Forward Kinematics is a function mapping a robots configuration to the orientation and position of the robots joints in 3D space (most importantly: the end effector position and orientation).
Forward Kinematic is often described using a matrix, also called a Homogenous Transformation matrix, that is structured as follows:

$$ {}^{i}A_j = \begin{bmatrix} {}^{i}R_j & d_j^i \\ 0 & 1 \end{bmatrix} $$

The matrix is of size 4×4, and it describes the transformation (translation and orientation) of a vector from the axes of joint i, to joint j. The most important transformation matrix is between the base of the robot and the tool, often written as follows: ${}^{0}A_t$.
The matrix is built of several components:

- ${}^{i}R_j$ - The 3×3 rotation matrix between axis i to axis j.
- d_j^i - The 3×1 translation vector in axes i to the zero position of axis j
- The last row of the matrix is made of three zeros and ends in a 1.

Inverse Kinematics is a function that maps a position and orientation in 3D space to a robot configuration. Note that for a serial manipulator there could be several mappings to the same point.
The function for inverse kinematics given in the HW expects two inputs, a homogenous transformation matrix, and parameters for each manipulator in the laboratory.
In order to find, for example, the robot configuration for a required end effector position, you need to find the transformation matrix relevant to that specific position from the base of the manipulator, to the end effector. d_t^0 is the vector from the base of the robot to the end effector position and ${}^{0}R_t$ is the rotation vector describing the orientation of the end effector.
The manipulator's frame of reference is according to the Denavit-Hartenberg convention.
The following example for a Homogenous Transformation Matrix is created using Tait-Bryan angles, which is a formalism similar to Euler angles.

$$ R = \underbrace{R_z(\gamma)}_{\text{roll}} \underbrace{R_y(\beta)}_{\text{pitch}} \underbrace{R_x(\alpha)}_{\text{yaw}} = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} $$

$$ = \begin{bmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta \end{bmatrix} $$

Code Usage Examples

```python
import numpy as np

transform = np.matrix([
    [np.cos(beta) * np.cos(gamma), np.sin(alpha) * np.sin(beta) * np.cos(gamma) - np.cos(alpha) * np.sin(gamma),
     np.cos(alpha) * np.sin(beta) * np.cos(gamma) + np.sin(alpha) * np.sin(gamma), tx],
    [np.cos(beta) * np.sin(gamma), np.sin(alpha) * np.sin(beta) * np.sin(gamma) + np.cos(alpha) * np.cos(gamma),
     np.cos(alpha) * np.sin(beta) * np.sin(gamma) - np.sin(alpha) * np.cos(gamma), ty],
    [-np.sin(beta), np.sin(alpha) * np.cos(beta), np.cos(alpha) * np.cos(beta), tz],
    [0, 0, 0, 1]
])

    IKS = inverse_kinematic_solution(DH_matrix_UR5e, transform)

    ur_params = UR5e_PARAMS(inflation_factor=1)
    env = Environment(env_idx=env_idx)
    transform = Transform(ur_params)
    bb = BuildingBlocks3D(transform=transform, ur_params=ur_params, env=env, resolution=0.1, p_bias=0.05)
    visualizer = Visualize_UR(ur_params, env=env, transform=transform, bb=bb)
    candidate_sols = []
    for i in range(IKS.shape[1]):
        candidate_sols.append(IKS[:, i])
    candidate_sols = np.array(candidate_sols)

    # check for collisions and angles limits
    sols = []
    for candidate_sol in candidate_sols:
        if bb.is_in_collision(candidate_sol):
            continue
        for idx, angle in enumerate(candidate_sol):
            if 2*np.pi > angle > np.pi:
                candidate_sol[idx] = -(2*np.pi - angle)
            if -2*np.pi < angle < -np.pi:
                candidate_sol[idx] = -(2*np.pi + angle)
        if np.max(candidate_sol) > np.pi or np.min(candidate_sol) < -np.pi:
            continue
        sols.append(candidate_sol)

    # verify solution:
    final_sol = []
```

```
    for sol in sols:
        transform = forward_kinematic_solution(DH_matrix_UR5e, sol)
        diff = np.linalg.norm(np.array([transform[0,3],transform[1,3],transform[2,3]])-
                                    np.array([tx,ty,tz]))
        if diff < 0.05:
            final_sol.append(sol)
    final_sol = np.array(final_sol)
```