Spline Racer:
Autonomous Racing Path Planning

A flexible path planning framework
6.945 Spring 2021

Nick Stathas
nistath@mit.edu

Introduction

Autonomous vehicle software is one of the most popular and challenging AI problems being targeted by large corporations and the scientific community. While the automotive industry is busy solving the problem reliably, a new and exciting area of research into autonomous racing has engineers working to build AIs that push race cars to their limits. In this project, I present a framework for autonomous racing path planners called Spline Racer. Spline Racer was invented by Chris Chang, Nick Stathas, and members of the MIT Driverless student group. Much like Lisp is a family of programming languages, Spline Racer is a family of path planning algorithms. In this work, I outline the main components underpinning Spline Racer implementations and provide a naive, but well-structured, open-source reference implementation built from scratch.

Problem Statement

Autonomous racing is a complex problem that has recently been posed in several competitions, including Roborace and the Indy Autonomous Challenge.
One relatively general problem statement for autonomous path planning is:

1. The car must never exceed the track boundaries
2. The car must complete X laps around the track
3. The wall time is the time it takes to complete these laps
4. The race time is the wall time, modified with any penalties and rewards

5. Objects are rectangular entities that move on the racetrack with a trajectory of their choosing
6. Objects will be observable within a ~150 meter radius of the car
7. An interaction between the car and an object begins when their bounding boxes overlap and ends when they stop overlapping
8. There are several types of objects with different considerations:
    a. Static collectibles
        i. fixed in place on the track
        ii. the first interaction with it results in a reward, by subtracting 2 seconds from the race time
    b. Static obstacles
        i. fixed in place on the track
        ii. the first interaction with it results in a penalty, by adding 30 seconds to the race time
    c. Dynamic obstacles
        i. moving at a constant velocity parallel to the track center line
        ii. each interaction with it results in a penalty, by adding 30 seconds to the race time
    d. Non-reactive cars
        i. moving around the track at a dynamically feasible velocity profile, while following a predetermined racing line
        ii. each interaction with it results in a penalty, by adding 30 seconds to the race time
    e. Reactive cars
        i. an agent that is also attempting to minimize its race time in accordance with these rules
        ii. interactions with it are reviewed by the competition officials and a 30 second penalty is applied to one of them

Previous Development

This ruleset incentivizes the creation of an agent that produces pretty interesting emergent behavior. MIT Driverless has developed a planner that can handle up to rule 8.c with great performance relative to our competitors. At the most recent

Roborace event, where rules 8.d and 8.e were not in place, MIT Driverless took second place.
Moving forward with the development and also retroactively fixing bugs and edge cases is proving painstakingly difficult due to the current structure of the code. For the most part, the concept/intent underlying the path planning algorithm and the corresponding code is not very complicated. However, there are many real edge cases and details that need to be properly handled for the successful completion of the race. At the same time, tight timeline constraints and a frequent competition schedule make it difficult to balance rearchitecting these components, fixing old bugs, and implementing new features necessary for the upcoming races.

This Work

In this project, I combine the learnings from 6.945 and our prior implementation of Spline Racer to develop a generic framework of interchangeable parts that can be composed to solve the path planning problem. The foundation of this framework is can be composed into a solution satisfying up to rule 8.c. To avoid distractions, I focus on implementing an efficient core library necessary to express this foundation. Details regarding vehicle dynamics are abstracted for simplicity, yet our implementation can easily be extended to support them.
To navigate the source code, look in the `src/spline_racer/include` and `src/spline_racer/src` directories. Make sure to read README.md for instructions on how to run it.

Components

In this section, I outline the fundamental concepts for Spline Racer, factorize them into components, and describe the relationships between them.

Point

Spline Racer uses the following hierarchy of point types, described in `Geometry.hpp`. Here A inherits from B, means that type B has all fields A has in addition to the ones described.

- Point
  float x; float y;
- Pose, inherits from Point
  float yaw; // a direction in radians relative to north
- PathPose, inherits from Pose
  float k; // curvature
- State, inherits from Pose
  float v; // velocity
- Setpoint, inherits from PathPose and State
  float a; // acceleration

The root Point class implements fundamental methods such as vector addition/subtraction, scalar multiplication/division, elementwise abs, distance from origin, distance from another point, and a predicate for whether a point is in an axis aligned bounding box.
The nice thing about this implementation is that I can pass a type lower in the hierarchy, such as Pose, into a function accepting a reference to a Point argument and it works transparently without copying or conversion.
I define custom C++20 concepts (which are essentially compile time predicates) for is_point<T>, which is true if T is derived from Point. I also define is_between<T, A, B> which is true if T is in between A and B in the type hierarchy. I use this to impose constraints on the type arguments of flexible generic methods that can accept or output point types within a specific range of the type hierarchy (see the Spline subsection).

Path

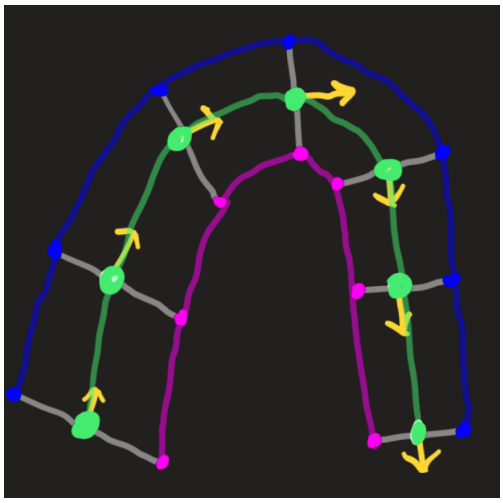
A path is a sequence of PathPose the vehicle might intend to travel through. PathPose contains a curvature, which is necessary to determine whether the path is feasible for the vehicle to drive (vehicles have a minimum turning radius which implies a maximum curvature).
Path is a PointVector<PathPose>.
PointVector is a generic container based on std::vector, with an additional method called get_length. Whoever constructs a PointVector might already know its length and can provide it at construction time. When another component needs to know the length of a path, it uses get_path. If the length is available, it is returned.

If not, it is computed by summing the pairwise distances between consecutive points and memoizing the result.

Track

The algorithm needs to know the boundaries and shape of the track. It needs the ability to represent the track which it loads from a map file generated by the competition officials. It needs to be able to answer queries about whether a point is outside the track and, more generally, whether a trajectory exceeds the track bounds and at which locations.

Track is a wrapper around an std::vector<TrackPose>. TrackPose inherits from Pose and extends it with a left width and a right width value. As shown in the figure below, an ordered and circular sequence of TrackPose polygonally defines the shape of the track and the travel direction:



Visualizing a sequence of TrackPose.
The green points with yellow arrows are the poses stored in the sequence.

The load_track function accepts an std::istream, which is an input stream such as an open file, and a transform. Each line of the user-prepared input file represents a TrackPose in CSV format. load_track converts each line to a TrackPose using a small and extensible parser and transforms it using the provided transform function. An overload of load_track accepts a buffer argument in lieu of the transform function, and calls the former load_track with a transform that shrinks the width by the buffer in either direction. This is used to account for the width of the car and potentially add a safety margin.

Iterator

Since the sequence of track points is circular, I implement a custom C++ iterator that tracks the index into the std::vector<TrackPose> in addition to a lap count.
If an iterator is incremented past the end of the vector or decremented past the beginning, the lap count is respectively incremented/decremented and the track index is brought back in bounds in a modular fashion. Our iterator implements std::random_access_iterator, the standard describing iterators that can be incremented and decremented by arbitrary amounts in constant time, using modular arithmetic.
Two such iterators may be lexicographically compared, first by the lap and then by the track index. This could allow us to answer the question of whether a point inside the track bounds is leading another from the perspective of the race. To do this, we need a way to project arbitrary points to the closest point on the track. Thus, Track implements a method find_idx, which takes a point and linearly scans the TrackPoses, returning the index of the one closest to the given point. find_iterator accepts a point and a lap count and uses find_idx to return an iterator for the given point at the given lap. This query can also be used to check whether an arbitrary point is in bounds: first find the closest point on the track and then confirm that it is within the width of the track at that location.
The design of spline racer is such that this find_idx query will be done frequently. A linear scan is too costly for complex scenarios, which is why I implement PrecomputedTrack. PrecomputedTrack extends track with a TrackIdxMap, a discretized 2D grid which is filled (in parallel with OpenMP) using Track::find_idx at construction time. At query time, PrecomputedTrack::find_idx calls TrackIdxMap::find to locate the grid cell containing that query point (using rounded division), and a small correction is applied to account for discretization error. PrecomputedTrack::find_idx and find_iterator thus run in constant time.

OnTrack<T>

OnTrack<T> (where T is a point) is a generic class extending T with a Track iterator, effectively representing a T that is on a particular track. T is constrained using the is_point<T> concept/predicate.

Racing Path

Our path planning algorithm might rely on a precomputed optimal racing path. It needs to load this from a file or from another component that computes the racing path in the background to account for varying track conditions during the race.
For this implementation of spline racer, the default path is the centerline of the track.

Path Interpolator

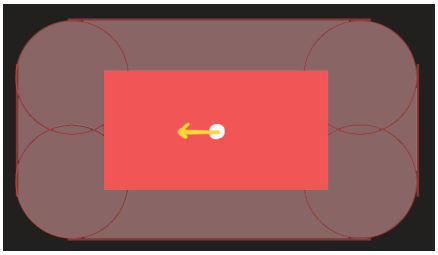
Our algorithm will pick several waypoints that it believes are necessary for the car to travel through. A path interpolator is responsible for generating a legal Path that traverses through these waypoints, stays within the track boundaries, and is feasible for the car to drive (respects minimum turning radius et al.).

Spline

One such path interpolator can be a cubic spline. Splines don't necessarily respect the track boundaries, so they might need to add additional waypoints to help them generate such legal paths.
Also there exist different kinds of splines with different properties: cubic splines have continuous second derivatives, spiro splines have continuous rate of change of curvature. These properties are important for racing.
In this work, I implement a cubic spline interpolator. When CubicSpline is constructed, a system of equations is solved to calculate the spline coefficients for a curvature continuous spline. CubicSpline::interpolate<T>(step_size) (where T must satisfy is_between<T, Point, PathPose>) returns a PointVector<T> of regularly spaced T (points). The nice thing here is that if the user wants T=Point, the spline coefficients are used to interpolate. If the user wants T=Pose, the first-order derivatives are additionally calculated to get the yaw. If the user wants T=PathPose, the second-order derivatives are computed to get the curvature. The implementation for all 3 of these types shares the same code and enables the derivatives calculation, at compile time, only when it is necessary, using constexpr-if.
Another neat trick is that the length of the path can be efficiently computed directly from the spline coefficients and is provided to the result PointVector when it is constructed.

Object

An object has a rectangular shape and an associated cost (positive is penalty, negative is reward) measured in seconds. To represent its shape, I define a Box, which extends Pose (representing the center point and the direction) with a width and a length. It defines a Box::contains(Point) predicate to check whether a point is inside the box. A Box provides a dilate method, returning a new box with an increased width and length. Dilating the object boxes allows our path planner to treat our own vehicle as a point mass. In practice, the dilate method should return a RoundedBox, with rounded edges, but I leave that as an exercise for our competitors.



A Box, defined by the white pose and yellow arrow, with a width and a length. Underneath it is the RoundedBox that Box::dilate should return.

In addition to a Box and a cost value, an Object has a fixed-size array of its four corners (of type OnTrack<Point>) in order of their track index.

Solver

The solver uses all of the above components to produce the trajectory that our car should follow. This is a computationally intensive process that might generate many trajectories before it succeeds. There are exponentially many decisions to make.

Abort Conditions

The solver has a tight deadline for computing a trajectory — the car moves at 100kph and has to react to objects that can be as close as a few meters away, leaving it with split seconds of reaction time. Thus it must be possible to interrupt the computation at any time and ask for the best answer.

Search Tree

A simple approach is to consider the objects in the order of the track iterator of their front corner. This is inspired from how an average human would react to the objects directly in front before they think about the objects further ahead. For each of these objects, we must decide whether to ignore it, overtake to the right or left, or collide with it. There are also several ways to collide with an object — we can collide head-on from the front, or from the right or left.

Let us compress this action space into 4 actions (defined in enum Decision): ignore, collide on the front, interact on the left, and interact on the right. For objects with a collision penalty, interacting means avoiding it. For objects with a collision reward, it means colliding with it while passing it on that particular side. The waypoints that correspond with each of these decisions

Thus, each node in our search tree has a reference to an object, the decision it has taken for that object, a std::shared_ptr<Node> (a reference counted pointer) to its parent, and a heuristic cost estimating how good that decision is in the context of all the parent decisions.

Once all the objects have been considered, we can walk up the parents to extract the decisions and the waypoints they imply using the recursive get_decisions and get_waypoints, which place their output in the output iterator passed as an argument.

Path Cost

calculate_total_cost is used to calculate the cost of a path measured in seconds. It is the summation of the path's calculate_travel_cost, which is a function of the path parametrized by the vehicle dynamics (in our code we assume the car travels at unit speed) and calculate_object_cost, which is the sum of the costs of all the objects the path collides with, as determined by Object::box::collides.

Search Process

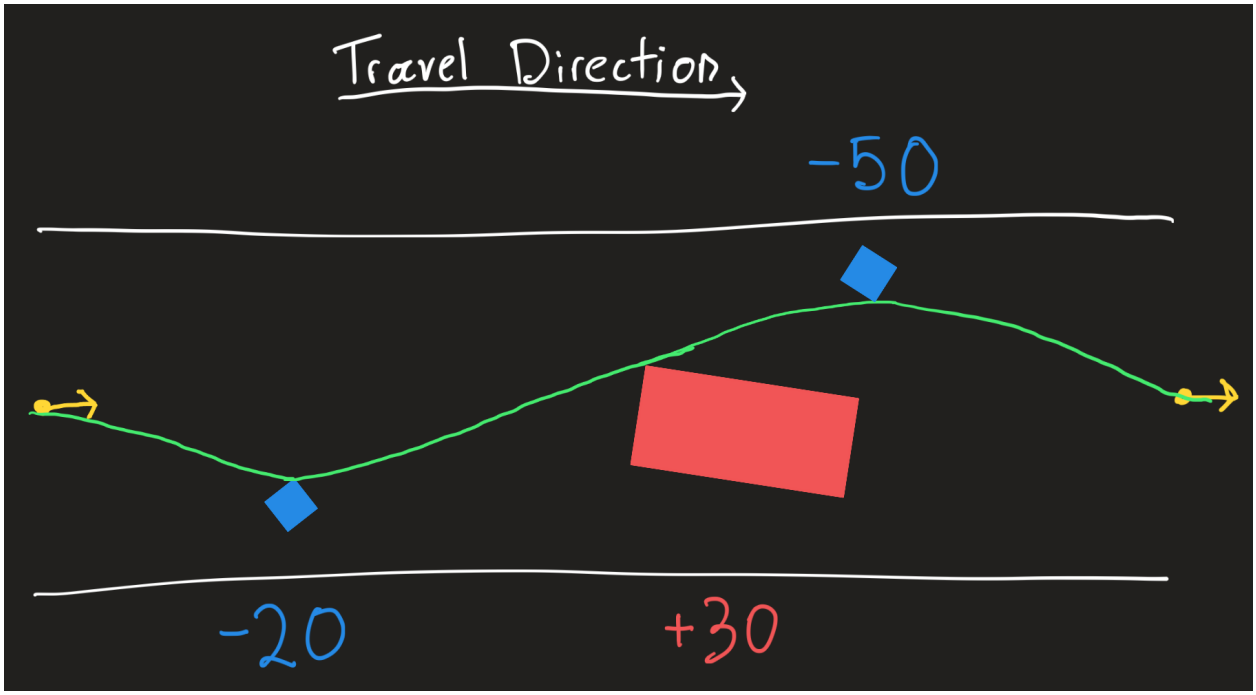
The Search object accepts a list of objects to consider and an initial path proposed to tackle them. The initial path can be a precomputed racing line or just the midline of the track. It initiates the best_path and best_cost variables using initial_path and calculate_total_cost(initial_path) respectively and best_decisions is initially an empty vector of Decision.

This class provides Search::advance, which always does a bounded amount of work, such that it may meet the real-time constraints of the algorithm. It returns false when there are more options left to explore and true when the search tree has been exhausted. The number of unexplored paths, if left unbounded, is exponential in the number of the objects.
The search class maintains a priority queue of unexplored nodes, sorted by increasing heuristic cost. In our implementation, the heuristic cost is random. However, it is worth noting that a cleverly crafted heuristic combined with a pruning/cutoff strategy should lead to a reduced effective branching factor, akin to best-first move-ordering in chess engines. A few exercises for the reader.
Search::advance pops the best available node, and constructs a waypoints vector, with the first waypoint being start_pose (the first pose in initial_path), followed by calling get_waypoints to get the points recommended by the node and its parents, and ending in the end_pose (the last pose in initial_path). It uses a PathInterpolator (in our case the CubicSpline) to create a Path. Then it evaluates this path using calculate_total_cost, and if it is the best available, it updates best_path, best_cost, and best_decisions. Finally, it creates unexplored nodes for the next object (if they have not yet been created), assigns their heuristic cost, and inserts them in the priority queue. The insertion operation is in the worst case (in the absence of pruning) logarithmic in the number of unexplored paths, equivalent to linear in the number of objects. The number of remaining nodes is tracked in a map from an object reference to the number of unexplored nodes referencing that object.
This search process is interesting because the search tree and move generation are lazily expanded. This allows us to postpone the exponential cost of expanding the search tree until it is absolutely necessary. Search::advance is continually called until our abort conditions become true. The nodes are evaluated in a best-first order, which, when paired with a meaningful heuristic, should produce good results earlier in the search, making it an anytime algorithm.

Example Output

In main.cpp, I define an example scenario and interrogate the search process (by accessing best_cost and best_decisions) after each call to Search::advance. The

scenario in question is visualized below, with the optimal path (the one the algorithm converges to and with which I agree) visualized in green.



Here is the output of the program:

```
Track loaded and precomputed in 4.34146 seconds
Object with cost: -20 has corners on track indices 202 203 203 204
Object with cost: 30 has corners on track indices 205 208 212 215
Object with cost: -50 has corners on track indices 217 218 218 219
Default path has travel cost 38.9929
Default path has total cost 68.9929
New best path has total cost 49.2058 with decisions: (Object Cost: -20,
Decision: Right)
New best path has total cost 49.1199 with decisions: (Object Cost: -20,
Decision: Collide)
New best path has total cost 49.0775 with decisions: (Object Cost: -20,
Decision: Left)
New best path has total cost 19.4591 with decisions: (Object Cost: -20,
Decision: Left) (Object Cost: 30, Decision: Left)
New best path has total cost 3.51051 with decisions: (Object Cost: -20,
Decision: Left) (Object Cost: 30, Decision: Left) (Object Cost: -50,
Decision: Collide)
```

```
New best path has total cost 2.96748 with decisions: (Object Cost: -20, Decision: Left) (Object Cost: 30, Decision: Left) (Object Cost: -50, Decision: Right)
```