University of Colorado - Boulder

ASEN 5519: Algorithmic Path Planning

December 16, 2021

Kinodynamic Goal-Bias RRT for a Self-Driving Car with Proximity Sensor in a Dynamic Environment

Author:
Fernando Palafox[a]
[a]SID:107070636

Professor:
Professor Morteza
Lahijanian

A path-planning algorithm for a simulated self-driving car was designed and implemented in MATLAB. This was done in three stages, each with accompanying pseudocode, implementation details and an example scenario. First a goal-bias RRT for path-planning with kinodynamic constraints was implemented. Then, it was modified for use in an environment with dynamic obstacles. Finally, a sensor suite in the form of a proximity was simulated and the algorithm was adapted with a path re-growth procedure in case of collisions with unexpected obstacles.

Ann and H.J. Smead
Aerospace Engineering Sciences
UNIVERSITY OF COLORADO **BOULDER**

I. Introduction

The worldwide adoption of automated technologies has the potential for widespread and far-reaching positive change. These technologies can be applied for a variety of tasks such as medical procedures, transportation, and industrial settings. For the purpose of this investigation, I focused on a specific problem within the field of autonomous systems: self-driving cars. Self-driving cars can help avoid hundreds of thousands of deaths due to easily avoidable crashes, can increase accessibility to people with disabilities, and potentially help curve the problem of global warming by reducing traffic jams and unnecessary idling. Additionally, it is an extremely complex problem at the intersection of many problems that interest me such as artificial intelligence, control theory, path planning and logic.

II. Problem statement

In order to gain a better of understanding and practical experience on the problem of creating fully-autonomous self-driving cars, I decided to code simulation and apply some of the concepts learned in class. I began by broadly defining an end deliverable: A simulated self-driving car with the following capabilities:

- Path-planning between start and goal states
- Path-planning which follows the kinodynamic constraints of a car
- Static and dynamic obstacle avoidance
- Sensing capabilities which can leveraged to adapt the path for unexpected changes in planning environment

With a clear deliverable in mind, I broke it down into a set of sub-problems used to ground the overarching goal into a concrete set of incremental actions:

1) Implement a goal-biased RRT with kinodynamic constraints
2) Adapt the algorithm to deal with dynamic obstacles.
3) Add sensing and re-planning capabilities to deal with unexpected obstacles
4) Implement a Path-Directed Subdivision Tree (PDST) planner to improve efficiency and get more experience with newer planning algorithms.

Each sub-problem is meant to build on existing knowledge and push it slightly beyond what I'd done before. For example, I had plenty of experience with sampling-based algorithms like goal-biased RRT, but I now had to adapt for correct sampling under the kinodynamic constraints of a car. This was meant to keep the overall goal tractable while also challenging me to learn new techniques.

III. Methodology

Although I am a firm believer in "standing on the shoulder of giants," for this project I chose to implement the entire deliverable myself, without the use of any pre-built physics engines, libraries or simulation suites. This decision was justified by my interest in gaining a low-level understanding of the intricacies of designing, building and testing a self-driving car simulation from the ground up. In robotics, much of the literature is dedicated to explaining high-level concepts, which although extremely important, tend to leave out many of the details of what it actually takes to implement an idea. To illustrate this, consider how most people are capable of explaining the way a self-driving car should behave. Anyone can tell you that it should follow traffic regulations, speed limits, road signage, traffic lights, etc..., drive in a calm and predictable manner and do its best to avoid any expected obstacle. Unfortunately, even though the intuition is easily acquired, actually making it happen is anything but easy, and attempting to do it will, more likely than not, uncover a multitude of new and unexpected problems. As a practically-minded person who is really interested in leveraging the power of autonomous systems for real-life applications, I decided that if I really wanted to understand what it takes to work on autonomous systems, I had to get my hands a little dirty. Therefore, this project was implemented almost entirely from the ground-up, using only a computer, MATLAB, online literature and the help of professors and peers. All but the most basic functions were coded from zero or otherwise noted, with an emphasis on a modular software architecture that facilitated the addition of new features.

IV. Results

A. Kinodynamic Goal-Bias RRT for a Simple Car

1. System Description

The first step was to define the system's kinodynamic constraints. The system is defined as kinodynamic because it is simultaneously subjected to kinematic (position bounds, obstacles, velocities limits, etc...) and dynamic constraints (equations of motion). For this simulation, I selected a standard simple car model based on [1], which is represented as a rectangular rigid body with length L and width W that moves in a plane whose configuration is denoted by $q = (x, y, \theta) \in \mathbb{R}^2 \times S^1$ as shown in figure 1.

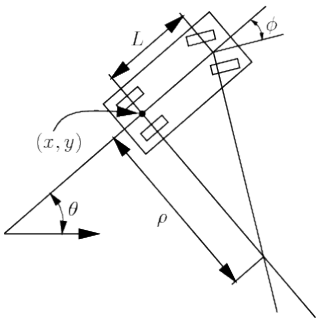

Fig. 1 Simple car model

In real-life, cars are mainly driven through two user inputs: acceleration and steering rate. Therefore, I chose to use a second-order dynamics model [2] in which the control inputs are linear acceleration $u_1 \in \mathbb{R}$ and steering angle rate $u_2 \in \mathbb{R}$. The full model is described by

- State $\mathbf{x} = (x, y, \theta, v, \phi)$
  - Position $(x, y) \in \mathbb{R}$
  - Orientation θ
  - Linear speed $v \in \mathbb{R}$
  - Steering angle $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
- Control inputs $\mathbf{u} = (u_1, u_2)$
  - Linear acceleration $u_1 \in \mathbb{R}$
  - Steering angle rate $u_2 \in \mathbb{R}$
- Equations of motion:

$$\dot{\mathbf{x}} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \frac{v}{L}tan(\phi) \\ u_1 \\ u_2 \end{pmatrix}$$

In order to simplify this investigation, a standard car was defined for use in all scenarios. This car has an initial state at rest, dimensions $L = 4, W = 2.5$, and state and control bounds $v \in [-1/3, 1]$, $\phi \in [-\pi/6, -\pi/6]$, $u_1 \in [-2/3, 2/3]$ and $u_2 \in [-\pi/6, -\pi/6]$. Workspace bounds, obstacles, start state position and orientation, and goal state were individually defined for every specific scenario.

2. Algorithm Selection

For this project, I selected a sampling-based algorithm due to its computational efficiency, probabilistic completeness and my previous experience. Specifically, I chose a rapidly-exploring random tree (RRT), an algorithm which searches a multi-dimensional space by drawing random samples from the state space, and connecting them to the nearest node on an incrementally growing tree [1]. In a goal-biased RRT, the drawn sample is selected to be the goal state a certain percentage of the time (commonly a small percentage, such as 5%). Traditionally, the connecting procedure finds the node on the tree closest to the random sample and then adds a connection. Determining what node is closest requires defining a distance metric which is a non-trivial task and a topic of much discussion in the literature. However, a weighted Euclidean norm is commonly used and, although not without its flaws (particularly when working with mix topologies), it works well enough for the case of a single simple car. Unfortunately, in the case of a system with

kinodynamic constraints, one cannot just rely on distance to determine whether two nodes should be connected, since the transition from one state to another may be physically impossible. For example, if the closest node to the sampled state is very close in terms of car position but includes a difference in orientation of $\Delta\theta = \pi$, adding a connection between these two nodes could imply the car will be able to do a complete turn in a turn radius smaller than what the equations of motion allow for. The way around is to add an extra step to the connecting procedure in which a set controls are randomly added to the closest node, and then propagated for a certain amount of time by numerically integrating the non-linear equations of motion.

3. Pseudocode & Implementation Specifics

This section includes pseudocode for the kinodynamic goal-bias RRT, along with any accompanying functions. Italics denote a variable name, boldface indicates a function and a subscript indicates indexing. Algorithm 1 is a mostly standard implementation of a goal-bias RRT search algorithm. However, some changes were made in how sampled states are handled. Instead of directly appending *state_rand* to the existing search tree, it is first run through **GenerateNewState**, a function which numerically integrates *state_rand* with a set of random inputs and saves the resulting state. Then, the best state is selected, defined by whichever one is closest to *state_rand*. This is done in a two-stage process in which the random inputs are first propagated using an imprecise, but fast, forward Euler integration. Then, only the set of controls which resulted in the best state are integrated again, but this time using a much more precise numerical integrator like MATLAB's **ODE45**. This allows for testing a variety of inputs while saving in computational power. Testing scenarios were specified by generating a *workspace* data structure with the following fields:

- Start state
- Goal state
- Car physical parameters (L, W)
- Car physical constraints (min/max velocity, turning rate, etc...)
- Goal bias probability
- Workspace bounds
- Static obstacle definitions
- Control specifications
- Numerical integration step sizes
- Algorithm parameters (number of iterations, nodes)
- Euclidean norm weights
- Goal tolerance (ϵ)

Algorithm 1 RRT_KG: Kinodynamic goal-bias RRT for a simple car

Input: *workspace*
Output: *path_to_goal*
  state_tree ← *workspace.start* ▷ Initialize tree rooted at start state
  while not at goal **do**
    state_random ← **GenerateRandState**(*workspace*)
    state_closest ← **FindClosest**(*state_rand*, *state_tree)*
    state_new ← **GenerateNewState**(*state_closest*, *state_random*)
    if IsStateValid(*state_new*, *workspace)* **then**
      distance ← **calcDistance**(*state_closest*, *state_new*)
      Add *state_new* to *state_tree* with edge length *distance*
      if calcDistance(*state_closest*, *workspace.goal*) ≤ *workspace.epsilon* **then**
        at goal
      end if
    end if
  end while
  path_to_goal ← **FindPath**(*state_start*, *state_end*, *state_tree*)

Algorithm 2 GenerateRandState

Input: *workspace*
Output: *rand_state*
  if rand $\leq$ *workspace.bias* **then** ▷ **rand**: uniformly distributed number $\in [0, 1]$
    rand_state $\leftarrow$ *workspace.goal*
  else
    Generate random car position (x, y)
    Generate random car heading θ
    Generate random tractor velocity v within bounds defined in *workspace*
    Generate random tractor steering angle ϕ within bounds defined in *workspace*
    rand_state $\leftarrow (x, y, v, \phi, \theta)$
  end if

Algorithm 3 FindClosest

Input: *state_rand, state_tree*
Output: *state_closest*
  for *state* in *state_tree* **do**
    $distances_{state} \leftarrow$ **calcDistance**(*state*, *state_rand*)
  end for
  state_closest $\leftarrow$ $state\_tree_{\min(distances)}$

Algorithm 4 GenerateNewState

Input: *state_closest, state_random, workspace*
Output: *state_new*
  Generate a (*workspace.num_controls* $\times$ 2) vector **u** of random control inputs.
  for $\mathbf{u}_i$ in **u do**
    $state\_euler_i \leftarrow$ **ForwardEuler**(state_closest, $\mathbf{u}_i$) ▷ Standard forward Euler integration
  end for
  state_best $\leftarrow$ **FindClosest**(*state_rand*, *state_euler)*
  state_new $\leftarrow$ **ODE45**(*state_best*, $\mathbf{u}_i$) ▷ MATLAB function

Algorithm 5 IsStateValid

Input: *state_new, workspace*
Output: *validity*
  valid_phys $\leftarrow$ any physical constraints (v, ϕ, etc...) within bounds
  valid_bound $\leftarrow$ car within workspace bounds
  valid_static $\leftarrow$ car does not collide with obstacles
  if *valid_phys* & *valid_bound* & *valid_static* **then**
    validity $\leftarrow$ TRUE
  else
    validity $\leftarrow$ FALSE
  end if

Algorithm 6 calcDistance

Input: *state_1, state_2*
Output: *distance*
  distance $\leftarrow$ **WeightedEuclideanNorm**(*state_1, state_2*) ▷ Can easily swap out for another metric

Algorithm 7 FindPath

Input: *state_start*, *state_end*, *state_tree*
Output: *path_to_goal*
  path_to_goal $\leftarrow$ **Dijkstra**(*state_start*, *state_end*, *state_tree*) ▷ Easily swapped out

4. Implementation Example

An example scenario showing a path generated by algorithm 1 in action is shown in figure 2. For this scenario, $\epsilon = 1$ and integration steps were set to 1 second. The initial car state is displayed as a green box at the bottom of the figure, while the desired final goal state is displayed as a red box. The gray box in the center of the figure denotes an obstacle which the car must avoid. Finally, the blue and and red dots represent the finalized state tree and the final planned path from start to goal, respectively.

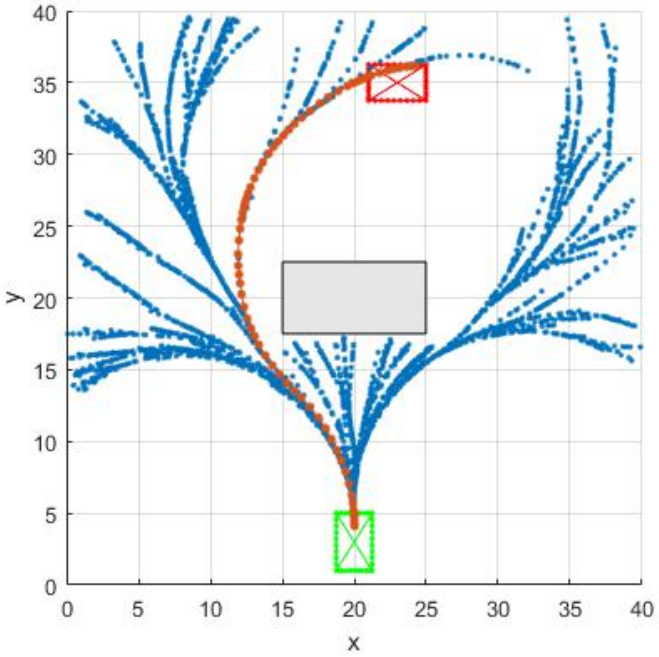

Fig. 2 Kinodynamic Goal-Bias RRT Example Scenario

B. Dynamic environment support

The next step was to adapt the algorithm to deal with moving obstacles so it could deal with other moving cars or pedestrians. For this section, I assumed that the car has perfect knowledge of the position of every obstacle at every point in time.

1. Pseudocode & Implementation Details

Fortunately, the modularity of the previously developed code meant that only relatively minor changes had to be made to the original algorithm. Specifically, **IsStateValid**, the function in charge of verifying whether a new state was valid or not, had to be modified to account for the different positions dynamic obstacles took over time. This meant that a time had to be associated with every state, which was done by noting that the integration times between states was set to a constant value and it was therefore only necessary to count the number of nodes to the start to determine to state's corresponding time. The modified **IsStateValid** is seen in algorithm 8. The *workspace* definition also had to be modified to correctly describe how a dynamic obstacle should move over time. This was done by implementing a dynamic obstacle object with a set of positions and associated times. If **IsStateValid** queried for the position of an obstacle at an intermediate time, position would be determined through linear interpolation. This implementation meant that the dynamic obstacles were not able to maneuver in any highly complex ways and could not rotate, but it was still enough to simulate a pedestrian crossing a street or another moving car. This new algorithm will be referred to as Dynamic Kinodynamic Goal-Bias RRT (**RRT_KGD**). For brevity, only functions with considerable architectural changes are included again.

Algorithm 8 IsStateValid

Input: *state_new, state_time, workspace*
Output: *validity*
  valid_phys ← any physical constraints (v, ϕ, etc...) within bounds
  valid_bound ← car within workspace bounds
  valid_static ← car does not collide with any static obstacles
  valid_dyn ← car at time *state_time* does not collide with any dynamic obstacles
  if *valid_phys* & *valid_bound* & *valid_static* & *valid_dyn* **then**
    validity ← TRUE
  else
    validity ← FALSE
  end if

2. Implementation Example

Figure 3 shows an example scenario simulating a situation in which a self-driving car moving from left to right is trying to exit a two-lane highway with other moving cars on the road (represented by two yellow rectangles). Both cars are moving parallel to the self-driving car, with the bottom one at a higher speed than the top one. In order to successfully reach its goal, the self-driving car follows a plan in which it first waits for the bottom car to drive by, and then turns into the exit road. This is all done while driving fast enough so that the car behind does not catch up. For an animated visualization of this scenario, see `dynamic.gif` in the companion `.zip` file.

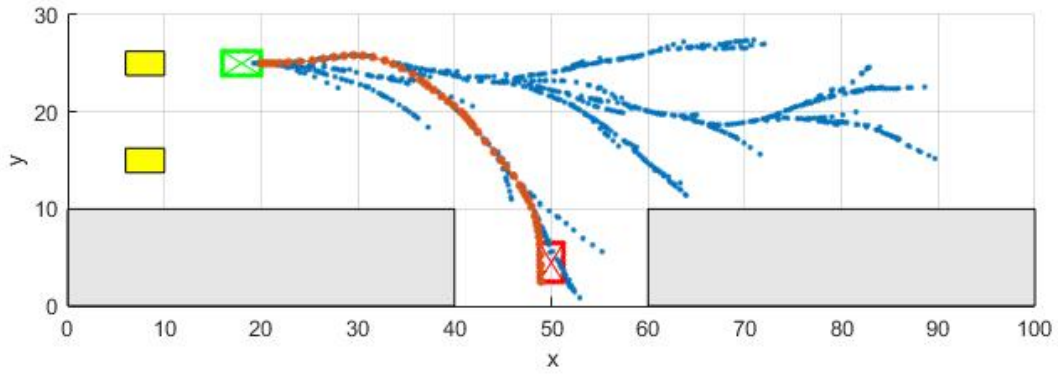

Fig. 3 Lane changing and exit with Dynamic Kinodynamic Goal-Bias RRT

C. Sensor Suite Simulation and Re-connecting Procedure

Now that the self-driving algorithm was working well, I decided to add a simulated sensor suite in the form of a radial proximity sensor which could detect unexpected obstacles within a certain radius and re-plan in case of collisions. This new algorithm can generate an initial plan, follow it, detect new obstacles and generate a new path to goal if a conflict is found. However, generating an entirely new plan with RRT is a computationally expensive task, so in order to take advantage of the work done to compute the initial path, I added a re-connection procedure inspired by [3]. This procedure removes only the problematic parts of the original path and then attempts to reconnect back to it at a point which is heuristically deemed to be far away from the recently discovered obstacle to avoid any collisions, as well as reachable in terms of the car dynamics. This increases computational efficiency by taking advantage of the work done to previously explore other areas of the environment.

For this section I again made the simplifying assumption of perfect knowledge of any obstacles once it is detected. That is, the car does not originally plan for unknown obstacles, but once it discovers them, it is given the full trajectory of the obstacle at any point in time. This assumption is justified by the rationale that in a real-life situation, a self-driving car would probably have some sort of estimation mechanism/module that could then provide a best guess of the trajectory of an unknown obstacle to the planning module.

Finally, it is worth mentioning that at this point in the investigation I decided to forego sub-task 4) in the interest of further refining and upgrading the RRT-based code-base. I decided to add more robustness and features (like the regrow procedure) before spending time coding up an implementation of a PDST planner. Moreover, once a PDST planner has been is ready in the form of a function, the rest of the code-base is modular enough such that swapping it in would only require changing a few lines of code.

1. Pseudocode & Implementation Details

Implementing this change required coding a new algorithm **RRT_KGD_regrow** (algorithm 9) which generates a path using **RRT_KGD**, follows it, detects any new obstacles, and then re-plans as necessary. Exact implementation details were slightly different than what is shown here to account for the detection of multiple new obstacles as well as proper connections between the original path and the bridging paths generated by the regrowing procedure.

Algorithm 9 RRT_KGD_regrow: Dynamic kinodynamic goal-bias RRT with regrow procedure

Input: *workspace*
Output: *path_to_goal*
  path_to_goal ← **RRT_KGD**(*workspace.start, workspace.goal, workspace*)
  while not at goal **do**
    for *state_current* in *path_to_goal* **do** ▷ Follow original path
      if ObstacleDetected(*state_current*, *workspace*) **then**
        if PathCollision(*state_path*, *workspace*) **then**
          goal_temp ← **FindNewGoal**(*state_path*, *workspace*)
          path_to_temp ← **RRT_KGD**(*state_current, goal_temp, workspace*) ▷ Bridge to original path
          path_to_goal ← **CombPaths**(*path_to_goal*, *path_to_temp*) ▷ Combine both paths
        end if
      end if
    end for
  end while

Algorithm 10 ObstacleDetected

Input: *state_current, workspace*
Output: *detected*
  for *hidden_obstacle* in *workspace* **do**
    if *hidden_obstacle* within *workspace.detect_radius* of *state_current* **then**
      detected ← TRUE
    else
      detected ← FALSE
    end if
  end for

Algorithm 11 PathCollision

Input: *state_path*, *workspace*
Output: *collision*

```
for state in state_path do                          ▷ Check all states in given path
    for detected_obstacle in workspace do           ▷ Check all obstacles
        if hidden_obstacle at state.time collides with state then
            collision ← TRUE
        else
            collision ← FALSE
        end if
    end for
end for
```

Algorithm 12 FindNewGoal

Input: *state_path*, *workspace*
Output: *goal_temp*

goal_temp ← first *state* in *state_path* that is *workspace.detect_radius* from any hidden obstacles. ▷ Can replace for another heuristic

Algorithm 13 CombPaths

Input: *path_to_goal, path_to_temp*
Output: *path_to_goal*

path_to_goal ← section with collision in *path_to_goal* replaced by *path_to_temp*

2. Implementation Example

Figure 4 shows a test scenario in which a self-driving car is moving along a busy highway with three other cars. Its goal is to exit the highway and move into a parallel street. The self-driving car has perfect knowledge of the trajectories for the other cars, the rationale being that in a not-so-distant future, cars will be able to broadcast their planned trajectories to the rest of traffic. However, the self-driving car has no idea that a pedestrian, represented by the small yellow block positioned roughly at $(60, 10)$, will be attempting to cross the highway exit. However, the car is equipped with a proximity sensor which is represented by a dotted yellow circle. This example is meant to display the importance of the re-planning procedure, as the initial computation takes quite a bit of time to fully explore the narrow side-street, and having to re-explore it entirely after detecting a pedestrian collision would not be very efficient.

Shown in blue and red are the initial search tree and planned path, respectively. The path successfully avoids colliding with any of the cars on the street. However, as the car moves along it eventually senses the pedestrian, it determines that it needs to generate a new search tree to avoid a collision. This is shown in figure 4 and can be seen in an animated visualization in `regrow.gif`. When watching the animation note that the car passes very close to the crossing pedestrian. Technically, no collision occurs, but in a real-life situation this would not be acceptable and a safety radius around any moving obstacles should probably be implemented. Moreover, politeness dictates that in most cases, a car should wait for a pedestrian to cross a street. In this case however, the planned path has the car speeding and passing in front of the pedestrian. Finally, note the "jump" in the car's state once it finishes bridging across to the original path. This happens because the final state in the bridge path does not exactly line up with the state it is trying to connect to. This could certainly be problematic in a real-life situation but, assuming the chosen temporary goal state is reachable, can be avoided by sampling more states until a better match is found.

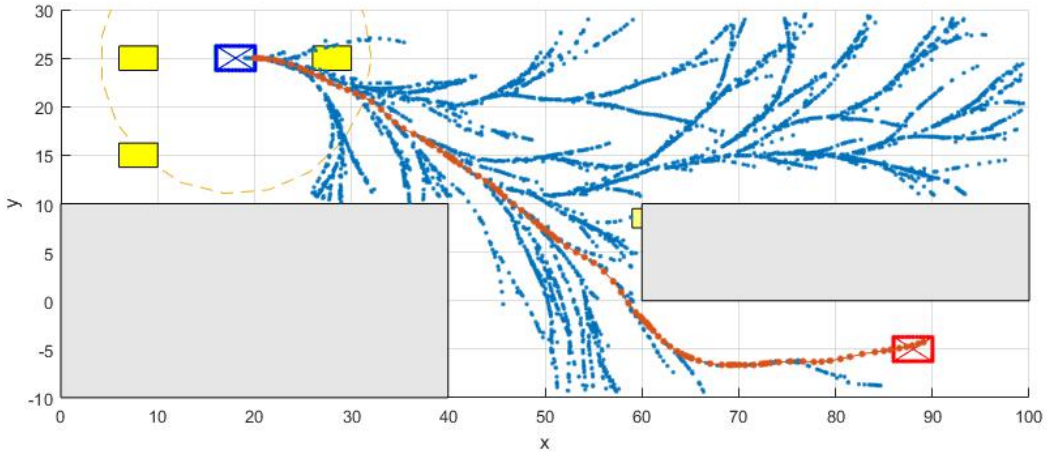

Fig. 4 RRT_KGD_regrow initial path

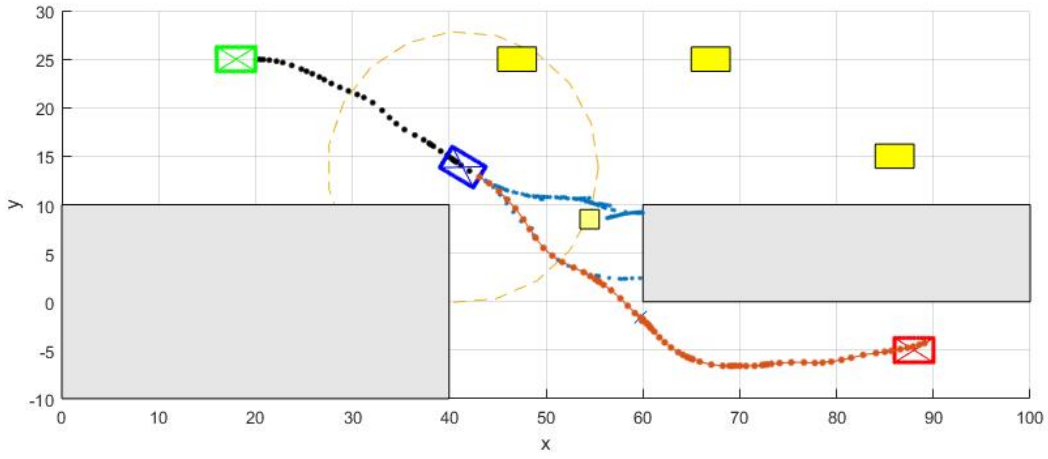

Fig. 5 RRT_KGD_regrow regrowth and final path

V. Conclusion and Future Work

This investigation successfully designed and implemented **RRT_KGG_regrow**, a goal-biased RRT planning algorithm which respects the kinodynamic constraints of a self-driving can, and has support for a dynamic environment with unexpected obstacles. The primary objective of this project was for me to learn about the practical considerations of implementing a self-driving car and I would say it was certainly successful in that respect. When describing the methodology I described the importance of practical implementations due to their uncanny ability to, almost without fail, uncover problems and questions that had not previously been considered. Unsurprisingly, that was the case for this project and as I was working through it I found myself asking questions such as: how does one define distances? How do you decide when to stop running the algorithm? How is an obstacle defined? How do you define a collision with an obstacle? In real-life, collision means physically running into an object, but how does a self-driving know it has "run into" a obstacle? What does "it" even mean in the context of a computer simulation? Practically implementing this algorithm required me to consider all of these questions and in the process gave me a much better understanding of just what is needed to apply many of the concepts we learned in class.

As I progressed through this project and got a taste of the complexity of the problem, I found myself constantly fighting off scope creep as I thought of newer and better ways of implementing a solution before I even had a first version working. Alas, my time was limited and as the project deadline loomed I was forced to limit myself to the results presented in this paper. I am proud of what I was able to complete, but I am also hungry to learn more about this subject and as such I kept a record of ideas for future work and improvements:

- Performance improvements by implementing the algorithm in a faster language such as C++ or Python
- Simulation of a more complex and realistic sensor suite (lidar, vision, GPS, user input)
- Exploration of other path-planning paradigms/algorithms (such as PDST)
- Exploration of a distance metric that can more intuitively deal with mixed topologies (i.e. angles and positions in the same state vector)
- Improvement of dynamic obstacle description so that more complex motions can be tested (i.e. rotations and curves)
- Integration of a higher fidelity car dynamics model
- Estimation module so that the car can generate a guess for the trajectory of any unexpected moving obstacles (rather than being handed their entire trajectories)
- Smoothing procedure to make the overall driving more "realistic"
- Inclusion of additional non-physical constraints such as traffic regulation or "politeness"

This was perhaps one of the most interesting, informative and rewarding projects I have ever taken on, and I look forward to continuing my exploration of the field of autonomous systems.

VI. References

[1] LaValle, S. (2006). Planning Algorithms. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511546877

[2] Lahijanian, M. ASEN 5519: Algorithmic Path Planning Fall 2021 Lecture Slides

[3] O. Adiyatov and H. A. Varol, "A novel RRT*-based algorithm for motion planning in Dynamic environments," 2017 IEEE International Conference on Mechatronics and Automation (ICMA), 2017, pp. 1416-1421, doi: 10.1109/ICMA.2017.8016024.

VII. Acknowledgments

- Professor Morteza Lahijanian for teaching most of this material, his invaluable comments and contagious passion for the field of autonomous systems.
- My classmate Eli Kravtiz for his idea to speed up numerical integration of the car dynamics by using a two-phase procedure with a mix of forward Euler and ODE45.

VIII. Appendix

All accompanying code, workspace definitions and animations can be found in the companion `.zip` file.