Lab 2
Explaining the Code

SerialLink Robot Model Creation (mdl_ur3)

%%Create the SerialLink robot (UR3 model in Robotics toolbox):
mdl_ur3
In this section, we have created a SerialLink robot model for the UR3 robot using the Robotics Toolbox in MATLAB. The mdl_ur3 function is likely a predefined function in the toolbox that sets up the kinematic and dynamic parameters for the UR3 robot.
%%Locate the tool frame at 15 cm in the z-direction:
ur3.tool = transl(0, 0, 0.15);
Here, we have adjusted the tool frame of the UR3 robot. The transl function is used to create a homogeneous transformation matrix representing a translation. In this case, we are translating the tool frame by 15 cm in the z-direction.

Creating Twists for Different Work Modes

Work Mode A: Translation

matlab
```
% Workmode A: Translation
T1_A = [1 0 0 0 0 0];
T2_A = [0 1 0 0 0 0];
T3_A = [0 0 1 0 0 0];
```

In this section, we had done the twists for Work Mode A, where the robot is required to perform translation. Each twist (T1_A, T2_A, T3_A) is a 6-element vector representing the motion in the 6 degrees of freedom (x, y, z, roll, pitch, yaw). For Work Mode A, the twists represent pure translations along the x, y, and z axes, respectively.

Work Mode B

matlab
```
% Work Mode B:
T1_B = [0 0 0 1 0 0];
T2_B = [0 0 0 0 1 0];
T3_B = [0 0 1 0 0 0];
```

Here, we have defined twists for Work Mode B. In this case, the robot is required to perform a different motion. T1_B represents a pure rotation about the x-axis, T2_B about the y-axis, and T3_B a pure translation along the z-axis.

Updating Jacobian Matrix

```
% Update Jacobian
J = ur3.jacob0(q);
```

In this section, we have updated the Jacobian matrix (J) for the UR3 robot model. The Jacobian matrix relates the joint velocities to end-effector velocities and is crucial for mapping the robot's joint

space to its task (end-effector) space. The ur3.jacob0(q) function is likely a built-in function in the Robotics Toolbox that computes the Jacobian matrix at the given joint configuration q.

Transformation Matrix Calculations

```
% Transformation matrix from base to TCP
Tbase_tcp = ur3.fkine(q);

% Transformation matrix from TCP to base
Ttcp_base = inv(Tbase_tcp);
```

In these lines, we have calculated transformation matrices.

- Tbase_tcp represents the homogeneous transformation matrix from the robot's base to the tool center point (TCP) given the joint configuration q. The ur3.fkine(q) function is likely a function in the Robotics Toolbox that computes the forward kinematics, providing the transformation matrix.
- Ttcp_base represents the inverse of the transformation matrix from TCP to the base. This matrix is obtained by taking the inverse of Tbase_tcp using the inv function.

Desired Differential Motion Calculation

```
% Desired differential motion (twist) in the new frame:
if work_mode == 0 % Translation
   T1 = T1_A';
   T2 = T2_A';
   T3 = T3_A';
   scale = 0.07;
   scale_B = 0.07;
end

if work_mode == 1 % Insertion
   T1 = tr2jac(Ttcp_base) * T1_B';
   T2 = tr2jac(Ttcp_base) * T2_B';
   T3 = tr2jac(Ttcp_base) * T3_B';
   scale = 0.1;
   scale_B = 0.07;
end
```

In this section, we have determined the desired differential motion (twist) based on the specified work mode.

- For work_mode == 0 (Translation): The twists T1_A, T2_A, and T3_A (defined earlier) are used, and a scaling factor (scale) is set to 0.07.
- For work_mode == 1 (Insertion): The twists T1_B, T2_B, and T3_B (defined earlier) are transformed using the Jacobian of the transformation from TCP to the base (Ttcp_base). The scaling factors (scale and scale_B) are set to 0.1 and 0.07, respectively.

Associate Twist with Joystick Inputs

```
% Associate our twist with the joystick buttons (s1, s2, s3):
vd = (scale * s1 * T1 + scale * s2 * T2 + scale_B * s3 * T3);
```

Here, the joystick inputs (s1, s2, s3) are used to modulate the desired twist. The twists are scaled by joystick values (s1, s2, s3) and the predefined scaling factors. The resulting desired twist (vd) represents the commanded motion based on joystick inputs.

Compute Joint Velocities

```
% Compute joint velocities:
q_dot = pinv(J) * vd;
```

Using the pseudo-inverse of the Jacobian matrix (pinv(J)), joint velocities (q_dot) are computed from the desired twist (vd). This is a common approach to solve the inverse kinematics problem, mapping end-effector velocities to joint velocities.

Check Joint Velocity Magnitude

```
% Check not to send q_dot with magnitude greater than 1:
if norm(q_dot) > 1
    q_dot = q_dot / norm(q_dot);
end
```

This section ensures us that the magnitude of the computed joint velocities (q_dot) does not exceed 1. If it does, the joint velocities are normalized to ensure they are within a reasonable range. This is often done to prevent excessive joint motions and maintain stability.

Code Justification and Theoretical Explanation

Creating the SerialLink Robot Model

- **Code**: **mdl_ur3**
- **Justification**: This line calls a predefined function in the Robotics Toolbox that sets up the UR3 robot's kinematic model. The UR3 is a commonly used robot arm in research and industry, and its model includes the robot's physical parameters and joint configurations.
- **Theory**: In robotic kinematics, a SerialLink model represents a series of linked joints, which is the fundamental structure of most robotic arms. This model is crucial for calculating the robot's pose and planning its movements.

Locating the Tool Frame

- **Code**: **ur3.tool = transl(0, 0, 0.15);**
- **Justification**: This line sets the tool frame of the UR3 robot model to be 15 cm along the z-axis. The tool frame is the reference point for all end-effector operations, such as gripping or welding.
- **Theory**: The **transl** function creates a 4x4 homogeneous transformation matrix representing a translation. In robotics, such matrices are used to describe the position and orientation of robot links or tools in 3D space.

Creating Twists for Different Work Modes

- **Code**: Twists **T1_A**, **T2_A**, **T3_A** for Work Mode A (Translation) and **T1_B**, **T2_B**, **T3_B** for Work Mode B.
- **Justification**: These lines define the twist vectors for two different work modes. Twists are a compact way to represent both rotational and translational velocities in 3D space. For Work Mode A (Translation), the twists are set to represent pure translations along the x, y, and z

axes, respectively. For Work Mode B, the twists represent rotation around the x and y axes and translation along the z-axis.

- **Theory**:
    - **Twist Vectors**: In robotics, a twist vector is a six-element vector where the first three elements represent angular velocity (for rotations) and the last three represent linear velocity (for translations). The concept of twists is derived from screw theory and is used extensively in robotic motion planning and control.
    - **Work Modes**: Different operational modes of a robot require different types of motions. By defining distinct sets of twists for each work mode, our script can adapt the robot's motion based on the current task requirements.

Updating the Jacobian Matrix

- **Code**: **J = ur3.jacob0(q);**
- **Justification**: This line calculates the Jacobian matrix of the UR3 robot at a given joint configuration **q**. The Jacobian is a fundamental concept in robotics, especially for manipulators like the UR3. It is crucial for understanding how changes in joint angles affect the position and orientation of the robot's end-effector.
- **Theory**: The Jacobian matrix is a mathematical representation that links the velocities of the robot's joints to the linear and angular velocities of its end-effector. In simple terms, it tells us how fast the end-effector is moving (both in translation and rotation) for a given set of joint velocities. This is vital for tasks like path planning, motion control, and inverse kinematics.

Calculating Transformation Matrices

- **Code**: **Tbase_tcp = ur3.fkine(q);** and **Ttcp_base = inv(Tbase_tcp);**
- **Justification**:
    - **Tbase_tcp = ur3.fkine(q);**: This line computes the forward kinematics of the robot, resulting in a transformation matrix (**Tbase_tcp**) that describes the pose of the tool center point (TCP) relative to the base frame, given the current joint angles (**q**).
    - **Ttcp_base = inv(Tbase_tcp);**: This line calculates the inverse of the transformation matrix, providing the transformation from the TCP back to the base frame.
- **Theory**:
    - **Forward Kinematics**: Forward kinematics in robotics is the process of determining the position and orientation of the robot's end-effector based on its joint parameters. It's a fundamental concept for understanding how a robot's configuration affects its pose in the workspace.
    - **Transformation Matrix**: A transformation matrix in robotics is a 4x4 matrix used to describe the spatial relationship between two frames of reference. It encompasses both rotation and translation components, allowing for comprehensive spatial transformations.
    - **Inverse Transformation**: The inverse of a transformation matrix is used when we need to calculate the pose of one frame relative to another in the opposite direction. For instance, finding out the base frame's position relative to the end-effector, which is useful in certain control and planning algorithms.

Calculating Desired Differential Motion (Twist)

- **Code**:
    - For Translation: **T1 = T1_A'; T2 = T2_A'; T3 = T3_A'; scale = 0.07; scale_B = 0.07;**
    - For Insertion: **T1 = tr2jac(Ttcp_base)*T1_B'; T2 = tr2jac(Ttcp_base)*T2_B'; T3 = tr2jac(Ttcp_base)*T3_B'; scale = 0.1; scale_B = 0.07;**
- **Justification**:

    - For Translation Mode (work_mode == 0), the script is setting the desired twists to represent translations along the x, y, and z axes with a scaling factor.
    - For Insertion Mode (work_mode == 1), it transforms the twists according to the TCP to base frame transformation. This change in reference frame is crucial for tasks like insertion, where the motion is relative to the tool's orientation.
- **Theory**:
    - **Twist Vectors**: A twist vector in robotics combines linear and angular velocities and is essential for representing general 6D motion.
    - **Transformation of Twists**: The **tr2jac** function converts a transformation matrix to a Jacobian matrix, which is used to transform twists from one frame to another. This is important in scenarios where the motion needs to be expressed relative to a different frame, such as the tool frame.

Associating Twist with Joystick Inputs

- **Code**: **vd = (scale * s1 * T1 + scale * s2 * T2 + scale_B * s3 * T3);**
- **Justification**: This line calculates the desired velocity (twist) **vd** by scaling and summing the individual twists based on joystick inputs (**s1**, **s2**, **s3**). It effectively maps user inputs to robot motions, allowing for intuitive control of the robot.
- **Theory**: This approach is typical in teleoperation or manual control scenarios where the operator uses a joystick or similar device to control the robot. The scaling factors adjust the sensitivity of the control.

Computing Joint Velocities

- **Code**: **q_dot = pinv(J) * vd;**
- **Justification**: Here, the script calculates the joint velocities **q_dot** required to achieve the desired end-effector velocity **vd** using the pseudo-inverse of the Jacobian **J**. This is a standard method in robotics for solving inverse kinematics problems, especially when the Jacobian is not square or the system has redundancy.
- **Theory**: The pseudo-inverse provides a least-squares solution to the inverse kinematics problem, effectively finding the most efficient joint movements to achieve the desired end-effector motion.

Checking and Normalizing Joint Velocities

- **Code**: **if norm(q_dot) > 1 q_dot = q_dot / norm(q_dot); end**
- **Justification**: This safety check ensures that the calculated joint velocities do not exceed a certain magnitude (in this case, 1). If the magnitude is greater than 1, the velocities are normalized. This prevents the robot from moving too fast, which could be unsafe or beyond the robot's physical capabilities.
- **Theory**: In robotics, maintaining joint velocities within safe limits is crucial for the longevity of the hardware and safety of the operation. Normalizing velocities is a straightforward and effective method to control this aspect.

Conclusion

Our MATLAB script represents a sophisticated integration of several key concepts in robotics, blending practical implementation with theoretical knowledge. Here's an overview of the major aspects:

Integration with ROS and Real-time Control

- Our script effectively utilizes the ROS (Robotics Operating System) for real-time data communication, demonstrating an understanding of modern robotic systems' architecture. This includes creating a ROS node, subscribing to necessary topics, and publishing commands, which are essential skills in robotic system integration.

Kinematic Modeling and Transformation Calculations

- The script includes the creation of a kinematic model for the UR3 robot, a widely used manipulator in both research and industry. Understanding and setting up this model is crucial for any robotic control system.
- The calculation of transformation matrices and their inverses shows a good grasp of spatial relationships within robotic frameworks, essential for accurate positioning and orientation of the robot's end-effector.

Motion Planning and Control

- Defining and manipulating twist vectors for different work modes (translation and insertion) showcases the script's versatility in handling different types of robotic motions.
- The use of Jacobian for converting desired end-effector velocities to joint velocities, and the application of pseudo-inverse for solving inverse kinematics, are indicative of a deep understanding of motion planning and control in robotics.

User Interaction and Safety Measures

- Incorporating joystick inputs for real-time control reflects an application-oriented approach, aligning robotic movements with user commands. This aspect is significant in teleoperated or manually controlled robotic systems.
- Implementing safety checks like velocity normalization indicates a conscientious approach towards maintaining the robot's operational integrity and safety.

Final Thoughts

Overall, our MATLAB script has overall impressive demonstration of robotic principles in action. It combines theoretical knowledge in kinematics, control systems, and user interaction with practical application skills in a ROS-enabled environment. This script could serve as a robust foundation for further development in various robotic applications, ranging from automated tasks to more complex, user-interactive scenarios. Our work not only addresses the functional aspects of robotic control but also integrates essential safety and usability considerations, vital in the field of robotics.