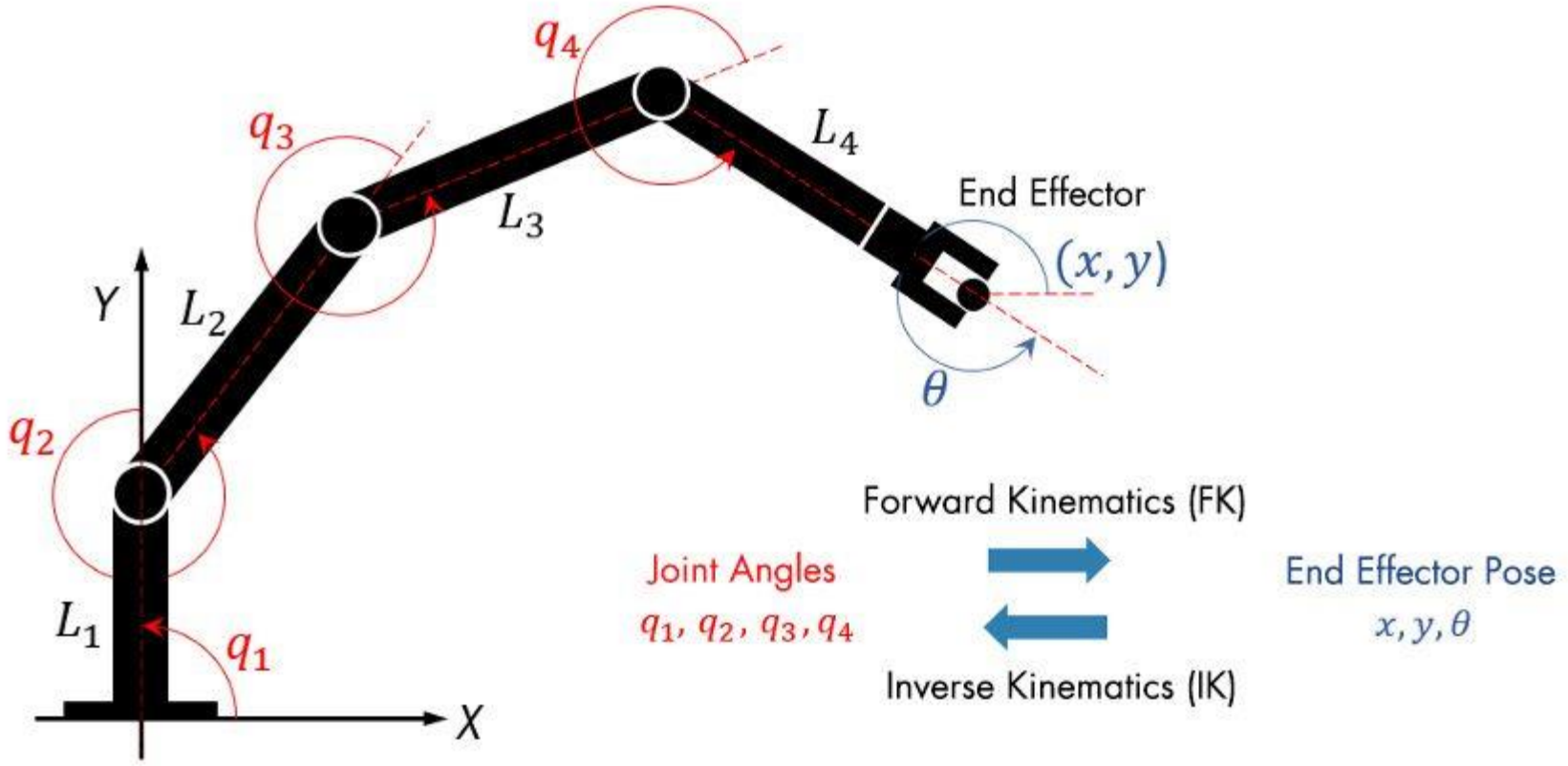
로봇학 실험 4

Kinematics & Inverse Kinematics

민준서 로봇학부

KWANGWOON UNIVERSITY



Forward Kinematics는 처음 주어진 각도를 이용해 end-effector의 position을 구하는 과정임. 이 과정에서 DH parameter을 이용하여 end –effector의 position을 대입을 통해 간단하게 구할 수 있음.

Inverse Kinematics는 최종위치를 제어하기 위해 각도를 구하는 과정을 의미함 forward kinematics 연산에서의 변환행렬 곱의 역연산을 통해 최종 위치(목표위치)값을 이용해 각 joint의 이동 각도를 구할 수 있음.

```
%% Kinematics
alpha = [0 0 0];
a = [0 0.5 0.45];
d = [0 0 0];
theta = [0*pi/180 0*pi/180 0];

% transformation matrix
T01 = dhTransform(theta(1), d(1), a(1), alpha(1));
T12 = dhTransform(theta(2), d(2), a(2), alpha(2));
T23 = dhTransform(theta(3), d(3), a(3), alpha(3));

%% Calculate the end-effector transformation matrix
T03 = T01 * T12 * T23;

%% DH transformation matrix
function T = dhTransform(theta, d, a, alpha)
    T = [
        cos(theta), -sin(theta)*cos(alpha), sin(theta)*sin(alpha), a*cos(theta);
        sin(theta), cos(theta)*cos(alpha), -cos(theta)*sin(alpha), a*sin(theta);
        0, sin(alpha), cos(alpha), d;
        0, 0, 0, 1;
    ];
end
```

편집기 - kinematics.m

T03

4x4 double

	1	2	3	4	5
1	1	0	0	0.9500	
2	0	1	0	0	
3	0	0	1	0	
4	0	0	0	1	
5					
6					

코드에서 계산의 용이성을 위해 DH parameter를 이용해 변환행렬을 만드는 함수를 정의하여 forward kinematics 연산부분에서는 행렬 곱 연산으로만 구할 수 있도록 구현함.

각 파라미터는 강의자료에 있는 파라미터를 이용하여 행렬을 구성하였음.

해당 연산의 결과로 0.9500 의 정확한 Forward Kinematics 연산이 수행된 것을 볼 수 있음.

Matlab Inverse Kinematics 계산

```
%% inverse kinematics
syms theta1  theta2;
theta_i = [theta1*pi/180 theta2*pi/180 0];
T_i01 = dhTransform(theta_i(1), d(1), a(1), alpha(1));
T_i12 = dhTransform(theta_i(2), d(2), a(2), alpha(2));
T_i23 = dhTransform(theta_i(3), d(3), a(3), alpha(3));

T_i03 = T_i01 * T_i12 * T_i23;

%end effector pose
x = T_i03(1,4);
y = T_i03(2,4);
z = T_i03(3,4);

x_desired;
y_desired;
z_desired;

T_d = [
    1, 0, 0, x_desired;
    0, 1, 0, y_desired;
    0, 0, 1, z_desired;
    0, 0, 0, 1;
];

eq1 = T_i03(1,4) == T_d(1,4);
eq2 = T_i03(2,4) == T_d(2,4);


solutions = solve([eq1, eq2], [theta1, theta2]);

theta1_solution = solutions.theta1;
theta2_solution = solutions.theta2;
```

이전 Forward Kinematics를 이용하기 위하여 일반화된 Inverse Kinematics 연산을 할 수 있는 코드를 구현함.

실제 배포 프로젝트에선 강의자료에 구현된 inverse kinematics를 계산하기 위한 DH Parameter를 이용한 방법이 아닌 기하학적 특성을 이용하여 구현한 코드를 구현함.

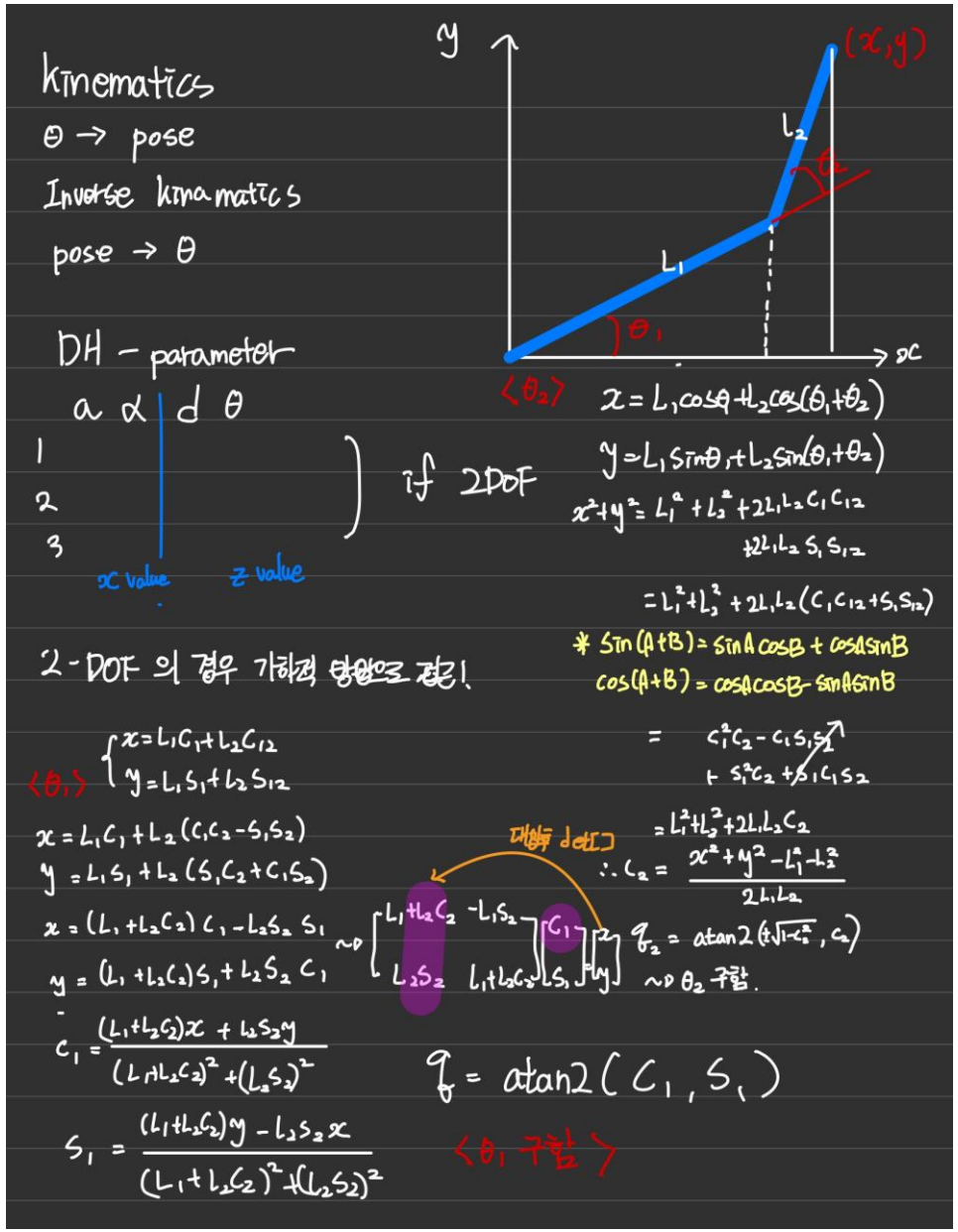
두 방법으로 모두 구현한 결과 실제 내부 연산은 둘 다 동일한 형태의 연산이 진행되는 것을 확인할 수 있음.

```
void CRobotExp_4Dlg::SolveForwardKinematics(double dAngle, double dAngle2, double* pdPos)
{

    double L1 = 1;
    double L2 = 0.5;

    pdPos[0] = L1 * sin(dAngle * DEG2RAD) + L2 * sin(dAngle * DEG2RAD + dAngle2 * DEG2RAD);
    pdPos[1] = 0;
    pdPos[2] = 0.5 + L1 * cos(dAngle * DEG2RAD) + L2 * cos(dAngle * DEG2RAD + dAngle2 * DEG2RAD);

}


void CRobotExp_4Dlg::SolveInverseKinematics(double dX, double dY, double dZ, double* pdAngle)
{
    double L1 = 1;
    double L2 = 0.5;

    dZ = dZ - 0.5;

    double c2 = (pow(dZ, 2) + pow(dX, 2) - (pow(L1, 2) + pow(L2, 2)))
        / (2 * L1 * L2);
    double s2 = abs(sqrt(1 - pow(c2, 2)));

    double q2 = atan2(s2, c2);

    double c1 = ((L1 + L2 * c2) * dZ + L2 * s2 * dX)
        / (pow(L1 + L2 * c2, 2) + pow(L2 * s2, 2));
    double s1 = (-L2 * s2 * dZ + (L1 + L2 * c2) * dX)
        / (pow(L1 + L2 * c2, 2) + pow(L2 * s2, 2));

    double q1 = atan2(s1, c1);

    // theta 1
    pdAngle[0] = q1 * RAD2DEG;
    // theta 2
    pdAngle[1] = q2 * RAD2DEG;
}
```

기하적인 특성을 이용하여 Kinematics를 계산하는 코드를 구현함.

실제로 DHparameter를 이용하여 코드를 구현할 경우 Matrix 객체와 행렬연산을 구현하여야 하는 과정에서 실제 필요한 연산에 비해 과한 연산과 구현 난이도를 필요로 한다고 판단함.

강의자료에 제공된 식과 같이 각 joint의 position과 end-effector의 position 등의 제공값을 이용하여 forward kinematics와 inverse kinematics를 구할 수 있었음.



구현 과정에서 DH-parameter를 이용하여 구현하려고 하던 중에 행렬연산과 행렬 객체를 만드는 과정에서 vector로 코드를 구현하려 하던 중 시간이 부족할 것 같아 강의자료에 나타난 문제를 계산하기 위한 코드로 구현함. 다형성 측면에서 부족한 코드라고 느낌. 또 코드에서 하드코딩 된 부분이 많아 실제로 활용할 경우에는 더욱 일반화된 상황을 해결할 수 있도록 코드의 수정이 필요할 것 같음.

실제 주어진 문제를 해결하기 위한 코드를 구현하기 위해 직접 수식을 계산하였음.