EECS 465/ROB 422: Introduction to Algorithmic Robotics
Fall 2024
Homework Assignment #4
Due 11/6/2024 at 11:59pm

Rules:

1. **All homework must be done individually, but you are encouraged to post questions on Piazza.**
2. No late homework will be accepted.
3. The goal of this homework is to develop your understanding of the algorithms presented in class. You should use python to implement solutions. You may not use any other language, only python will be accepted.
4. Submit your zipped python files along with a `pdf` of your answers to Gradescope. Do not paste your code into your `pdf`.
5. Remember that copying-and-pasting code from other sources is not allowed.

Questions

1. (10 points) Explain why it is more difficult to define a distance metric that leads to good performance for a non-holonomic motion planning problem than a holonomic one. Use a car as an example that illustrates this difficulty and include diagrams.

Software

1. Download and unzip `HW4files.zip`. Run `test_utils.py` in the `pointclouds` folder. You should see a cloud of points in blue, that same cloud translated by $(1, 1, 1)$ in red, and a plane in green. If you do not see this, make sure `matplotlib` is installed. Refer to this script when you need to plot points and planes for the implementation problems involving point clouds.
2. Install `scipy` in your `venv` using `pip install scipy`.

Implementation

The following implementation problems should be done in python starting from the provided templates. Only edit what is inside the `### YOUR CODE/IMPORTS HERE ###` block. You should implement your own code from scratch unless explicitly stated otherwise. **Include all code you write in your `zip` file.**

1. Implement a Jacobian-based Inverse Kinematics Solver for the seven joints of the PR2's left arm. The code for this problem is in the `ik` folder. To simplify this problem, we will ignore end-effector rotation and use only the position variables in the Jacobian. We will also ignore collisions.

   a (15 points) Assume the robot has been placed in a certain configuration and you would like to compute the Jacobian at that configuration. Write a function to compute the Jacobian numerically, using the method described in class. Do this by filling in the code in the `get_translation_jacobian` function in `ik_template.py` in the `ik` folder. You will need to use the `get_joint_axis`, `get_joint_position`, and `get_ee_transform` functions defined in the template. You may not use any functions to generate Jacobians directly from `pybullet`.

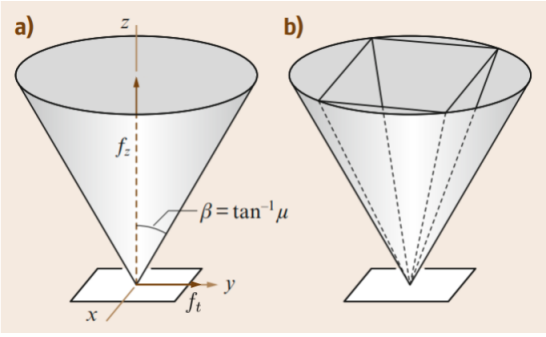
b (10 points) Now write a function to compute the pseudo-inverse of the Jacobian. Do this by filling in the `get_jacobian_pinv` function in the template. Because the robot may get into singularities, use the damped least-squares method with a small λ.

c (25 points) Implement the Iterative Jacobian Pseudo-Inverse Inverse Kinematics algorithm from lecture in `ik_template.py`. You will need to tune the parameters. In the template you can select one of five end-effector position targets. Try to make your algorithm achieve all the targets when starting from $q = \{0, 0, 0, 0, 0, 0, 0\}$. Do not allow q to exceed joint limits; if the algorithm goes past a joint limit, set that joint's value to be the limit. For each target you can achieve, save the configuration that places the end-effector at that position and a screenshot of the robot in that configuration in your `pdf`.

d (15 points) Make a copy of your code called `ik_template_nullspace.py`. Modify your algorithm in this file to use the left null-space of the Jacobian pseudo-inverse to repel the configuration away from joint limits while achieving the desired end-effector position. You will need to tune β so that the secondary task of repeling from joint limits does not interfere with task of reaching the target. For each target you can achieve, save the configuration that places the end-effector at that position and a screenshot of the robot in that configuration in your `pdf`.

2. Force closure evaluation: The `closure_template.py` script executes grasps using a 3-finger Robotiq gripper and a 7-DOF Kuka iiwa arm simulated in PyBullet. The code for this problem is in the `grasping` folder. You will be implementing friction cone calculation and a method that uses a convex hull to determine if grasps are in force closure.

   (a) (15 points) Implement the `calculate_friction_cone` and `compare_discretization` functions in `closure_template.py`. The friction cone, shown below, can be calculated given a vector f_z which represents the normal force associated with the contact, a vector f_t perpendicular to the normal force vector, and a coefficient of friction μ. f_t is an arbitrary vector lying on the plane perpendicular to f_z. We will discretize the friction cone to help calculate our set of wrenches.
   To generate a vector f_0 on the friction cone, we can rotate f_z about f_t by β, where $\beta = \tan^{-1}(\mu)$. To generate the full set of vectors to approximate the frictions code, we will then rotate f_0 about f_z. We want n evenly spaced vectors to form the approximation to the cone. We can achieve this by rotating f_0 by the set of angles $(2\pi i)/n$, for $i = 0...n-1$. Note that $n = 4$ in the figure below.



B. Siciliano, O. Khatib, and T. Kröger, Springer handbook of robotics, vol. 200. Springer, 2008.

To perform rotations, you should use the imported `Rotation` class from `scipy`. This class allows you to define rotations which can then be applied to input vectors. Specifically, you should use the `Rotation.from_rotvec` method. This method takes in a vector to rotate about, where the magnitude of the vector specifies the amount to rotate.

The number of vectors you choose (n) to include in your discretization will impact how close the discretization is to the true friction cone. Implement the `compare_discretization` function in `closure_template.py` to calculate the volumes of the true friction cone, a 4-vector approximation, and an 8-vector approximation.

In the `pdf`, include the volume of the true cone, 4-vector discretization, and 8-vector discretization for contact point 1 from Grasp 1. Round the volumes to 4 decimal places. Explain how changing the number of vectors changes the error of the discretization.

Note that you only need to implement the friction cone and volume calculations. The provided code will run your implementation on a given contact point.

(b) (25 points) Implement the `convex_hull` function in `closure_template.py` to determine if grasps are in force closure.

To do this, we first calculate wrenches acting on the center of mass of the object using the friction cone vectors. We will calculate a wrench vector for each force vector in our friction cone approximation. The wrench w at the center of mass associated with a contact force f is:

$$w = \begin{bmatrix} f \\ (\vec{r} \times f)/r_{max} \end{bmatrix} \tag{1}$$

where $\vec{r}$ is the vector pointing from the center of mass to the location where the contact is applied. The cross product with the force calculates the torque about the center of mass induced by the contact. We divide the torque by the maximum radius r_{max} of the object to ensure that the force and torque are in comparable units (Newtons). The code for converting from a contact force to a wrench on the object is provided.

To check for force closure, we will first construct a convex hull of the wrenches and then analyze its structure to determine if it contains the origin. We will use the Q-Hull algorithm to compute a convex hull. Scipy provides an implementation in the `ConvexHull` class (when computing the convex hull, use the `'QJ'` parameter to prevent numerical issues). The calculated convex hull can be thought of as an intersection of halfspaces in the 6D wrench space. Q-Hull provides the equations of these halfspaces, in `ConvexHull.equations`. Use these equations to determine if the grasp being tested is in force closure. If the grasp is in force closure, calculate the radius of the largest hypersphere centered at the origin that fits within the hull.

In the `pdf`, include an explanation of your method for checking force closure and computing the radius of the hypersphere. Also include the force closure output for each grasp and the hypersphere radius if the grasp is in force closure. Explain why knowing the hypersphere radius is useful.

You can run `closure_template.py` to execute one of two grasps and evaluate your force closure implementation. Run `python closure_template.py --g1` to evaluate Grasp 1. Run `python closure_template.py --g2` to evaluate Grasp 2. Include `closure_template.py` in your submission `zip` file. We will be running the submitted `closure_template.py` script on a new grasp to verify the implementation.

3. Run `pca_template.py` in the `pointclouds` folder. Here a point cloud of a randomly-sampled planar surface has been rotated and noise has been added to it. Rotate the plot to get a sense of how the points are distributed. Edit the template to accomplish the following.

   a (15 points) Use the Principle Component Analysis (PCA) algorithm to compute a rotation for the point cloud that aligns the surface in the point cloud with the XY plane as best as possible. Save a picture of the rotated point cloud in your `pdf` along with the V^T matrix you applied at the last step. Adjust the view in the plot so it is clear that the point cloud aligns with the XY plane.

   b (10 points) Use PCA to do the same rotation while also eliminating the noise in the data. The resulting point cloud should be two-dimensional. You will need to set the threshold for variances to properly filter out the noise. Save a picture of the point cloud after PCA is applied in your `pdf` along with the matrix V_s^T you applied at the last step. Adjust the view in the plot so it is clear that the point cloud aligns with the XY plane and all the z values are 0.

   c (10 points) Use PCA to fit a plane to the cloud and draw that plane in green in a plot with the point cloud. Include the plot in your `pdf`.

4. Run `ransac_template.py` in the `pointclouds` folder. Here a point cloud of a randomly-sampled planar surface has been rotated and noise has been added to it. We have also added some outlier points to the cloud. Rotate the plot to get a sense of how the points are distributed. Edit the template to accomplish the following.

   a (30 points) Use the RANSAC algorithm from lecture to find the equations of a plane that best fits the surface in the point cloud. You should select three points at each iteration to fit the model of the plane. The `Error` function for inliers should encode least-squares error, i.e. "the sum of squared residuals (a residual being the difference between an observed value, and the fitted value provided by a model)." (Wikipedia) You will need to set the number of iterations and the other parameters.

   Once you have computed the plane that best fits the data, include the equations for that plane in your `pdf`. Also include an image showing the plane in green fitting to the data, the inliers for that plane in red, and all other points in blue. Rotate the plot to clearly show the plane fitting to the data.

5. Now we will compare RANSAC and PCA with varying amounts of outliers. Run `pca_vs_ransac.py` in the `pointclouds` folder to see a series of point clouds with added outliers. At each iteration, we add 10 outliers to the cloud. At each iterations where outliers are added, run your PCA and RANSAC algorithms from above so that each produces a plane that fits to the cloud. Use the `Error` function from RANSAC to compute the error of each algorithm's plane to its inliers. Determine the inliers for PCA the same way as was done in the RANSAC algorithm.

a (10 points) For the last iteration, generate two plots: 1) Showing PCA's plane fitting to the data in green, the inliers for that plane in red, and all other points in blue; 2) Likewise for RANSAC. Clearly label the plots so it is clear which one corresponds to which algorithm.

b (15 points) Generate a plot of Error vs. Number of Outliers for both algorithms and include it in your `pdf`. Also record the computation times of PCA and RANSAC at each iteration.

c (15 points) Discuss the performance difference between the algorithms. Was there a clear performance difference in terms of computation time and error? If so, which algorithm performed better and why? If not, explain why not.

6. Run `icp_template.py` in the `pointclouds` folder to see a source and target point cloud of a mug. Implement the ICP algorithm to align a source point cloud to a target point cloud. Use Euclidean distance to determine correspondences at every iteration. You will need to tune the parameters and determine a reasonable termination condition.

   a (40 points) In the template you will see a way to load in one of four target point clouds. Run ICP on each target cloud separately. You may tune parameters separately for each target. You may not be able to produce a good alignment for all the targets, but try to make the source align to as many targets as you can.

   b (10 points) For each target, generate 1) A plot of Error vs. Iteration of ICP; 2) A plot showing the target point cloud and the source point cloud at the final iteration (in different colors). Include these in your `pdf`.

   c (15 points) Discuss the results; which targets were more difficult for ICP and why?