Before you begin

- Download the `robot_arm.py` file from Canvas. Read through the code and run the code in the `if __name__ == '__main__'` section once.

Learning Objectives:

The goal of this homework is to give you some experience in dealing with writing code which can be used to simulate real physical systems and visualize them, as well as being able to frame problems as optimization problems to solve them creatively.

Thoughts, come back after you have read the problems:

- Though this problem deals with the kinematics of robot arms, very little knowledge of kinematics beyond a basic concept of what a robot arm is required for this problem.
- If you're having problems thinking about how to approach the fmin problem, remember that all you're doing is defining a function of some variables which spits out a scalar and then throwing it into the black box that is fmin. fmin has no knowledge that changing the θ's is rotating the robot arm to move closer to a point; it simply changes the values of the θ's and observes that the number moves in some direction.

Grading Checkpoints [24 points]

Check the footnotes for associated rubric items [*].

Assignment Grading Checkpoints (20 points total + 2 extra credit)

- [2 points] `Link.check_wall_collision` (1 point) and `RobotArm.get_collision_score` (2 points) are implemented correctly [*].
- [6 points] `RobotArm.plot_robot_state()` shows the links (1 point), colliding links (1 point), target (1 point), vertical walls (1 point), and outputs it to the given file name (1 point) [*].
- [4 points] `RobotArm.ik_grid_search` returns the desired values (1 point) with a point which minimizes the distance (2 points) while ignoring wall collisions (1 point) [*].
- [4 points] `RobotArm.ik_fmin_search` returns the 3 desired values (2 point) with a point which minimizes the distance consistent with our implementation (2 points) [*].
- [3 points] Scatter plot showing fmin evaluations looks correct (2 points) with a corresponding visualization of the found IK solution (1 point).
- [2 points] **Extra Credit:** `RobotArm.ik_constrained_search` produces points which respect constraints (1 point)*, with corresponding plots from constrained and unconstrained optimization (1 point).

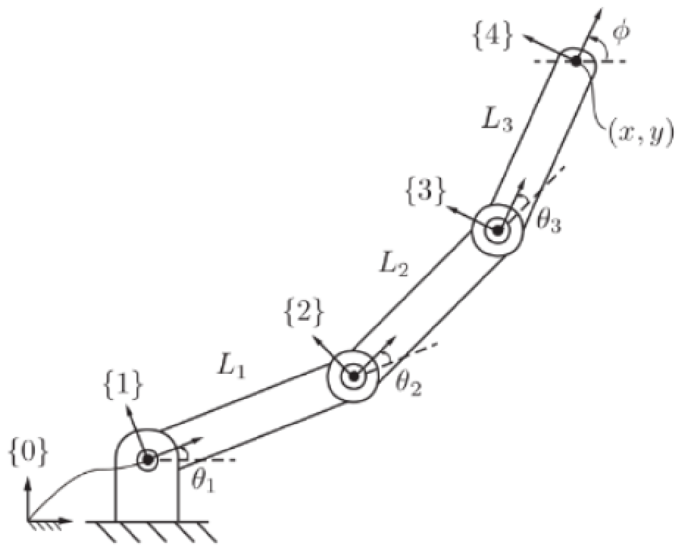
Standard Grading Checkpoints (4 points total)

- [2 points] Code passes PEP8 checks with 10 errors max [*].
- [2 points] Code passes TA-reviewed style checks for cleanliness, layout, readability, etc.

[*]Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.

Overview

In this assignment, we've provided you a file `robot_arm.py` which contains a `class RobotArm` representing a planar robot arm with an arbitrary number of links, such as the one shown below:



RobotArm currently has two methods already implemented for you which will get the physical location of each of the robot's links as `Link` objects, as well as the location of the end effector. Read through the class and get an understanding of how to initialize a robot arm, as well as how to get forward kinematics and links. **Don't change anything that's already implemented.**

The goal of this assignment is twofold: To be able to represent this data visually via matplotlib, as well as to gain some insight into the problem of `inverse kinematics`, i.e. given a desired target in the environment, what set of joint angles (if any) will achieve it? Your goal will be to implement all of the methods in `robot_arm.py` which are currently marked as NotImplementedError.

For this assignment, we also have the possibility of having obstacles in the robot's environment. In particular we have provided a class VerticalWall which represents a wall in the robot's environment located at x=c for some c.

Problem 1 Setup

- First, implement the `check_wall_collision` method of the `Link` class. It should take in an instance of a VerticalWall and return `True` if the link intersects the vertical wall at the corresponding object's x-location, and False otherwise.

  Example:

  ```
  my_link = Link((1.1, 5.0), (3.0, 3.3))
  my_link.check_wall_collision(VerticalWall(2.1)) # True
  my_link.check_wall_collision(VerticalWall(-0.3)) # False
  ```

- Next, implement the `get_collision_score` method of the `RobotArm` class. For each link, it should count the number of walls in the robot arm's environment which the link is in collision with. It should then sum up all of those numbers and return the `negative` of the sum. So when there are no collisions, the score is 0; when there is 1 link colliding with 1 wall, the score is -1; when there is 1 link colliding with 2 walls or 2 links colliding with 1 wall each, the score is -2; etc.

  Example:

  ```
  my_arm = RobotArm(1, 1, 1, obstacles=[VerticalWall(1.5)])
  # Arm is vertical, should be 0
  my_arm.get_collision_score}([np.pi/2, 0, 0])
  # Arm is horizontal but then folds back, should be -2
  my_arm.get_collision_score([0, 0, np.pi])
  ```

Comment: Yes, the first part is as easy as it seems. The point of this is that in general, you can have arbitrary representations of obstacles in the environment and have a check to see if your link collides with those walls. Simulating such collisions can begin to be quite computationally expensive when dealing with 3D models of robot links and environments, but thankfully we're not asking you to implement anything like that.

Problem 2 Plotting

Next, we would like to visualize any given state of a robot arm, as well as its environment. We will do this via matplotlib. Implement the method `plot_robot_state` which takes in a required argument, the joint angles, as well as an optional value `target` representing an (x,y) target point in the environment. It should output a plot to the input file name with a visualization of the following:

- The vertical walls of the environment (if any)
- If target is not None, a single point with the target location. You may choose whatever marker you wish for the target so long as it is clearly visible.
- The links of the robot. If a link is in collision with any of the walls, you should distinguish the link by **coloring it differently** and plotting it as a **dashed** line. The joints should also be visible via some sort of marker, even if the arm is at its zero-angle configuration.
- The x and y bounds should be set so that if the robot is fully stretched out in one direction, it will still fit within the plot. (What's the maximum value the arm can be stretched out?) If the target or walls fall outside of these bounds, they do not have to show up on the plot.
- No need for a title or axis labels.

Then for the robot `RobotArm(2, 1, 2, obstacles=[VerticalWall(3.2)])`, run `plot_robot_state([0.2, 0.4, 0.6], target=[1.5, 1.5])`, and place the plot into the document here.

Problem 3 Finding IK solutions

The goal of this section is to solve the inverse kinematics problem, i.e. given a target in the environment, what set of joint angles will achieve that target, if any? This is a tough problem, and in general analytical solutions for arbitrary robot arms may not exist, which is why we need to search for them numerically.

- Implement the method `ik_grid_search`, which does the following. It uses the input `intervals` to split up the interval $[0, 2\pi]$ into the specified number of endpoints, e.g. if `intervals = 4`, then it picks $[0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}]$. It then tries every possible combination of those θ's for the joint angles and **returns two values**: The θ's which resulted in the closest distance to the target, as well as the distance from the target. It should **exclude** any combinations which are in collision with any walls.

  Note that this means the number of points searched will be `intervals**num_links`, e.g. a grid search of 4 for a robot with 3 links will result in $4^3 = 64$ points being searched. Hint: You may be interested in the `product` generator from the itertools package. You can use `product(grid_coordinates, repeat=n)` to get the desired grid points (where `grid_coordinates` is something like $[0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}]$).
- Implement the method `ik_fmin_search`, which searches for an IK solution by utilizing `scipy's fmin function`, where the cost function we are trying to minimize is the distance from the target. It should initialize the estimate to `thetas_guess`, and when calling `fmin`, it should set `maxfun` to `max_calls`, to limit the number of function evaluations.

  It should **return three values**: the θ's which resulted in the closest distance to the target, the distance from the target, and the actual number of function calls it took for convergence (not the number of iterations).

  Note: fmin cannot take constraints into account, so it may end up with an IK solution which collides with a wall. This is OK.

  Note: For the purposes of autograding, we will track how many times that `RobotArm.get_links()` gets called, which should match up with the count `fmin` returns. Because of this, you should only call `RobotArm.get_ee_location` once per `fmin` evaluation.

- **Analysis:** For the 3-arm robot `RobotArm(2,1,2)` and a target of $[-1.5, 1.5]$, for `max_calls = 0, 5, 10, . . .`, run `ik_fmin_search` with an initial guess of $[0, 0, 0]$ until the produced result converges and `fmin` no longer utilizes all of the allotted calls. Produce a scatter plot showing the distance from the `target` as the y-axis against the number of function calls on the x-axis. Include the plot in your submission, as well as a plot from `plot_robot_state` verifying that the chosen IK solution actually reaches the target.

Extra Credit: Constrained Optimization As noted above, fmin is an `unconstrained` optimization, so it cannot take constraints such as wall collisions into account. In this section, we'll implement an optimizer that can take into account constraints so that we can get an IK solution which is guaranteed to not have any wall collisions.

Read the documentation on the `minimize` function in scipy. The method which we wish to use for a constrained optimization is COBYLA. Read up on how to pass in constraints, where the constraint type is 'ineq' and the constraint function we use is `get_collision_score`.

Then use this method to implement the `ik_constrained_search` method. Also make sure to pass the max_iters value into the options dictionary. It should return a **single** result, which is either the array of optimized θ's, or the value False if the optimization terminated unsuccessfully.

Then, for the robot arm `RobotArm(1, 4, 2, obstacles=[VerticalWall(-0.5), VerticalWall(1.0)])` and a `target: [-0.3, 1.5]`, run `ik_constrained_search` with an initial guess of $[0, 0, 0]$ and `max_iters=1000` and include the corresponding `plot_robot_state` IK visualization below. Do the same for fmin with `ik_fmin_search` to show how COBYLA respects the constraints while `fmin` doesn't.