# Project 1: Non-prehensile Manipulation Planning

In this project, we will implement a physics-based planner for nonprehensile tabletop manipulation. Specifically, we consider manipulation tasks where the robot needs to rearrange some surrounding movable objects by pushing so that certain task goals will be achieved. An example task environment is illustrated in Figure 1. For this, among other possibilities, we will implement the algorithm of Kinodynamic Rapidly-exploring Random Tree (Kinodynamic RRT) .

We will use a 7-DoF Franka Emika Panda robot simulated in PyBullet. A code framework, hosted on github (https://github.com/robotpilab/COMP462/tree/main/project1), is provided to facilitate the implementation. . The kinodynamic motion planning problems considered in this project are modeled by three major 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 $\theta$).

2. Control space: We define the control in the Cartesian space of the robot end-effector. Specifically, the control consists of the end-effector's linear velocity $(\dot{x}, \dot{y})$, and angular velocity about the z-axis $\dot{\theta}$, 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, as will be described in Task 1.

3. Physics law: The physics of the entire robot-object-environment system is commonly represented by a propagation function. The propagation function takes the current system state and a control as inputs and will infer the state at the next time step. In this project, rather than analytically deriving the propagation function, we will use the physics engine in the simulator as our propagation function.

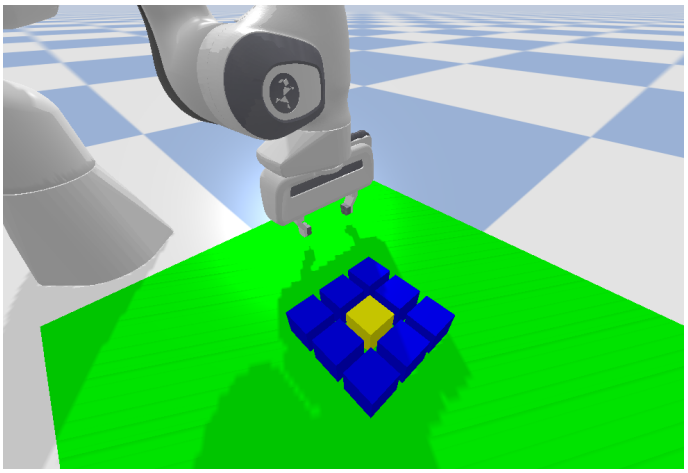


Figure 1: A 7-DoF robot manipulator rearranging cubic objects in `PyBullet` simulator.

## Task 1: Jacobian-based Control Projection

To more effectively represent and plan pushing-based manipulation motions, the control is defined in the Cartesian space of the end-effector as stated above. As such, the robot's motion is constrained such that its end-effector only moves parallel to the plane where manipulation happens. To execute

such end-effector motions with a robot arm, one needs to iteratively project the Cartesian controls to the robot's joint space and command the robot with the projected joint velocities.

Instead of analytically calculating the time derivatives of the forward kinematics, the first task in this project is to numerically calculate the Jacobian matrix of the robot and use it to project the Cartesian velocity controls of the end-effector to the robot's joint velocities that can be used to move the robot arm. Major steps for this task are detailed below.

1. 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 $6 \times 7$ 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 $6 \times 7$ Jacobian matrix when the robot is at the configuration $\mathbf{q}$.

   The Jacobian matrix should be computed column-wise by finite differences. Remember that each column of the Jacobian matrix corresponds to one robot joint. The provided member function `JacSolver.forward_kinematics()` for the robot's forward kinematics can be used in this implementation if needed.

   For implementing this part, it might be useful 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 $[\dot{x}, \dot{y}, \dot{\theta}, d]$ for 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), iteratively evaluate the Jacobian matrix and use it to calculate the projected joint velocities for executing this control on the 7-DoF robot arm. In theory, the Jacobian needs to be updated as often as possible, i.e., with high update resolution, to ensure smooth execution of the constrained motions. Please read the code to understand how often is the Jacobian updated in the provided framework.

   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, the Jacobian matrix of the robot needs to be evaluated iteratively by calling `self.get_jacobian_matrix()`, and be used to calculate the projected joint velocities. The joint velocities should be represented by a `numpy.ndarray` of shape $(7,)$, and stored in the variable `vq`.

Besides, feel encouraged to try additional tricks in this function that might be helpful for more stably and efficiently planning and controlling the robot motions, as will be seen in the other tasks that follow. For example, the tricks can be about:

(a) avoiding singularity of the robot;

(b) avoiding the robot to reach its joint limits;

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

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

If both the Jacobian calculation and the control projection are implemented correctly, you should see the robot's end-effector moving along a square path parallel to the ground plane while rotating itself about the $z$-axis for 10 times, as illustrated in Figure 2. The quality of robot motion, defined by the accuracy of the Jacobian-based control, will be evaluated by the Cartesian differences between the executed and the desired motion trajectories. The averaged Cartesian motion errors are printed in the terminal standard output.

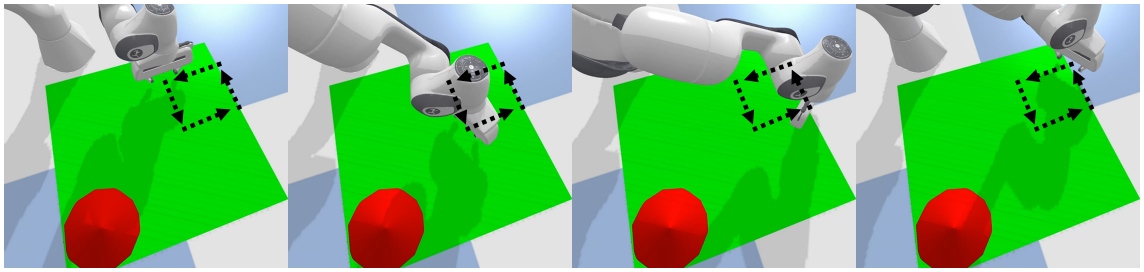


Figure 2: An illustration of the expected robot motions in Task 1. The end-effector moves along a square path shown by black arrows while rotating itself about the $z$-axis.

## Task 2: Kinodynamic Motion Planning

In this part, we will implement 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 provided code framework, the state of the robot-object system is represented by a dictionary consisting of 2 key-value pairs: 1) `"stateID"` maps to an unique ID (`int`) of a state; 2) `"stateVec"` maps to the state values (`numpy.ndarray`). For a `state`, the first 7 values of

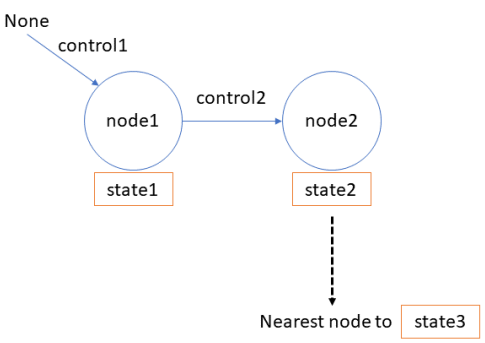
`state["stateVec"]` (i.e., `state["stateVec"][0:7]`) are the joint angles of the robot, and the rest values are the poses of all objects in $SE(2)$. Major steps for this task are detailed below.

1. A class `ProblemDefinition` is provided in `pdef.py`. This class can be used to access any information relevant to the manipulation problem, including, for example, the simulation instance, the goal, the bounds for state and control, etc. Finish the following member method of `ProblemDefinition` for the state validity checker that will later be used by the manipulation planner:

```
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: `self.bounds_state.is_satisfied(state)` can be called to check if a `state` satisfies the bounds of the state space. `self.panda_sim.is_collision(state)` can be called to check if the simulation instance has any collision. Feel free to add any other state validity criterion that are potentially going to facilitate the planner.

2. Implement the main algorithm of the Kinodynamic RRT. Please first read and understand the provided 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, implement the following method of `KinodynamicRRT` class in `rrt.py`. It should return `True` and the motion plan (a list of `Node`) if the planner finds the solution within the given time budget (in seconds), 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 (in seconds).
    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
    """
```

It is strongly recommended to read the provided code of other relevant classes (`Node`, `Tree`, `StateSampler`, `ControlSampler`, etc) that might be called. Some hints for implementing the planner:

(a) `self.state_sampler.sample()` can be used to sample a state vector within the bounds, and `self.control_sampler.sample_to(node, stateVec, k)` can be called to sample $k$ candidate controls from `node` towards `stateVec`. The latter 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) `self.pdef.goal.is_satisfied(node)` can be called to check if a `node` satisfies the goal.

After completing this part, following command can be invoked in the terminal to test the implementation: `python main.py --task 2`

You will be able to see the robot fast simulate its motions for finding a solution. 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.

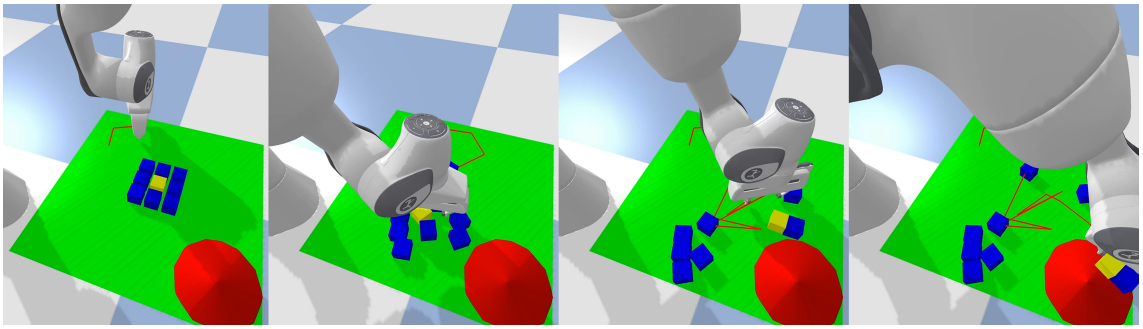


Figure 4: An example showing the robot executing a solution plan to push the yellow target object into the red goal region. The end-effector's trajectory is drawn by red lines.

## Task 3: Grasping in Clutter

In this part, we will implement the goal criterion function for a grasping task. As a reference, the relocating goal in the previous Task 2 is implemented by `RelocateGoal` in `goal.py`

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()` to help with necessary kinematics calculation.

```
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: For a grasp to be feasible, the end-effector should be close enough to the target object, and the orientation of the end-effector should be well aligned with the target object. In a `state`, the pose of the target object in $SE(2)$ (i.e., $x$, $y$ and $\theta$ of the target object) can be accessed 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, the following command can be invoked in the terminal to test the implementation: `python main.py --task 3`. This command will run the algorithm implemented in Task 2 for the new grasping task. If the grasping goal is implemented correctly, you should

be able to see the end-effector moving towards the target object while the solution plan is being executed. The robot should successfully grasp the target object with its fingers.

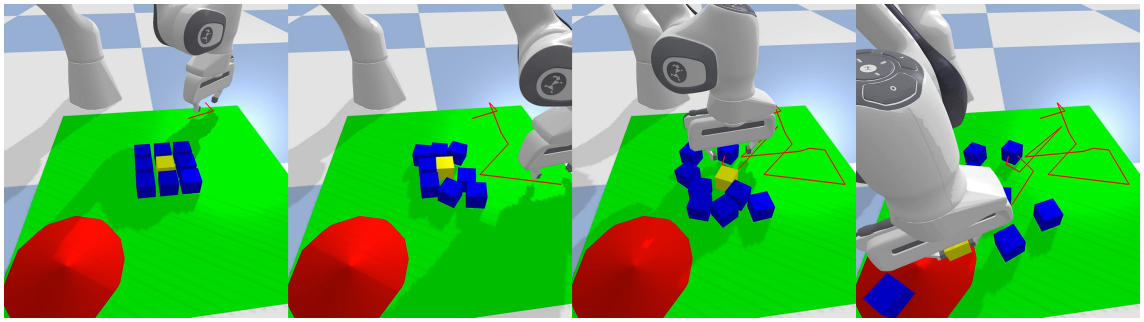


Figure 5: An example of the robot executing a solution plan to approach the yellow target object for grasping. In the end, the end-effector will grasp the target object by closing its fingers. The end-effector's path is drawn by red lines.

## Task 4: Trajectory Optimization (Optional)

In this task we will implement some trajectory optimization techniques for a relocating or grasping task. Feel free to implement any functions in `opt.py` and call the function under `TODO` in `main.py` to run the implemented methods. Anything relevant to the optimization of the manipulation motion trajectory will be of interest. For example, the length of the solution path can be minimized, or the total motion of non-target objects can be minimized. Stochastic trajectory optimization [1] can be a potential approach for this task. Note that the solution generated by the planner implemented in Task 2 can be used to generate an initial solution to be optimized. Show the method and the relevant findings in the 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.