Algorithmic Robotics
COMP/ELEC/MECH 450/550
Project 2: Shapes on a Plane

When planning the motions of a robot, most planning algorithms plan within the *configuration space* ($\mathcal{C}$-space) of the robot. The $\mathcal{C}$-space of a robot can be radically different than the robot's workspace $\mathcal{W}$; the robot's entire state is encoded as a single point in $\mathcal{C}$-space. Generally, there is not an exact representation of the workspace in the robot's $\mathcal{C}$-space. To check validity of a robot configuration, we must *place* the robot back into workspace at the configuration and see whether the configuration is valid. For rigid bodies and articulated mechanisms, this mapping can be computed using *rigid transformations*.

In this project, you will be implementing collision checking routines for three different robots: a point robot, a circle robot, and a square robot. Each of these robots operates within the plane. These robots allow us to implement three different kinds of collision checking routines: one that uses an exact representation of $\mathcal{C}$-space, one that uses an implicit representation of $\mathcal{C}$-space, and one that must probe an unknown $\mathcal{C}$-space. All obstacles in this project will be **axis-aligned rectangles**, which will remain constant for each experiment. The obstacles are defined in the file `Project2.cpp`.

Note: Although this project does not use OMPL, the routines developed in this project can (and should) be used in subsequent projects, so it is important to implement them cleanly and correctly.

Point robot

The simplest robot in this project is a point that translates in the plane. When planning for the point, the collision checker must decide whether the point is inside an obstacle. In this case, the $\mathcal{C}$-space for the robot is the same as the workspace. It then follows that the following is true if and only if robot is in collision with an axis-aligned rectangle:

$$(x_{min} \leq x \leq x_{max}) \qquad \text{and} \qquad (y_{min} \leq y \leq y_{max}) \tag{1}$$

where x_{min}, x_{max}, y_{min} and y_{max} are the range of the axis-aligned obstacle's coordinates, and x, y are the coordinates of the point robot.

For the project, you will fill out `isValidPoint` in `CollisionChecking.cpp` with the necessary code to evaluate validity of a point robot's configuration.

Circle robot

The second robot is a circle that translates in the plane. Exact collision checking is more challenging when the robot has geometry because we do not usually have a representation of the obstacles in the $\mathcal{C}$-space. When the reference point on the circle robot is the center point, however, we can implicitly compute the $\mathcal{C}$-space representation of an axis-aligned rectangle by *expanding* the obstacles an amount equal to the radius of the circle robot. For the circle robot, rectangular workspace obstacles become larger rectangles with rounded

corners in the $\mathcal{C}$-space. The condition below is true if and only if the circle robot with radius r intersects with an axis-aligned rectangle:

$$\begin{aligned} &((x_{min} - r \leq x \leq x_{max} + r) \text{ and } (y_{min} \leq y \leq y_{max})) \text{ or} \\ &((x_{min} \leq x \leq x_{max}) \text{ and } (y_{min} - r \leq y \leq y_{max} + r)) \text{ or} \\ &\quad \exists e, \text{ a vertex of the rectangle, } \|\overrightarrow{ce}\| \leq r \end{aligned} \tag{2}$$

where x_{min}, x_{max}, y_{min} and y_{max} are the range of the axis-aligned rectangle's coordinates, $c = (x, y)$ is the center of the circle robot, and $\|\overrightarrow{ce}\|$ is the Euclidean norm of the vector from point c to point e.

For the project, you will fill out `isValidCircle` in `CollisionChecking.cpp` with the necessary code to evaluate validity of a circle robot's configuration.

Square robot

When the robot rotates, an implicit $\mathcal{C}$-space representation (as above for the circle robot) is difficult to derive analytically, even for the simple shapes used in this project. Think about why this is. Since we do not have any representation of the $\mathcal{C}$-space obstacles, we must perform collision checking in the workspace. The shape of the robot is assumed to be rigid, allowing us to unambiguously specify the geometry of the robot in a local coordinate frame, as shown in Figure 1(a). With this representation, it is easy to rigidly transform the geometry and place the robot at a specific configuration in the workspace for collision checking.

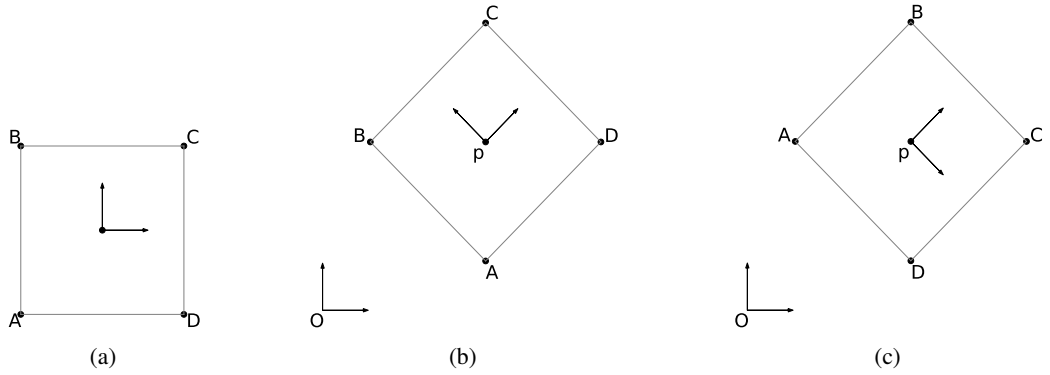

Figure 1: (a) The geometry of the square robot in its local coordinate frame. The points A-D are specified with respect to the local origin at the center of the square. The choice of local origin is arbitrary, but must remain constant. (b) The robot at $p = (x, y)$ and $\pi/4$ rotation in the workspace. (c) The robot at $p = (x, y)$ and $-\pi/4$ rotation in the workspace. Although (b) and (c) look similar, these are two geometrically distinct configurations of the robot.

Define $\mathcal{T}(\theta, \mathbf{v})$ as a transformation matrix:

$$\begin{pmatrix} R(\theta) & \mathbf{v} \\ \mathbf{0} & 1 \end{pmatrix}$$

where $R(\theta)$ is the rotation matrix for angle θ and $\mathbf{v}$ is a column vector representing the translation with respect to the origin. Each transformation matrix $\mathcal{T}$ represents a coordinate frame with a specific origin and rotation. Let $\mathcal{T}_O$ be coordinate frame for the workspace origin. The coordinate frame for the robot $\mathcal{T}_R$ at a specific configuration in the workspace is:

$$\mathcal{T}_R = \mathcal{T}_O \mathcal{T}(\theta, \mathbf{v}). \tag{3}$$

In other words, $\mathcal{T}_R$ is the transformation of the robot's local coordinate frame to a specific configuration in the workspace. Given $\mathcal{T}_R$, all that remains is to reassemble the robot's geometry using the new frame. For each homogeneous point $\mathbf{q}$ in the robot's local frame, the resulting point in the workspace is:

$$\mathbf{q}' = \mathcal{T}_R \cdot \mathbf{q}. \tag{4}$$

Figures 1(b) and 1(c) show examples of rigid transformations. First, the robot's local coordinate frame is placed at the desired configuration, then each point in the geometry (A-D) is mapped into the workspace.

Collision checking Equations (3) and (4) reassemble the robot's geometry at a specific configuration in the workspace, but we still need to test whether a configuration of the robot is collision-free. One simple way to check for collisions in the plane is to intersect the line segments that compose the robot and environment geometry. If any of these segments intersect, then the robot collides with an obstacle in the environment. There exist more efficient methods to perform collision checking, but an *all-pairs* test between line segments, combined with a check that the robot does not lie completely within any obstacle, is sufficient for this project.

For the project, you will fill out `isValidSquare` in `CollisionChecking.cpp` with the necessary code to evaluate validity of a square robot's configuration.

Project exercises

1. Implement exact collision checking procedures for the point robot, circle robot, and square robot described above. Provided code is given through Canvas. You are required to fill out the implementation in `CollisionChecking.cpp` and `CollisionChecking.h`. The representation of the environment along with the code to run your validity checker is contained in `Project2.cpp`. You should look at this code, but you should not modify it. You may assume the workspace is unbounded and contains only axis-aligned rectangular obstacles.
2. Verify your implementation. Your implementation must accurately classify robot configurations as collision-free or in-collision. A set of configurations are provided in the code template. The directory `configs` provides a set of test cases that you *must* pass for full credit. The directory `optional` provides a set of test cases that are, as it says, optional. However, if your implementation is truly correct, it should pass with 100% on the optional test cases as well.

Note This project involves computational geometry, which can be tricky even for seasoned programmers. You may disregard the degenerate cases where line segments overlap (intersect at many points) or where the endpoint of one segment intersects another segment. These cases require a tedious implementation that is outside the scope of this project, and should not appear in any of the provided test configurations.

Protips

- An explicit matrix representation of the coordinate frames may be overkill since we are operating in the plane with only two coordinate frames. Consider expanding the matrix multiplications and implementing those equations instead to simplify your implementation.
- Implement the collision checkers in the order presented. You may be able to use portions of the code from the simpler collision checkers in the more complex checkers.
- Visualization is not required for this project, but can be immensely helpful, especially when debugging rigid transformations. We *highly* recommend you develop a visualizer. Use any software you want:

MATLAB, Excel, gnuplot, Python's matplotlib, etc. A simple visualization may be required in future projects. Any code written for visualization also falls under the honor code policy for the class.

Deliverables

This project may be completed in pairs. **Submissions are due Tuesday Sept. 24 at 1pm**.

To submit your project, clean your build space with `make clean`, zip up the project directory into a file named `Project2_<your NetID>_<partner's NetID>.zip`, and submit to Canvas. Your code must compile within a modern Linux environment. If your code compiled on the virtual machine, then it will be fine. If you deem it necessary, also include a README with details on compiling and executing your code. In addition to the archive, submit a short report that summarizes your experience from this project. The report should be no longer than 5 pages in PDF format, and contain the following information:

1. **(7.5 points)** A succinct statement of the problem that you solved.
2. **(7.5 points)** A short description of the robots (their geometry) and their configuration spaces.
3. **(10 points)** A summary of your experiences in implementing the different collision checkers. Were there any cases that were particularly easy/difficult? Did you run into any numerical precision issues or other similar complications? Does your implementation accurately classify the given test sets? How did you debug your implementation? Does your code accurately classify all of the optional test sets?
4. **(5 points)** Rate the difficulty of each exercise on a scale of 1–10 (1 being trivial, 10 being impossible). Give an estimate of how many hours you spent on each exercise, and detail what was the hardest part of the assignment. **Additionally**, for students who completed the project in pairs, describe your individual contribution to the project.

Take time to complete your write-up. It is important to proofread and iterate over your thoughts. Reports will be evaluated for not only the raw content, but also the quality and clarity of the presentation.

Additionally, you will be graded upon your implementation. Your code must compile, run, and solve the problem correctly. Correctness of the implementation is paramount, but succinct, well-organized, well-written, and well-documented code is also taken into consideration. The breakdown of the grading of the implementation is as follows:

1. **(15 points)** Collision checking routine for the point robot.
2. **(25 points)** Collision checking routine for the circle robot.
3. **(30 points)** Collision checking routine for the square robot.

Those completing the project in pairs need only provide one submission.