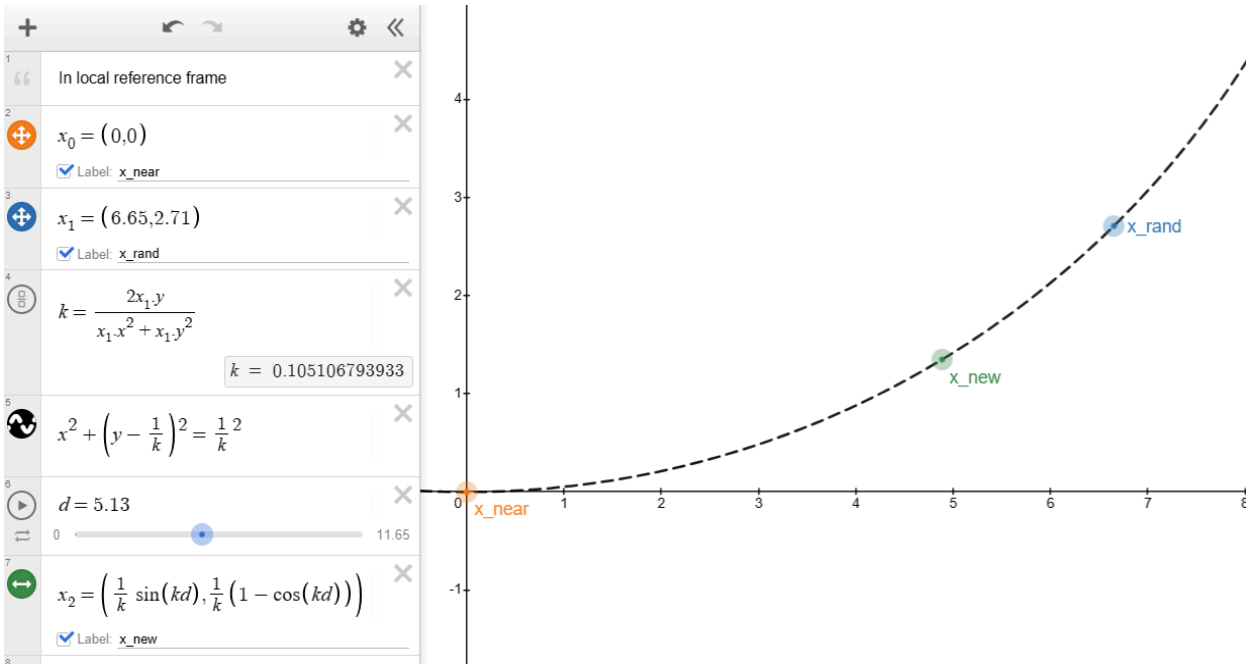
# CS 378H: Autonomous Robotics
# Milestone 3 Report

## 1) Previous Mathematical/Algorithmic Computations

Rapidly Exploring Random Tree: We use the RRT algorithm to plan paths. An additional constraint is that we limit how far the new steered path goes toward the random sample to avoid "spaghettification" and reduce the likelihood of the path intersecting a wall, allowing the tree to build iteratively.

1. Start with a graph only having our start point
2. Sample a random point, x_rand
3. Find nearest neighbor in graph, x_near
4. Transform both points to local reference frame around nearest neighbor
5. Steer towards random point with optimal constrained curve and max dist
    a. This is the only step in local reference, collision checks are in global reference frame.
6. Find this new end point, transform to global reference frame, x_new
7. Do coarse bounding box check for potential intersecting wall segments
    a. For each found, check if true intersection via arc intersect below
8. If no collisions from the checks, add to graph
9. Repeat from step 2 until within epsilon distance of goal point.

Steer to Point: Given the random point selected by RRT, we calculate the "near" true point we steer to in the direction of the random goal point. This is done by first transforming from global to local reference frame (rotation + translation), then the ideal curvature is given by $k = \frac{2y}{x^2+y^2}$, where x and y are the coordinates of the transformed goal point. If the curvature is too tight, we clamp to the max curvature allowed by our limits. The arc path segment is then constructed, and a maximum distance $d$ is set, so the planning occurs in small increments. The final point coordinates in local reference frame are $x' = \frac{1}{k}sin(kd), y' = \frac{1}{k}(1 - cos(kd))$. We then transform back to global coordinates (translation + rotation).




Bounding Box Wall Collision: As part of the RRT planning, we have to compute whether the travel path arc is collision-free. First, a bounding box method is used to coarsely isolate the line segments of the map that may intersect with the travel path (the arc intersect collision check is computationally expensive). We define the bounding box to be from x_near (step 3) to x_new (step 6), with additional margin to account for the dimensions of the car: $margin = \sqrt{l^2 + (\frac{w}{2})^2}$.

Arc Intersect Wall Collision: For each segment (and for the outer and inner radius of the car), we do the following:

1. Extend the path arc into a circle and extend the segment into a line.
2. Compute the intersection points by formula if they exist (if not, collision-free):

$$x = \frac{a + m^2 c - md + mb + \sqrt{-m^2 a^2 + 2m^2 ac + 2mab - 2mad + m^2 r^2 + 2mcd + 2db + r^2 - m^2 c^2 - 2mcb - d^2 - b^2}}{1 + m^2}, x$$
$$= \frac{a + m^2 c - md + mb - \sqrt{-m^2 a^2 + 2m^2 ac + 2mab - 2mad + m^2 r^2 + 2mcd + 2db + r^2 - m^2 c^2 - 2mcb - d^2 - b^2}}{1 + m^2};$$
$$m \neq i, m \neq -i$$

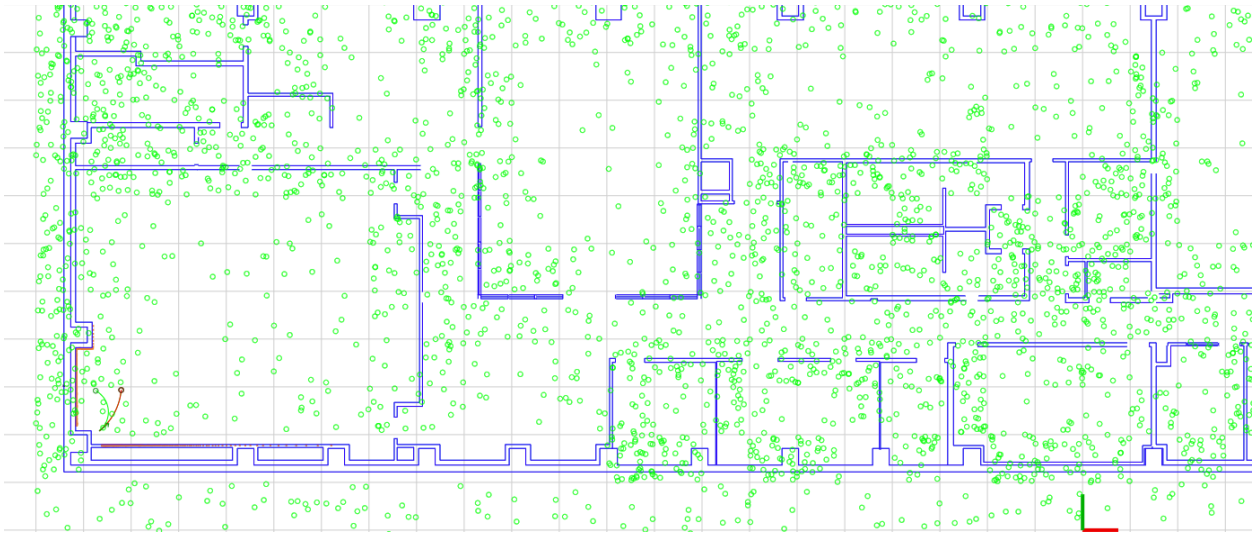
Where m is the slope, (a,b) is the center of the circle, (c,d) is a point on the line.

3. Switch back to arc and segment and check if the intersection points overlap the ranges of both the arc and the wall segment or if it's outside the bounds (false positive).

For vertical lines, we currently adjust the values slightly to avoid division by 0, but this can lead to very large slope values that may lose precision, so a further improvement is to do a second implementation based on y-values and choose that when slope > 1 or < -1.

Desmos graph with derivation: https://www.desmos.com/calculator/nyco4akfli

Distance Transform + Sampling Points: In order to randomly sample points from the grid for RRT, we set up a distribution based on the distance of a point from the walls, biasing sampling towards narrower areas. We then set up a CDF using this as our distribution. See below for an example:



## 2) New Adjustments

Accounting for Dynamic Obstacles: planning collision detection was adjusted by converting line segments from the map into point clouds and using prior collision checking algorithms. This allows for easier integration with the LIDAR point cloud scan enabling us to account for both static wall obstacles and obstacles not found on the blueprint.

Dynamic Replanning: in line with the milestone goals, we rerun the RRT algorithm if we find that our planned path intersects an obstacle ahead of us. Currently, this involves creating a new random tree starting from the current location toward the same goal.

## 3) Code Organization

graphgen.cc

- steerToPoint(): Determines the arc needed to travel closest to the exploration goal.
- pathLineIntersection(): Computes the arc wall collision above.
- computeBoundingBox(): Computes the bounding box that is first used as a quick filter for relevant map lines.
- validPath(): Checks if a path (from steerToPoint) is valid (collision-free) via above functions.
- createGraph(): Implements the RRT tree graph algorithm.
- visualizePath(): backtraces the RRT graph from the end goal to the start via parent pointers and visualizes them.

steer.cc

- steer(): Manages the kinodynamic constraints of the car to ensure the generation of a feasible path

dynamic_planner.cc

- comparePlanWithScan(): Takes the current point cloud and uses it to update a kdTree representing the map in places where it detects points. Each detection increases our confidence that a point is there
- needToReplan(): Checks our current state and determines if our planned path is invalid due to high confidence in a particle intersecting our future path

distance_transform.cc

- getDistanceTransform(): Takes the current world map and generates a joint distribution over the x,y plane with higher values in regions of greater wall density

## 4) Challenges Faced

We had some problems with earlier milestone code, whose hidden bugs are now rearing their ugly heads. We had to fix our point targeting code from MS 1, since it only assumed forwards motion was necessary. We also had to improve our RRT because it struggled to path through chokepoints.

## 5) Individual Contributions

**Aadarsh** - tweaked RRT parameters in order to improve performance, including involving multiple nearest neighbors, biasing depth over branching, reducing path length, debugged point targeting, wrote dynamic replanning logic

**Jeriah** - Debugging RRT failing to branch out, improving particle filter to jitter less, tuning RRT params, debugging and fixing linking issues (again) to make the code actually run, wrote Report
**Luke** - Rewrote MS 1 code in order to better approximate global plan, implemented distance transform, added CDF sampling logic from distance transform, updated scoring function, played with different versions of clearance, path planning tuning, code refactorization, debugging

6) **GitHub Link**: https://github.com/Aadarsh271/Autonomous-Driving/tree/lt-final-pathing
**Commit Hash**: ed0b2e4d17ae4b02c86dd117d1cfc43ea0227193
7) **Videos**
**Navigating Around the GDC 1 (No replanning)**
**Navigating Around the GDC 2 (With Replanning)**