FUNDAMENTALS OF ROBOTICS

Dorijan Di Zepp, Stefano Camposilvan, Pietro Giannini, Pier Guido Seno

TASK PLANNER

The high-level planning technique used consists of a path planning in the operational space, where the initial point is the initial pose of the end effector, and then a series of intermediate points are passed through to finally end up in a predetermined final end effector pose. All this has been implemented with a state machine composed of 10 states, where the majority of them represent a point to reach and therefore an end effector pose to be sent via a service to the motion planner node.

1. **START**: The first state consists of receiving the different poses of the blocks and their class from the vision node via a service. These values are then used to assign the final poses of the end effector for each block based on its class and to generate the final poses of the end effector for each corresponding block pose. (The poses must be converted from the world frame to the robot frame before being sent to the motion node.)

2. **BLOCK REQUEST**: Each time this state is entered, it moves to the next block to be picked, determining the pose for grasping the block and the release pose based on the current block class.

3. **HIGH BLOCK TAKE:** From this state onwards, each state corresponds to an end effector pose sent to the motion node via a service until returning to the block request state where the next block will be passed. This state is used to send via service the pose that corresponds to an elevated block position (same pose as block take but with z position = 0.55 in Robot Frame) to avoid collisions with the table and the blocks on it during arm movement. These intermediate points (high states) are used to avoid some geometric constraints such as obstacles like the blocks placed on the table during the movement between grasping and releasing the blocks. In this state, the gripper is open because the next state will be BLOCK TAKE, which is the grasping of the object.

4. **BLOCK TAKE**: This state is used to move from the intermediate point HIGH BLOCK TAKE to grasping the object, and thus a final pose for block grasping will be passed to the motion node via service.

5. **HIGH BLOCK RELEASE**: From this state, the release part of the block begins, i.e., moving the block from its original position to the position corresponding to its class. The pose sent to the motion node is the same as the HIGH BLOCK TAKE state, the only difference being that before moving the block, the gripper is closed.

6. **HIGH CLASS RELEASE**: Before releasing the block in the designated position, the end effector is brought to an elevated position relative to the block's class (z=0.55 in Robot Frame) to avoid collision with obstacles.

7. **CLASS RELEASE**: In this state, the block is brought to the corresponding predetermined class position based on the block's class.

8. **HIGH CLASS TAKE**: From this state, the operation of grasping the next block begins if available. First, the gripper is opened to release the block, and then the end effector is brought to an elevated position. Additionally, in this state, it is determined if there are still blocks available to be picked, and if not, the future state is selected as NO MORE BLOCKS to terminate the process in a safe predetermined final position. If there are other blocks to be picked, the future state will be BLOCK REQUEST, where the grasping and releasing of the next block will be planned.

9. **MOTION ERROR**: This state is used in case of an error in the motion node during the calculation of the trajectory from the current pose to the future one just sent. This can happen for various reasons:

    - Initial unreachable position: The point sent to the motion planner is unreachable, and therefore it moves to the next block. This event should not occur but handles the case where a block is generated outside the working space. This case is directly handled by the current states and not in the motion error state (a check is made of the error variable value returned in the service by the motion planner, and this occurs for each node that sends an end effector pose to the motion node).

    - Motion error: There are two available implementations due to the two singularities management.

        1. DYNAMIC DAMPING FACTOR: In this case, singularities are managed during the motion phase using the technique of dynamic damped pseudo-inverse. Control then occurs on the final position calculated by the motion planner, and if it is outside the working space, an intermediate safe homing pose is sent before, which serves in most cases to avoid a shoulder singularity (safe position = (0,-0.44,previous z) in Robot Frame). We use this pose to prevent the end effector from passing near the axis of the base of the UR5 robot arm.

        2. INTERMEDIATE POINT: In this case, singularities are not managed by a damped pseudo-inverse but the end effector is directly passed through a safe point (equal to the one described above) so as not to end up in a shoulder singularity.

10. **NO MORE BLOCKS**: This state is used to terminate the process and to move the end effector to a final safe pose.

In general, therefore, the task planner is responsible for designing a path planning acting as an intermediary between the vision node, from which it will receive information about the blocks, and the motion node, which will translate the poses sent by the task planner into the joint space to send the configurations via a topic to the UR5 node in Gazebo.
The documentation created by Doxygen highlights the presence of numerous values, including: block retrieval height, gripper opening, effective workspace, and others. All these values have been calculated directly using the tools provided by rViz and refined through tests. For each block class, there will be a different set of values assigned to the variables GRIPPER_CLOSE, GRIPPER_OPEN, and GRASPING_HEIGHT based on the dimensions of the block.

MOTION PLANNER

The module dedicated to motion control is responsible for calculating the trajectory from a defined starting point to a destination point, managing singular situations and positions that are not allowed or unreachable. In this specific circumstance, we had several options available, including implementing inverse differential kinematics using quaternions or Euler angles, and the interpolation method for points.

We opted for the implementation of inverse differential kinematics using quaternions, as this choice allows for easier control of singularities caused by critical wrist configurations. Although interpolation would have been a valid alternative, it proved more suitable for complex trajectories such as curves. The solution based on differential kinematics simply employs linear interpolation between two points, as it did not require the use of higher-degree polynomials.

The proposed solution involves an initial check to verify that the indicated final position is within the workspace, which is delimited by the surface of the table itself. A further check involves verifying that, through inverse kinematics, the final position can actually be reached using a specific joint configuration. If either of these checks returns a negative outcome, the motion planner would not compute any trajectory and would return an error code to the position task planner indicating an unreachable position.

In the case of a positive outcome, a trajectory would be calculated, with the initial point being the current configuration of the manipulator, using a specific topic for reading joint values, direct kinematics to calculate the end-effector pose, and a function to convert Euler angles of the initial and final positions into quaternions.

Once the initial and final values of the trajectory are obtained, it is computed as a set of joint configurations (joint space) defined over a time interval. These configurations are calculated as the product of the pseudo-inverse Jacobian matrix, in cases where it is invertible, and the velocity vector, defined by SLERP functions (for angular velocity) and linear interpolation (for linear velocity).

Any deviations from the optimal trajectory, calculated as the difference between the current position and the calculated one, are scaled with a multiplicative factor to reduce this error over time.

After calculating the trajectory, the various configurations are sent via a specific topic using a message, and the motion planner returns a message indicating successful movement.
In the event of errors during trajectory calculation, similar to what was done for workspace control, the manipulator remains in its original position, and an error code is returned to the motion error task planner.

In cases of singular situations where the Jacobian matrix is not invertible, rendering the use of the pseudo-inverse impossible, we had to address one of the most critical issues, requiring the implementation of additional solutions to handle them.

We developed several solutions, with some proving more effective than others. Below is an analysis of the tested solutions:

Static damping factor:

In this solution, the determinant of the Jacobian matrix is calculated, and a threshold determines whether to use the pseudo-inverse matrix or the damped pseudo-inverse matrix, allowing deviation from the calculated trajectory near singularities with a damping factor. During testing, various thresholds and damping values were evaluated to minimize deviation from the trajectory. Eventually, we decided to abandon this solution because different situations required different damping values.

Dynamic damping factor:

Building upon the previous solution, it was decided to try using a damping factor that varies based on the current configuration of the manipulator, particularly based on the eigenvalues of the Jacobian matrix. Although this solution provided better results than the previous one, rare cases were encountered where the deviation from the trajectory was high or the singularity was not resolved.

$$eigenvalues = \left(J \cdot J^{\top}\right).eigenvalues$$

$$dampingFactor = \frac{MAX\_DAMPING\_FACTOR}{eigenvalues.maxCoeff}$$

$$J = J^{\top} \cdot \left(J \cdot J^{\top} + \left(dampingFactor^2 \cdot I_{6x6}\right)^{-1}\right)$$

Intermediate point:

The final solution was to resort to a safe intermediate point in the presence of severe singularities defined by a specific threshold. Although not the most elegant solution, as it involves adding an additional point, it proved to be the most robust in handling singular situations, albeit increasing the time required to reach the final destination.

From these solutions, it was decided to maintain the possibility, through appropriate compilation directives, to use both the dynamic damping factor and the intermediate point. Although the first solution does not work in all cases, we wanted to highlight a possible valid solution, which, due to timing constraints, could not be optimized to the fullest.

DATASET CREATION

To create the dataset, we started with the provided images, which included a background and a table with various types of Lego blocks on top.

Each type of block had a specific name (e.g., X1-Y1-Z2, X1-Y2-Z1, X1-Y2-Z2, X1-Y2-Z2-CHAMFER, X1-Y2-Z2-TWINFILLET, X1-Y3-Z2, X1-Y3-Z2-FILLET, X1-Y4-Z1, X1-Y4-Z2, X2-Y2-Z2, X2-Y2-Z2-FILLET), and each class had geometric characteristics different from the others.

For each image, we created a text file in Darknet format containing the class type and the bounding box of each block. We uploaded all this information, along with a file containing class definitions, to Roboflow to create the dataset. Once everything was uploaded and the dataset was created, we exported it in YOLOv5 Format.

DATASET TRAINING

To train the dataset, we used Google Colab, changing the runtime type and utilizing the GPU. We configured the working environment to use the YOLOv5 framework.

Cloning the repository was necessary to obtain the source code, and the installation of requirements ensured that all necessary libraries were present in the system.

Subsequently, we downloaded the annotated dataset from Roboflow, specifically in the format required by YOLOv5.

To conclude, we ran a script to train the model on the provided dataset. Before executing this script, we specified the number of epochs, image dimensions, and the path to the configuration file containing dataset information. Once everything was finished, we downloaded 'best.pt,' containing the best weights obtained during the model training process.

VISION

In order to be able to correctly identify and localize the blocks, the vision node takes the view of the table, as an RGB image, and the PointCloud data from the Zed camera.
Then, by loading a yolov5 pre-trained machine learning model, we are able to get information about the blocks such as their bounding boxes' coordinates (x, y, z and h) and the name of their classes.

After obtaining and storing this information, we proceed by focusing on one block at time.
For each block, we crop the image in order to only have the section inside the block's bounding box, and then, by applying a color-based masking technique, we isolate the block's parts in the cropped image, whose color is red, from the background parts, which are gray, to obtain block's 2D points.

After doing so, by using the ROS function "read_points", we convert these points into the corresponding 3D points, and then we transform these 3D points, which are relative to the Zed camera's frame, into the World frame, using the camera's known position and orientation.

Once we have all of the block's 3D points, we have to obtain its pose and central point's coordinates; to do so, we decided to find, first, three vertices of the block's lower surface: the one with the lowest value of x among the points with the lowest value of y, A, the one with the lowest value of x among the points with the highest value of y, B, and the point with the lowest value of y among the points with the lowest value of y, C.

Having obtained the three vertices, we then calculate the distance between A and C: if the value is high enough, we can state that the two points are not the same and, so, that the angle phi between the block and the x-axis is not zero, otherwise it is zero.

If the two points do not coincide, we determine which point, between A and B, has an higher value of x: if A is higher, the block is rotated of phi around the z-axis in the clockwise direction, while if B is higher, the block is rotated of phi around the z-axis in the counter-clockwise direction.
We then calculate phi using trigonometry; in the first case, that's the final value, while in the second case, we have to subtract $\pi/2$ to phi, so as to have it be the right rotation for the block from our interested point of view.

In the end, to find the block's center point coordinates, and therefore to localize the block, we use the C vertex found above, along with the known values for the block's sides and pose, and by applying trigonometry, we are able to achieve our goal.