COMP462
Project 1: Non-prehensile Manipulation Planning

In this project, you are asked to practice a motion planner for physics-involved manipulation planning. Specifically, we consider manipulation tasks where the robot needs to rearrange the surrounding movable objects by pushing them to achieve certain goals, as illustrated in Figure 1. You need to implement the algorithm of a Kinodynamic Rapidly-exploring Random Tree (RRT) for solving such tasks.

The kinodynamic motion planning problems involved in this project are standardized with the following components:

1. State space: Under the quasi-static assumption (i.e., the motion of the robot and the objects are slow enough such that the inertial forces are neglectable), the state space of the problem is defined by the Cartesian product of the robot's joint space and the state space of all the movable objects in $SE(2)$. For example, if there are two movable objects in the workspace, the state space will be $7 + 2 \times 3 = 13$ dimensional, since the robot has 7 joints and each object adds 3 more dimensions for its position and orientation in $SE(2)$ (i.e., x, y, and θ).
2. Control space: We define the control in the Cartesian space (the end-effector's space). Specifically, the control includes the end-effector's x-y linear velocity $\dot{x}$, $\dot{y}$ and angular velocity about z-axis $\dot{\theta}$ (the velocities in a 2D space), associated with the duration of the control d. We represent a control by the vector $[\dot{x}, \dot{y}, \dot{\theta}, d]$. To execute such a control on the robot manipulator, we need to project it onto the robot's joint space, which will be described in Task 1.
3. Physics law: The physics of the whole robot-object system is represented by a propagate function. The propagate function takes the current system state and a control as inputs and will infer the state at the next step. In this project, rather than analytically deriving the propagate function, we use the simulation to simulate it.

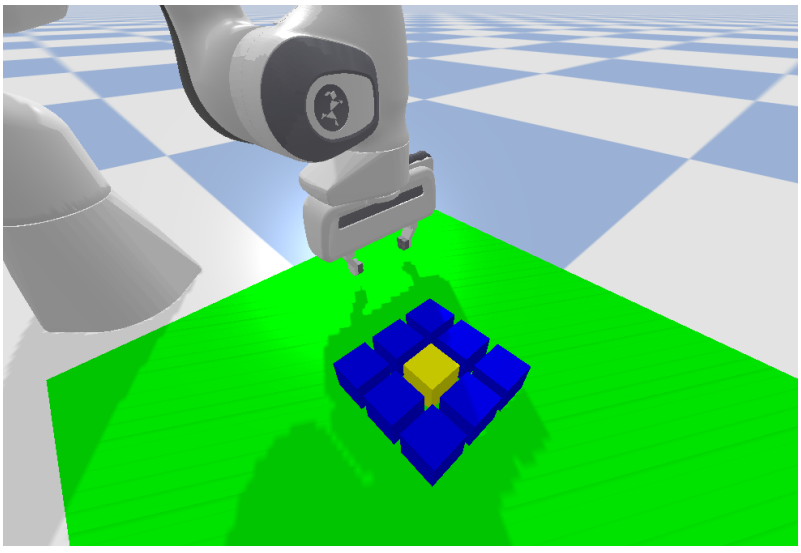


Figure 1: The illustration of a 7-DoF robot manipulator rearranging the surrounding cubic objects in `pybullet` simulator.

Task 1: Jacobian-based Control Projection

To generate more effective pushing motions, the control space is defined in the Cartesian space, as stated above. And the robot's motion is constrained such that its end-effector only moves parallel

to the ground plane. To perform the control defined as such, while executing the control, you need to iteratively project the Cartesian velocity to the robot's joint space and command the robot with the projected joint velocities.

In this part, you are required to numerically calculate the Jacobian matrix of the robot and use it to project a Cartesian velocity to get the robot's joint velocity for control.

Below are the detailed guidelines:

1. First, you need to implement the following member method of the `JacSolver` class in `jac.py` for calculating the jacobian matrix of the robot:

```
def get_jacobian_matrix(self, joint_values)
    """
    Numerically calculate the Jacobian matrix based on joint angles.
    args: joint_values: The joint angles of the query configuration.
                        Type: numpy.ndarray of shape (7,)
    returns:         J: The calculated Jacobian matrix.
                        Type: numpy.ndarray of shape (6, 7)
    """
```

   Given that the robot has 7 degrees of freedom, the Jacobian matrix should be 6×7 and relate the Cartesian velocities (the twist of the end-effector) and the robot's joint velocities by the following equation:

$$\begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{\omega} \end{bmatrix} = J(\mathbf{q})\dot{\mathbf{q}}$$

   where the notations are

   (a) $\boldsymbol{v} = [v_x, v_y, v_z]^\top$ is the linear velocity of the end-effector in 3D space;

   (b) $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^\top$ is the angular velocity of the end-effector, which can be interpreted as rotating about the axis $\frac{\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|}$ at a rate of $\|\boldsymbol{\omega}\|$;

   (c) $\mathbf{q}$ is a 7-dimensional vector of the robot's joint angles, and $\dot{\mathbf{q}}$ is a 7-dimensional vector of the robot's joint velocities;

   (d) $J(\mathbf{q})$ is the 6×7 Jacobian matrix when the robot is at the configuration $\mathbf{q}$.

   You are expected to compute the Jacobian matrix column-wise by finite differences. Remember that each column of the Jacobian matrix is associated with one robot joint. You can use the provided member function `JacSolver.forward_kinematics()` for the robot's forward kinematics if needed.

   For implementing this part, you might find it sometimes necessary to do kinematics-related calculations, e.g., transformations, conversions between different orientation representations, etc. Feel free to use libraries like `PyKDL` (http://docs.ros.org/en/diamondback/api/kdl/html/python), or transformation-related functions provided by `pybullet` (https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit#heading=h.y6v0hy52u8fg).

2. Next, given a control $[\dot{x}, \dot{y}, \dot{\theta}, d]$ on the end-effector ($\dot{x}$ and $\dot{y}$ for x- and y- linear velocities respectively, $\dot{\theta}$ for angular velocity about z-axis, and d for the duration), you need to iteratively evaluate the Jacobian matrix and use the Jacobian matrix to calculate the projected joint velocities for executing this control.

   It is worth noting that this control only specifies values for some dimensions of the desired Cartesian velocity, not all dimensions. Therefore, it is needed to construct the complete desired Cartesian velocity (twist of the end-effector) by setting the unspecified dimensions to zero, i.e., $[\dot{x}, \dot{y}, 0, 0, 0, \dot{\theta}]^\top$.

   For this, you need to implement the `TODO` in the following function `PandaSim.execute()` in `sim.py`:

```
def execute(self, ctrl, sleep_time=0.0):
    """
    Control the robot by Jacobian-based projection.
    args:       ctrl: The robot's Cartesian velocity in 2D and its
                      duration - [x_dot, y_dot, theta_dot, duration]
                      Type: numpy.ndarray of shape (4,)
          sleep_time: sleep time for slowing down the simulation
                      rendering. (you don't need to worry about this
                      parameter)
    returns:    wpts: Intermediate waypoints of the Cartesian trajectory
                      in the 2D space.
                      Type: numpy.ndarray of shape (# of waypoints, 3)
    """
```

Basically, you need to evaluate the Jacobian matrix by calling `self.get_jacobian_matrix()` and use it to calculate the projected joint velocities. As required, the joint velocities should be represented by a `numpy.ndarray` of shape $(7,)$, and stored in the variable `vq`.

Besides these, you can try whatever you think might be helpful for more stably and efficiently controlling the robot, for example,

(a) avoid singularity of the robot

(b) avoid the robot to reach its joint limits

(c) monitor the end-effector's position and stop moving the robot when the end-effector is too far away from the workspace, etc.

After completing this part, you can run the following command in your terminal to test your program: `python main.py --task 1`

If your Jacobian calculation and projection are implemented correctly, you should see the robot's end-effector move along a square parallel to the ground plane while rotating itself about z-axis for 10 times, as illustrated in Figure 2. We will evaluate how well by your calculation the end-effector moves along the desired 2D trajectory (the square), and you can see the averaged Cartesian waypoint position and orientation errors in the terminal.

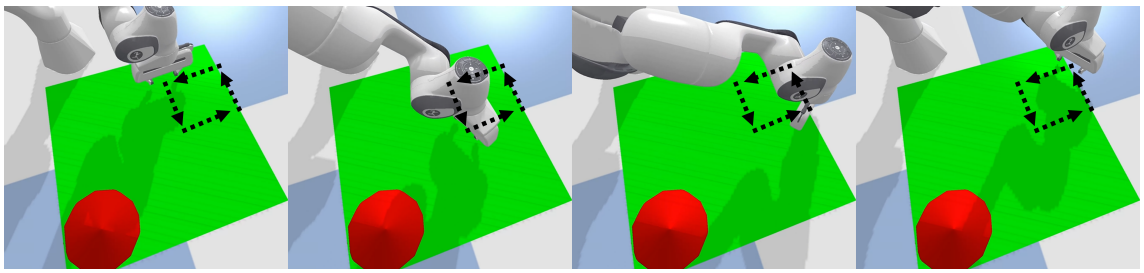


Figure 2: The illustration of the expected robot motions in Task 1. The end-effector moves along a square trajectory shown by black arrows while rotating itself about z-axis.

Task 2: Kinodynamic Motion Planning

In this part, you need to complete the main algorithm of a Kinodynamic RRT. The task of the robot is to push the yellow cube into the red goal region.

In the program, the state of the simulation is represented by a dictionary, where `"stateID"` maps to its ID (an `int`) and `"stateVec"` maps to a `numpy.ndarray` containing its state values. For a `state`, the first 7 values of `state["stateVec"]` (i.e., `state["stateVec"][0:7]`) are the joint angles of the robot, and the rest values are the poses of other objects in $SE(2)$.

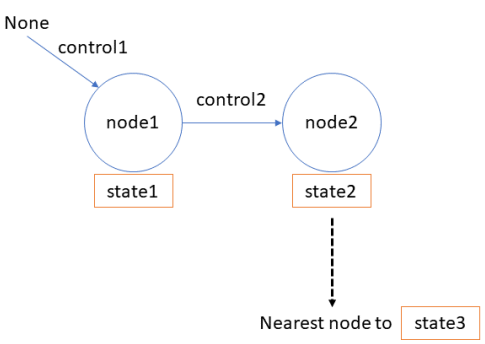
Below are the detailed guidelines for this part:

1. First, you need to implement the state validity checker for the problem. You are provided with a class `ProblemDefinition` in `pdef.py`, by which you can access any information relevant to the problem, for example, the simulation instance, the goal, the bounds for state and control, etc. You are asked to finish the following member method for the state validity checker:

```
def is_state_valid(self, state)
    """
    Check if a state is valid or not.
    args: state: The query state of the system.
                 Type: dict, {"stateID": int, "stateVec": numpy.ndarray}
    returns: Ture or False
    """
```

   Hints: You can call `self.bounds_state.is_satisfied(state)` to check if `state` satisfies the bounds of the state space. And you can use `self.panda_sim.is_collision(state)` to check if the simulation instance has any collision. And please feel free to add any other state validity criterion that you believe can facilitate your planner.

2. Next, You need to implement the main algorithm of the Kinodynamic RRT. Please first read and understand the provided codes of the classes `Tree` and `Node` in `rrt.py`. They relate to the data structure that will be used by the Kinodynamic RRT planner. Below is a quick example of how to declare `Node` instances, set its associated controls, set its parent node, and add nodes to an instance of `Tree`, as illustrated in Figure 3.




```
# add node1 into the tree (an instance of Tree)
node1 = Node(state1)
node1.set_control(control1)
node1.set_parent(None)
tree.add(node1)

# add node2 into the tree and set its parent to node1
node2 = Node(state2)
node2.set_control(control2)
node2.set_parent(node1)
tree.add(node2)

# find the nearest node in the tree to state3
node_nearest = tree.nearest(state3["stateVec"])
```

Figure 3: A basic example of how to use `Node` and `Tree`.

Now, you need to implement the following method of `KinodynamicRRT` class in `rrt.py`. It should return `True` and the Cartesian plan (a list of `Node`) if the planner finds the solution within the given time budget, otherwise, it should return `False` and `None`.

```
def solve(self, time_budget)
    """
    The main algorithm of Kinodynamic RRT.
    args:  time_budget: The planning time budget.
    returns:    solved: True or False.
                  plan: The motion plan found by the planner,
                        represented by a sequence of tree nodes.
                        Type: a list of rrt.Node
    """
```

You are recommended to read the provided codes of other relevant classes (`Motion`, `Tree`, `StateSampler`, `ControlSampler`, etc) that might be called. Below are some notes that could be useful:

(a) You can call `self.state_sampler.sample()` to randomly sample a state vector of `numpy.ndarray` within its bounds. And you can call `self.control_sampler.sample_to(node, stateVec, k)` to sample k candidate controls from `node` towards `stateVec`. It has two returns: 1. the best control among candidates whose outcome state is nearest to `stateVec`; 2. the outcome state associated with this best control.

(b) You can call `self.pdef.goal.is_satisfied(node)` to check if `node` satisfies the goal.

After completing this part, you can run the following command in your terminal to test your program: `python main.py --task 2`

You will be able to see the robot fast simulate its motions for finding a solution. And after a solution plan has been found, you will see the robot slowly execute the solution plan. Meanwhile, the end-effector's trajectory will be drawn by red lines, as illustrated in Figure 4.

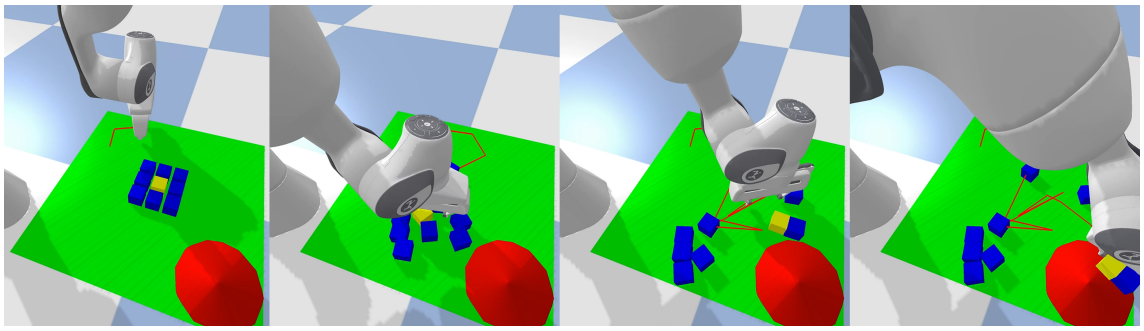


Figure 4: Example of the robot executing a solution plan to push the yellow target object into the red goal region. The end-effector's trajectory is drawn by red lines.

Task 3: Grasping in Clutter

In this part, you need to implement your own criterion function for a grasping goal. You can refer to the provided implementation for the relocating goal in Task 2, which is located at `RelocateGoal` in `goal.py`

Please complete the following method of class `GraspGoal` in the same file. The `GoalGrasp` has been provided with a member variable `self.jac_solver` of the type `jac.JacSolver()`.

```
def is_satisfied(self, motion):
    """
    Check if the state satisfies the GraspGoal or not.
    args: state: The state to check.
                 Type: dict, {"stateID": int, "stateVec": numpy.ndarray}
    returns: True or False.
    """
```

Hint: You can check if the target object is close enough to the end-effector and if their orientations are well aligned. For a `state`, you can access the pose of the target object in $SE(2)$ (i.e., x, y and θ of the target object) by `state["stateVec"][7:10]`.

The position of the robot base is $[-0.4, -0.2, 0]$, and the orientation of the robot base is zero.

After completing this part, you can run the following command in your terminal to test your program: `python main.py --task 3`, which will run your algorithm implemented in Task 2 for the new grasping task. If your grasping goal is implemented correctly, you should be able to see the end-effector approach the target object while the solution plan being executed and successfully grasp the target object with its fingers.

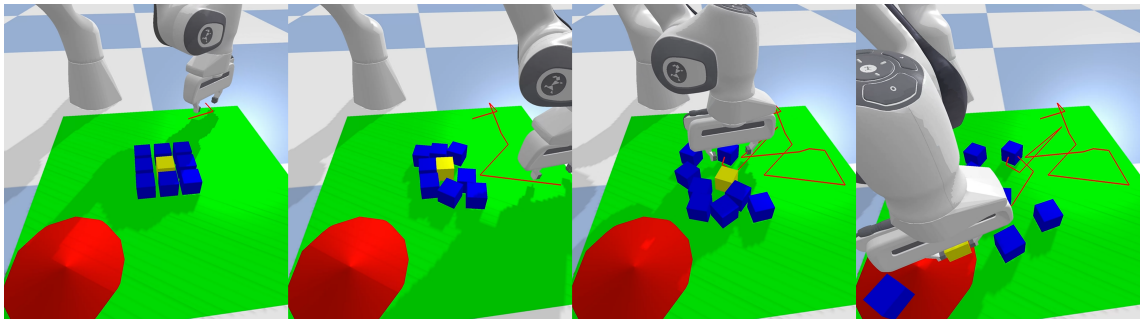


Figure 5: Example of the robot executing a solution plan to approach the yellow target object for grasp. In the end, the end-effector will grasp the target object by closing its fingers. The end-effector's trajectory is drawn by red lines.

(Optional) Task 4: Trajectory Optimization

This part asks you to implement some trajectory optimization techniques for a relocating or grasping task. Please feel free to implement your own functions in `opt.py` and call your function under `TO DO` in `main.py` to run your methods. You can do anything relevant, for example, stochastic trajectory optimization [1]. Just show what you have experimented with and found in your report.

References

[1] Wisdom C Agboh and Mehmet R Dogar. Real-time online re-planning for grasping under clutter and uncertainty. In *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 1–8. IEEE, 2018.