp[1], JointLimit_[0].upperlmt,
            JointLimit_[0].lowerlmt, aux);
if (solunum != 0) {
    theta1_aux[jt1_num++].angle = aux[0];
}

if (jt1_num == 0) {
    fprintf(stderr, "Could not solve joint 1 , Abort!\n");
    return 0;
} else if (jt1_num == 1) {  //determine final joint 1 angle
  theta[0] = theta1_aux[0].angle;
}
else // if (jt_num[i] == 2){
  theta[0] = fabs(theta1_aux[1].angle - jtcurrnt[0]) <
                   fabs(theta1_aux[0].angle - jtcurrnt[0]) ?
                   theta1_aux[1].angle : theta1_aux[0].angle;
}
```

- Continue checking θ_1 by verifying TCP positions.

```
double L3 = sqrt(d_off[3] * d_off[3] + a[3] * a[3]);;
double L2 = a[2];
double s_2 = (Pz - d_off[0]) * (Pz- d_off[0]);
double r_2 = pow(Px - a[1] * cos(theta[0]), 2) + pow(Py - a[1] * sin(theta[0]), 2);
double Lp = sqrt(s_2 + r_2);
double offsetAngle = atan(a[3]/d_off[3]);

const double max_angle  = elbow_ext_ang_limit_deg /180.0 * M_PI ; //
const double min_angle = elbow_bend_ang_limite_deg/180.0 * M_PI ;

double cos_zeta = (L2 * L2 + L3 * L3 - Lp * Lp) / (2 * L2 * L3);

if (fabs(cos_zeta) > 1.0 + CALC_TOLERANCE){
    fprintf(stderr," ERROR : not reachable End effector position, Abort!\n");
}

double sin_zeta = sqrt (1 - cos_zeta * cos_zeta);
double angleZeta = atan2 (sin_zeta, cos_zeta);

if (angleZeta > max_angle  || angleZeta < min_angle){
  fprintf(stderr," ERROR : TCP is not reachable: angle between L2 and L3=%g
              is not in the range(%g ~ %g) (degree)\n",
              angleZeta /M_PI * 180, min_angle /M_PI * 180, max_angle / M_PI * 180);
  return 0;
}
```

- Solve joint 3 angle

```
//Solve joint 3 angle
//there is one only possible solution for joint 3 because Hex7bot is elbow up.
//Note: in modified DH, joint 3 rotation angle 0 position is when a2 and a3
//  in parallel, and d4 is perpendicular to a2 and a3.
//  so we have theta3 = AngleDelta + offsetAngle
//  where AngleDelta= PI/2 - AngleZeta.
rtemp[0] = - M_PI / 2 + angleZeta - offsetAngle;
solunum = bound_check(rtemp[0], JointLimit_[2].upperlmt,
                JointLimit_[2].lowerlmt, aux);
if (solunum != 0) {
   theta[2]= aux[0]; //convert it back from DH to joint space (-65,60)
}
else {
  fprintf(stderr, "Could not solve joint 3 , possible joint %g and %g are not
  in the range, Abort!\n", rtemp[0], rtemp[1]);
  return 0;
}
```

- Solve Joint 2.

```
//Solve Joint 2 by considering initial rotation of
//-90 degree between axis1 and axis2
double omega = atan2 (Pz - d_off[0], sqrt(r_2));
double lamda = acos ((L2 * L2 + Lp * Lp- L3 * L3) / (2 * L2 * Lp));
// Note: in modified DH, Axis 2 frame's Z is horizontal . The negative
// sign indicates that the rotation occurred in the opposite direction.
// In addition , there are  an initial rotation of -90 between axis 1 and
// and axis.
rtemp[0] =  -(omega + lamda)  + JointLimit_[1].angOffset ;
//there are two possible values for it.
omega = atan2 (Pz - d_off[0], -sqrt(r_2));
rtemp[1] = - (omega - lamda) + JointLimit_[1].angOffset ;
for (i = 0; i < 2; i ++){
  solunum = bound_check(rtemp[i], JointLimit_[1].upperlmt,
          JointLimit_[1].lowerlmt, aux);
  if (solunum != 0) {
      theta2_aux[jt2_num++].angle = aux[0];
   }
}
if (jt2_num==0) {
    fprintf(stderr, "Could not solve joint 2 , Abort!\n");
    return 0;
} if (jt2_num == 1) {//find joint 2 final results
   theta[1] = theta2_aux[0].angle;
}
else {
   theta[1] = fabs(theta2_aux[0].angle - jtcurrnt[1]) <
             fabs(theta2_aux[1].angle- jtcurrnt[1]) ?
                      theta2_aux[0].angle : theta2_aux[1].angle;
 }
```

- Solve joint 5 angle

```
//Solve theta4, and 5 by R6_3 orientation matrix
double c1 = cos (theta[0]);
double s1 = sin (theta[0]);
double c32 = cos(theta[2] - theta[1]);
double s32 = sin(theta[2] - theta[2]);
double g11 = c1 * c32 * nx + s1 * c32 * ny + s32 * nz;
double g12 = c1 * c32 * ox + s1 * c32 * oy + s32 * oz;
double g13 = c1 * c32 * ax + s1 * c32 * ay + s23 * az;
double g21 = -c1 * s32 * nx - s1 * s32 * ny + c32 * nz;
double g22 = -c1 * s32 * ox - s1 * s32 * oy + c32 * oz;
double g23 = -c1 * s32 * ax - s1 * s32 * ay - c32 * az;
double g31 = s1 * nx - c1 * ny;
double g32 = s1 * ox - c1 * oy;
double g33 = s1 * ax - c1 * ay;
//Resolve joint 5
if (fabs(g23) > 1 + 0.001){
  fprintf(stderr, "Could not solve joint 5: fabs(g23)=%g > 1, Abort!\n",
          fabs(g23));
  return 0;
}
else {
  if (g23 > 1.0) g23= 1.0;
  else if (g23 < -1.0) g23 = -1.0;
  rtemp[0] = atan2(-sqrt(g13 *g13 + g33 *g33), g23)
  rtemp[1] = -rtemp[0];
  for (i=0; i < 2; i++){
    solunum = bound_check(rtemp[i], JointLimit_[4].upperlmt,
                  JointLimit_[4].lowerlmt, aux);
    if (solunum != 0) {
       theta5_aux[jt5_num++ ].angle = aux[0];
    }
  }
}
if (jt5_num == 0){
  fprintf(stderr, "Could not find joint 5 in the range: %g and %g are out
                  of range (%g ~ %g), Abort!\n", rtemp[0], rtemp[1],
                  JointLimit_[4].lowerlmt, JointLimit_[4].upperlmt);
  return 0;
}
else if (jt5_num == 1)  {
  theta[4] = theta5_aux[0].angle;
}
else {
  theta[4] = fabs(theta5_aux[1].angle - jtcurrnt[4]) <
                    fabs(theta5_aux[0].angle - jtcurrnt[4]) ?
                    theta5_aux[1].angle : theta5_aux[0].angle;
}
```

- Solve joint 4 & 6 angles

```
  //There are two possible values for singular configuration
  //   when joint axes 4 and 6 area parallel.
  // joint5 = 0 case: joint4+joint6 =
  if (fabs(theta[4]) < CALC_TOLERANCE) {
    double currnt46 = jtcurrnt[3] + jtcurrnt[5] ;
    rtemp[0] = atan2(-g12, g11); //tan46
    rtemp[1] = rtemp[0] + M_PI;
    double delta = fabs(rtemp[0] - currnt46) < fabs(rtemp[1] - currnt46) ?
                rtemp[0] - currnt46 : rtemp[1] - currnt46;
    theta[3] = jtcurrnt[3] + delta /2 ;
    theta[5] = jtcurrnt[5] + delta /2 ;
  }
  else {
    //solve joint _4
    double s5 = sin(theta[4]);
    theta[3] = atan2 (g33 /s5,-g13/ s5);
    solunum = bound_check(theta[3], JointLimit_[3].upperlmt,
                    JointLimit_[3].lowerlmt, aux);
    if (solunum == 0) {
       fprintf(stderr, "Joint4 (%f) is not in the range(%f ~ %f),
           Abort! \n", theta[3], JointLimit_[3].lowerlmt,
            JointLimit_[3].upperlmt);
    }
    //solve joint 6
    theta[5] = atan2(-g22/s5,g21 /s5);
    solunum = bound_check(theta[5], JointLimit_[5].upperlmt,
            JointLimit_[5].lowerlmt, aux);
    if (solunum == 0) {
       fprintf(stderr, "Joint5 (%f) is not in the range(%f ~ %f),
                Abort!\n",theta[5], JointLimit_[5].lowerlmt,
                JointLimit_[5].upperlmt);
    }
  }

  fprintf(stdout, "IKF solutions(deg): J1=%g, J2=%g, J3=%g, J4=%g, J5=%g, J6=%g\n",
        theta[0] * 180/ M_PI, theta[1] * 180/ M_PI, theta[2] * 180/ M_PI,
        theta[3] * 180/ M_PI, theta[4] * 180/ M_PI, theta[5] * 180/ M_PI);

  return 1;
}
```

Note that since this function is specially for Hex7botModel, we implemented it in a subclass (inherited from BaseMode) for fast inverse kinematics computation. In one later chapter of this book, we will specially discuss software architecture, such as how to modularize different software components with the OOP (Object-Oriented Programming) paradigm so that we can reuse common code common libraries in the near future.

5.5 Robot Jacobian

We have explored forward and inverse kinematics for robot position control. What about velocity or speed control? While we could compute robot forward and inverse kinematics with a high loop rate to derive joints velocities, this approach is not efficient for advanced tasks such as motion planning and trajectory following. Instead, we turn to differential kinematics, which aims to establish the relationship between joint velocities and the end-effector's linear and angular velocities. This mapping is described by a matrix called the geometric `Jacobian` , which depends on the manipulator's configuration.

In total, we tend to describe the end-effector linear velocity $\dot{p}$ and angular velocity ω as a function of joint velocities $\dot{q}$.

$$v_{6x1} = \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix}_{6x1} = J(q)\dot{q} = \begin{bmatrix} J_{P(3xn)} \\ J_{O(3xn)} \end{bmatrix}_{6xn}$$

The $(6xn)$ matrix J is the manipulator geometric `Jacobian` which in general is a function of the joint variables.

In resolved motion control, one needs to determine how each infinitesimal joint motion affects the infinitesimal motion of the manipulator hand. One advantage of resolved motion is that there exists a linear mapping between the infinitesimal joint motion space and the infinitesimal hand motion space. This mapping is defined by a Jacobian.

The time derivative of the kinematics equations yields the Jacobian of the robot, which relates the joint rates to the linear and angular velocity of the end-effector. The principle of virtual work shows that the Jacobian also provides a relationship between joint torques and the resultant force and torque applied by the end-effector. Singular configurations of the robot are identified by studying its Jacobian.

5.5.1 Velocity kinematics

The robot Jacobian results in a set of linear equations that relate the joint rates to the six vectors formed from the angular and linear velocity of the end-effector, known as a twist. Specifying the joint rates yields the end-effector twist directly.

The inverse velocity problem seeks the joint rates that provide a specified end-effector twist. This is solved by inverting the Jacobian matrix. It can happen that the robot is in a configuration where the Jacobian does not have an inverse. These are termed singular configurations of the robot.

In this section we derive the `jacobian` matrix for 6R robot arms. We showed that the `jacobian` matrix can be shown as $(6xn)$ matrices. J can be partitioned into $(3x1)$ column vectors. In these arms the J matrices are $6x6$ and can be show as:

$$J = \begin{bmatrix} J_{P1} & J_{P2} & J_{P3} & J_{P4} & J_{P5} & J_{P6} \\ J_{O1} & J_{O2} & J_{O3} & J_{O4} & J_{O5} & J_{O6} \end{bmatrix}$$

Since all the joints are revolute, J_{Pi} and J_{Oi} are computed by:

$$J_{Pi} = z_{i-1}x(P - p_{i-1})$$

$$J_{Oi} = z_{i-1}$$

Where P is the position of end-effector with respect to Base reference. p_{i-1} is the position of each revolute joint with respect to base frame and can be derived from the first three elements of the fourth column of the transformation matrix T_n^0.

$$p_{i-1} = A_i^0(q1) \dots A_{i_1}^{i-2}(q_{i-1})p_0$$

Where $i = 1:6$ and $p_0 = [0001]^T$ allows selecting the fourth desired column. z_{i-1} is the joint axis vector of each revolute joint and can be expressed by

$$z_{i-1} = R_1^0(q_1) \dots R_{i-1}^{i-2}(q_{i-1})z_0$$

Where i=1:6 and $z_0 = [001]^T$ allows selecting the third desired column.

Kinematic Singularity

While singularities have non-local implications for the control and use of manipulators, they arise as local or instantaneous phenomena from the rand deficiency of a derivative. For serial manipulators, it is the singularity of the kinematic mapping/forward kinematics and trajectories that are of interest. When singularity occurs:

1. Loss of freedom:

The derivative of the kinematic mapping - specifically, the forward kinematics - represents the conversion of joint velocities into generalized end-effector velocities(i.e., linear and angular velocities). This linear transformation is commonly referred to as the manipulator Jacobian in robotics literature. A drop in the rank of this matrix reduces its effective dimention, corresponding to a loss of one or more degrees of instantaneous motion for the end-effector.

2. Workspace:

When a manipulator is at a boundary point of its workspace, it can not move freely in all Cartesian directions and making the inverse kinematics unsolvable or degenerate. This condition often causes motors to stall (i.e., stop due to reaching their maxium torque). Additionally, when the manipulator moves from one singular region to another that has different inverse kinematic solution, a significant change in posture(e.g from elbow-up to elbow-down) may occur which can cause the robot to overspeed.

3. Loss of control:

A variety of control systems is used for manipulators. Rate control systems require the end-effector to traverse a path at a fixed rate and therefore determine the required joint velocities by means of the inverse of the derivative of the (known) forward kinematics. Near a singularity, this matrix is ill-conditioned, and either the control algorithm fails or the joint velocities and accelerations may become unsustainable. Conversely, force control algorithms, well-adapted for parallel manipulators, may result in intolerable joint forces or torques near singularities of the projection onto the joint space.

```
//Create Jacobian matrix with Modified DH for velocity and control where jt_current
// is the position of each revolute joint. Require all all last three joint intersect
// and don't consider d6. So Ptcp = T6_0 * P6
vector<vector<double> > Hex7BotModel::getJacobian(vector<double>& jt_currnt){
  vector<vector<double> >Ai;
  T_[0] = TransformMatrix(modDH_Param_[0].a_i_1, modDH_Param_[0].alpha_i_1,
       modDH_Param_[0].d_i, jt_currnt[0] - JointLimit_[0].angOffset);
  for (int i = 1; i < myDOF; i++){
    if (i < myDOF - 1) {
       Ai = TransformMatrix(modDH_Param_[i].a_i_1, modDH_Param_[i].alpha_i_1,
              modDH_Param_[i].d_i, jt_currnt[i] - JointLimit_[i].angOffset);
    }else {
       Ai = TransformMatrix(modDH_Param_[i].a_i_1, modDH_Param_[i].alpha_i_1, 0,
             jt_currnt[i] - JointLimit_[i].angOffset);
    }
    T_[i] = vecmat::matrixMultiply(T_[i-1], Ai);
  }
  //Create all zi and pi
  vector<vector<double> > z(T_.size(), vector<double>(3, 0.0));  // 6 x 3 matrix
  vector<vector<double> > p(T_.size(), vector<double>(3, 0.0));  // 6 x 3 matrix
  vector<double> Pee = { 0, 0, 0};
  //Where p is the wrist center with respect to Base. Pee is end-effector (TCP)
  // with respect to with respect to base frame.
  for (size_t i = 0; i < T_.size(); i ++){
      z[i][0] = T_[i][0][2]; z[i][1] = T_[i][1][2]; z[i][2] = T_[i][2][2];
      p[i][0] = T_[i][0][3]; p[i][1] = T_[i][1][3]; p[i][2] = T_[i][2][3];
  }
  Pee[0] =  p[myDOF-1][0] + z[myDOF-1][0]  * modDH_Param_[myDOF-1].d_i ;
  Pee[1] =  p[myDOF-1][1] + z[myDOF-1][1]  * modDH_Param_[myDOF-1].d_i ;
  Pee[2] =  p[myDOF-1][2] + z[myDOF-1][2] * modDH_Param_[myDOF-1].d_i ;
  //Create cross
  vector<vector<double> >Jp(T_.size(), vector<double>(3, 0.0));
  vector<double> subP (vector<double>(3, 0.0));
  for (int i = 0; i < T_.size() ; i ++){
    subP = vecmat::operator- (Pee , p[i]);
    Jp[i] = vecmat::cross_product(z[i], subP);
  }
  vector<vector<double > > jacobian(6,  vector<double>(T_.size(), 0));
  for (size_t i = 0; i < T_.size(); i++){
    jacobian[0][i] = Jp[i][0];
    jacobian[1][i] = Jp[i][1];
    jacobian[2][i] = Jp[i][2];
    jacobian[3][i] = z[i][0];
    jacobian[4][i] = z[i][1];
    jacobian[5][i] = z[i][2];
  }
  return jacobian;
}
```

5.5.2 SVD of the Jacobian

`SVD` (Singular value decomposition) is used in robotics to understand and manage singularities, which are configurations where the robot's `Jacobian` matrix loses rank, and the end-effector loses degree of freedom.

The `SVD` decomposes the Jacobian into three matrices, revealing singular values that indicate the degrees of freedom in the input (joint space) and output (task space). By examining the singular values, particularly those close to zero, the robot controller can identify the degenerate directions of motion associated with the singularity and plan appropriate joint or task space actions to avoid them or safely navigate through them.

The SVD decomposes the Jacobian matrix J into three parts:

$$J = U\Sigma V^T$$

where U: An orthogonal matrix containing the left singular vectors that span the output space.

Σ: A diagonal matrix of singular values, ordered from largest to smallest. V^T: The transpose of an orthogonal matrix containing the right singular vectors, which span the input space.

- Identifying degenerate Directions: Singular values (σ) close to zero in the Σ matrix signify the loss of degrees of freedom in the task space. These directions correspond to the null space of the Jacobian, where the robot's1 end-effector cannot move effectively.
- Singularity Avoidance: By monitoring the singular values, a robot can identify approaching singularities and adjust its path to prevent them. In MoveIt!, MoveIt! servo is the tool which monitors end-effector trajectories by real-time calculate `SVD` of the Jacobian.

In the next chapters, when we do robot arm control even with position control only, we not only update new commands by computing robot inverse kinematics for new joints' positions, prior to sending these commands to actuator/servo controller, we also need to check all joints' velocities by computing the robot `Jacobian` matrix for two purposes: One is to ensure all joints are within their maximum joint velocities limits. The second is to make sure end-effector is not moving toward or closing singularity space. Enforce joint velocity limits if new commands cause joint velocity to be too high or potentially damage the system. If the end-effector is moving toward singularity or in singular space and the Jacobian matrix is singular or (non-invertible). The current control command has to be interrupted or halted when very close to singularity.

5.6 Robot Kinematics with Quaternions

We have seen from the previous sections' code snippet that for any control sampling points, e.g., for a given `TCP` pose to get expected joint variables for control; there are a lot of vector and matrix computations ($4x4$ matrix multiplications or matrix and $4x1$ vector dot product). Particularly for real robot hardware control, due to real-robot hardware in the closed-loop, it is vital to check all boundaries and real constraints by calculating its forward kinematics and inverse kinematics back-and-forth. Usually, the default robot model provided by the ROS URDF, which is good for vitalization and offline simulation but not transparent and cannot solve this real-time issue effectively. So customizing your robot kinematics with a high-performance linear algebra library such as `tf2` or `Eigen'` by implementing efficient computation method, e.g., `quaternions` become more and more popular and End-Effector pose is expressed as `geometry_msg/Pose` message. This message contains both a `Point` for position(x,y,z) and a `quaternion` for orientation. While quaternions are the preferred way to represent orientation in ROS due to their mathematical advantages (e.g., avoiding gimbal lock), you can convert them to Roll, Pitch, and Yaw (RPY) angles for easier interpretation.

```
#include <ros/ros.h>
#include <geometry_msgs/Pose.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>

void endEffectorPoseCallback(const geometry_msgs::Pose::ConstPtr& msg)
{
   ROS_INFO("End-effector Position: x=%g, y=%g, z=%g", msg->position.x,
        msg->position.y, msg->position.z);
   //Convert the quaternion to Roll, Pitch, Yaw
   tf2::Quaternion q(msg->orientation.x, msg->orientation.y,
                     msg->orientation.z, msg->orientation.w);
   tf2::Matrix3x3 m(q);
   double roll, pitch, raw;
   m.getRPY(roll, pitch, yaw); //Roll, Pitch, Yaw in radians

   ROS_INFO("End-Effector Orientation(RPY), Roll=%g, Pitch=%g, Ya=%g",
            roll, pitch, yaw);
 }
```

```
int main(Int argc, char **argv) {
  ros::init(argc, argc, "end_effector_pose_ndoe");
  ros::NodeHandle nh;

  //Subscribe
  ros::Subscriber sub = nh.subscribe("/my_robot/end_effector_pose", 10,
         endEffectorPoseCallback);
  //Publish a static pose
  ros::Publisher pub= nh.advertise<geometry_msgs::Pose>
  ("/my_robot/end_feector_pose",10);

  geometry_msgs:;Pos pose_to_publish;
  pose_to_publish.position.x = 0.5;
  pose_to_publish.position.y = 0.1;
  pose_to_publish.position.z = 0.2;

  // Set orientation to a specific RPY (.e.g 0, 0. pi/2)
  tf2::Quaternion q_rpy;
  q_rpy.setRPY(0.0, 0.0. M_PI/2); //Roll, Pitch , Yaw in radians
  pose_to_publish.orientation = tf2::toMsg(q_rpy);

  ros::Rate loop_rate(1); //1Hz
  whiel(ros::ok()
  {
      publ.publish(pose_to_publish);
      ros::spinOnce();
      loop_rate.sleep();
   }
   return 0;
 }
```

`Eigen` is also a powerful C++ template library for linear algebra, and it finds extensive use in various aspects of robotics due to its efficiency, versatility, and comprehensive features. Eigen provides efficient data structures for representing 3D positions, velocities, accelerations, and offers robust tools for handling rotations and orientations using `quaternions` and rotation matrices, enabling smooth and singularity-free rotation interpolation.

We have learned in the last chapter that rotating along a 3D axis/point/vector along θ can be accomplished using quaternion multiplication/division and addition/subtraction operations, more efficient for rotations compared to the rotation matrix (12 floats for $4x4$ matrix). Quaternions are much easier for robot trajectory planning, especially interpolation (i.e., rotate from angle a to b over t seconds). For rotation matrices, you need to recalculate all of the trig values for each intermediate step, whereas for quaternions you simply have a linear combination of your start and end quaternions. A matrix can also represent other transformations than just a rotation, for example, scaling. Sometimes numerical instabilities can cause a rotation matrix to "drift" from being a true rotation matrix (such as its determinant is not 1 or not full rank) and can not solve robot joint angles by inverse kinematics even if the robot is not in singular space.

Another significant advantage of quaternions, as they are not subject to `gimbal lock` , making them ideal for singularity-free rotation interpolation. Gimbal lock is the loss of one degree of freedom in a

multi-dimensional mechanism at certain alignments of the axes.

In robotics, `Gimbal lock` is commonly referred to as "wrist flip", where three axes of the wrist intersect one common point. Gimbal lock occurs when frame {4} and frame {6} coordinates coincide, i.e., $z - axis$ of frame {4} and frame {6} are lined up, "locking" the end-effector into rotation in a degenerate two-dimensional space.

Warning: If you have three `Euler angle` rotations for the X, Y, and Z axes, turning them into a quaternion and then into a matrix won't help you. Gimbal lock arises because of the representation of your desired rotation. If you store the rotation that you want as X-Y-Z Euler angles, then you will get `Gimbal lock` .

However, the drawback of quaternions is that they are only for rotations, not suit for translation.

In `Eigen` , you can combine a quaternion's rotation with a translation into a single 4x4 homogeneous transformation matrix using Eigen::Affine3d. A quaternion represents the rotational part, and you can convert it to a $3x3$ rotation matrix using its .matrix() method. Then, construct an Eigen::Affine3d and use its .translate() and .rotate() methods or directly set the rotation and translation to form the final $4x4$ matrix that represents both.

Matrices have the advantage of being "easier" to understand what they mean. With the high-performance CPU such as Raspberry Pi 1.5GHz, communication efficiency is not an issue for one robot-arm. The sample code in this books does not use Eigen library, instead build one lightweight vector and matrix library.

5.7 Summary

Although a lot of robotics resources elaborate kinematics from different prospective, they seldom explained how to implement them for robot real-time control. We are trying to bridge theory and practice step by step with real examples. With more and more modern emerging software/tools for design automation, it may not be necessary for you to know all knowledge points in this chapter, especially a lot of linear algebra mathematics behind it.

However, if you really want to build your robot arm and derive your model for real-control, insights to your robot model and kinematics in this chapter are really essential. It will also take a long time and effort to improve your robot kinematics model (from simulation to real-time control with a lot of practices, experiments, and verifications). From the next chapter, we will discuss how to apply your robot kinematics for real-time control.