Prof. Dr. Maren Bennewitz
M.Sc. Padmaja Kulkarni

Bonn, 4 June 2017

Humanoid Robots

Exercise Sheet 8 - Anytime Repairing A*, Inverse Kinematics and RRT

Exercise 13 (10 points)

Extend the grid-based A* planner from exercise 11 to use Anytime Repairing A* (ARA*).

Exercise steps:

a) Implement the ARA* heuristic that inflates the Euclidean heuristic by a factor w in `ARAStarHeuristic::heuristic`.

b) "Anytime" algorithms continuously try to improve their solution until a time limit is met. Hence, the planning algorithm must be able to abort the search for a plan. Copy you code from the `PathPlanning::planPath` of exercise 11 to `ARAStarPlanning::planPath` and modify it so that it aborts searching when the time frame is exceeded. The method should then return an empty path.

c) Implement the `ARAStarPlanning::runARA` method that runs the path planner iteratively and decreases w in each iteration until either the optimal path is found, or the time limit has exceeded.

The Gnuplot script in the `scripts` directory and the animation in the Wiki show three plots: The figure on the left shows the path in red and the expanded nodes for each w in light blue. The top right figure shows how the path length of the returned path changes when w decreases. The bottom right figure shows how the planning time for each iteration increases with decreasing w.

Exercise 14 (10 points)

Consider the following robot arm in a 2D environment:

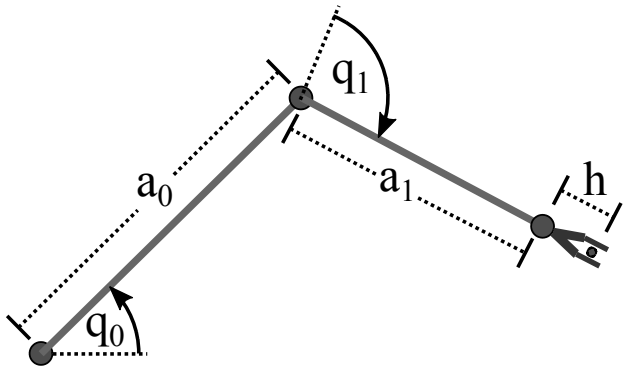

Figure 1: Robot arm with two links and a gripper

The arm has two links with lengths a_0, a_1, and a gripper of length h attached to the end of the arm. The links are connected to each other and to the base with two rotational joints. The current joint angles are q_0 and q_1.

The robot would like to grasp an object with its gripper. The location $\mathbf{g}$ of the object is given. Compute suitable joint angles q_0, q_1 so that the gripper arrives at the object location $\mathbf{g}$ by applying the inverse kinematics algorithm from slide 16 as follows:

a) Implement the method `forwardKinematic` that computes the endeffector pose $\mathbf{e}$ given the current joint angles $\mathbf{q}$. The equation for the endeffector pose in homogeneous coordinates is:

$$\begin{pmatrix} e_x \\ e_y \\ 1 \end{pmatrix} = \mathbf{R}(q_0) \cdot \mathbf{T}(a_0, 0) \cdot \mathbf{R}(q_1) \cdot \mathbf{T}(a_1, 0) \cdot \begin{pmatrix} h \\ 0 \\ 1 \end{pmatrix} \tag{1}$$

with the homogeneous rotation and translation matrices

$$\mathbf{R}(\theta) := \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{T}(x, y) := \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix} \tag{2}$$

Do not forget to convert $\mathbf{e}$ back to Cartesian coordinates in the end, as homogenous coordinates would disturb later when calculating the Jacobian.

b) Using pen and paper, derive the Jacobian $\mathbf{J}$ of the endeffector pose $\mathbf{e}$ with respect to the joint angles $\mathbf{q}$ from Eq. (1). The Jacobian is defined as

$$\mathbf{J}(\mathbf{e}, \mathbf{q}) := \begin{pmatrix} \frac{\partial e_x}{\partial q_0} & \frac{\partial e_x}{\partial q_1} \\ \frac{\partial e_y}{\partial q_0} & \frac{\partial e_y}{\partial q_1} \end{pmatrix}. \tag{3}$$

Afterwards, implement the result in the method `jacobian`.

c) Implement the remaining methods up to `computeIK` following the algorithm from slide 16 and the comments in the code.

Note: In `computeIK`, you have to initialize the joint angles $\mathbf{q}$. You can choose the initialization values arbitrarily with the following restrictions:

- Do not initialize an angle to 0, as this would lead to a singularity. If you initalize all joint angles to 0, all tangents will be vertical, so $\frac{\partial e_x}{\partial q_i} = 0$. In this case, the Jacobian is not invertible and the algorithm will fail.
- For most goal locations, there exist two solutions. The algorithm will return the solution that is closer to the initialization values. If the initialization values are very far from the solution, it may occur that the algorithm oscillates between the two solutions.

Now we would like to have our robot serve a drink. While the robot arm from above can reach any given point within its range, it cannot control the orientation of the hand, so it would spill out the drink. Hence, we have to add another link to the arm:

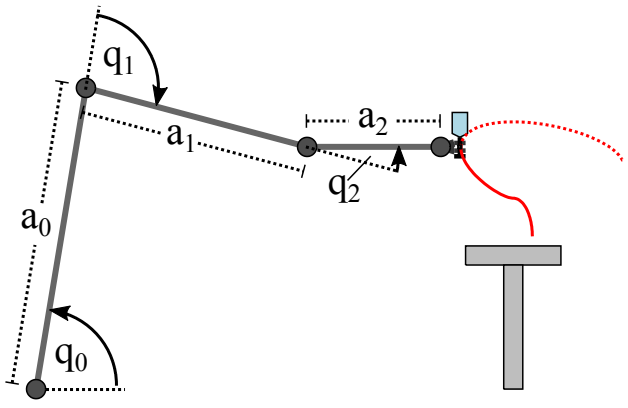

Figure 2: Robot arm with three links and a gripper serving a drink

With this configuration, the robot can move the glass along a trajectory while keeping the glass upright. The endeffector pose now consists of the position and orientation of the hand: $\mathbf{e} := (e_x, e_y, e_\theta)^T$. Here e_θ is the orientation of the hand with respect to the ground, so $e_\theta = 0$ means that the hand is horizontal and the glass is upright.

d) Implement the method `InverseKinematics_3Links::forwardKinematic` for the new robot. Similarly to Eq. (1), the position can be computed as

$$\begin{pmatrix} e_x \\ e_y \\ 1 \end{pmatrix} = \mathbf{R}(q_0) \cdot \mathbf{T}(a_0, 0) \cdot \mathbf{R}(q_1) \cdot \mathbf{T}(a_1, 0) \cdot \mathbf{R}(q_2) \cdot \mathbf{T}(a_2, 0) \cdot \begin{pmatrix} h \\ 0 \\ 1 \end{pmatrix} \tag{4}$$

Additionally, compute e_θ from q_0, q_1, q_2 and return the complete vector (e_x, e_y, e_θ).

e) The Jacobian is now a 3×3 matrix:

$$\mathbf{J}(\mathbf{e}, \mathbf{q}) := \begin{pmatrix} \frac{\partial e_x}{\partial q_0} & \frac{\partial e_x}{\partial q_1} & \frac{\partial e_x}{\partial q_2} \\ \frac{\partial e_y}{\partial q_0} & \frac{\partial e_y}{\partial q_1} & \frac{\partial e_y}{\partial q_2} \\ \frac{\partial e_\theta}{\partial q_0} & \frac{\partial e_\theta}{\partial q_1} & \frac{\partial e_\theta}{\partial q_2} \end{pmatrix} \tag{5}$$

As deriving the Jacobian for more than two joints is a lot of work, we use a numerical estimation of the Jacobian instead. The top left component can be esimated using the difference quotient as follows:

$$\frac{\partial e_x\left((q_0, q_1, q_2)^T\right)}{\partial q_0} \approx \frac{e_x\left((q_0 + \varepsilon, q_1, q_2)^T\right) - e_x\left((q_0, q_1, q_2)^T\right)}{\varepsilon} \tag{6}$$

for a small ε. The other components can be calculated similarly.

In `InverseKinematics_3Links::jacobian()`, calculate the difference quotients with the help of the `forwardKinematic()` method and fill the Jacobian matrix.

f) Implement `InverseKinematics_3Links::computeIK()` to complete the computation of the inverse kinematics.

The Gnuplot script in `scripts/plot.gp` and the animation in the Wiki show an animation of the robot arm serving a drink.

Exercise 15 (10 points)

Consider a robot in a 2D environment represented by a grid map. Our objective is to find a path between a start cell and a goal cell. The robot is initially located at the start cell's center and it can move to any of the eight neighboring cells' centers (if they are not occupied). Use RRTs to find a possible path for the robot.

Exercise steps:

a) Implement the method `getRandomNode` that returns a random grid cell in the map, with the following conditions:

- The cell must be free of obstacles.
- The cell must not already belong to the tree that will be expanded in the current step.
- The random generation should be biased towards the goal of the current tree, i.e., the method should return the goal node instead of a random node with a probability of 10%.

Hint: Use the standard C function `rand()` that returns a random integer between 0 and `RAND_MAX`.

b) Implement the method `distance` that computes the Euclidean distance between two given cells.

c) Implement the method `getClosestNodeInList` that returns the grid cell with the smallest distance to the current node from a list of cells.

d) Implement the method `getNeighbors` that returns a list of neighbors of the current cell that are not occupied and have not already been expanded in the current tree.

e) Implement the method `tryToConnect` that checks for each neighbor whether it is is already contained in the list of expanded nodes of the other tree. If this is the case, then we have found a connection between the two trees and the method should return the neighbor as the connection node. Otherwise, the method should return `NULL`.

f) Implement the method `addNearestNeighbor` that determines which neighbor is the nearest with respect to the random node and adds that neighbor to the tree.

g) Have a look at how the method `extendClosestNode` combines the above methods (you don't have to implement anything here).

h) Implement the method `constructPath` that reconstructs the path from the start node to the goal node once the trees have been connected.

i) Implement the method `planPath` which grows the trees from the start and goal nodes and returns the complete path, using the RRT-connect algorithm on slide 30.

The Gnuplot script in `scripts/plot.gp` and the animation in the Wiki show an animation of the node expansion and the final path found by your implementation.

Deadline: 18 July 2017, 11:59 am