Project 2: Nonholonomic Control

Reid Dye, Eugene Ng
Mechanical Engineering
University of California, Berkeley
{reid.dye, eugeneng}@berkeley.edu

Daniel Municio
Undeclared and Rivian Intern
University of California, Berkeley
danielmunicio@berkeley.edu

***Abstract*—This paper describes the implementations for three second order nonholonomic systems (cars) including, modified RRT planner, optimization-based planner, and sinusoidal steering methods. Furthermore this paper describes methods for developing closed-loop controllers for these paths**

***Index Terms*—RRT, Optimization Planner, Sinusoidal Steering**

I. Methods

In this section methods are discussed for implementation for Modified RRT, Optimization, and Sinusoidal Steering based planners.

A. Modified RRT

In this implementation we follow the standard RRT procedure, where we take a start configuration s and a goal configuration g. Where in the configuration space C class there are defined methods used in the standard RRT implementation to define a valid next action in the path.

1) Distance Metric:
The distance metric we decided on is:

$$d = ||x_1 - x_2||_2^2 \text{ where } x_i = [x, y] \tag{1}$$

$$dist = d^2 + \frac{3}{d + 0.2} + sin\left(\left(\frac{\theta_1 - \theta_2}{2}\right)^2\right) \tag{2}$$

This distance metric includes the standard l_2 norm as well as adding weights for the angle to persuade against large angle changes that cannnot be achieved.

We incorporate the sin function to deal with angle changes. If we had $\theta_1 = 30^o, \theta_2 = 360^o$, just subtracting the two angles to get the difference would yield 330^o degrees, however we know the difference in angle should be 30^o, so the sin operator is applied here. We square the difference between the angle to make sure the output is always positive, and the difference is divided by 2 to lower its impact in the final distance metric.

2) Check Collision:
The function iterates through all obstacles and computes the squared Euclidean distance between the configuration $\mathbf{c}$'s position (x, y) and each obstacle's center:

$$d^2 = (x_o - x)^2 + (y_o - y)^2$$

A collision occurs if the distance between the configuration and the obstacle is less than the sum of their radii:

$$d^2 \leq (r_o + r_c)^2$$

where r_c is the radius associated with the configuration $\mathbf{c}$.

3) Motion Primitives: To get our motion primitives, we created a list of possible inputs that should be checked in the path generation, and then calculated the resulting motion primitive from that input over a time-step dt, using the vehicle dynamics model given in the project spec, and using Euler's method, as shown below:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \\ \phi_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \\ \phi_k \end{bmatrix} + \begin{bmatrix} \cos(\theta)u_{1_k} \\ \sin(\theta)u_{1_k} \\ \frac{1}{l}\tan(\phi)u_{1_k} \\ u_{2_k} \end{bmatrix} * dt$$

We also modified the inputs that create the motion primitives depending on the situation and map. For example, in parallel parking, it's required to back up, but in a simple path planning task, backwards movement won't be required.

Lastly, we experimented with sampling over a large Gaussian distribution over our goal state in the x and y positions. This gave the planner bias to sample more towards the goal state, instead of completely randomly. We kept a very large standard deviation however, to allow the robot to still sample in all directions, to get around obstacles.

In practice, this sampling heuristic worked very well for the parallel parking and point turn plans, as it kept the planner sampling where it should be, while it struggled to get around obstacles in the other plans, due to the sampling bias.

B. Optmization

We took two different optimization approaches, one utilizing the cost function given in the project spec, and one utilizing a cost function based on the total time.

For both approaches, we used the FATROP solver from [1], which takes advantage of the highly banded structure of optimal control problems to solve linear systems involving the optimization's constraint jacobian much faster than other more general solutions. We reformulated the optimization problem by reordering the variables and constraints to clump each stage together, creating a series of dense blocks on the diagonal, each corresponding to one timestep of the optimization.

To enforce the dynamics constrants, we discretized the continuous dynamics of the bicycle model using a 3-step 2nd-order explicit Runge-Kutta method (no corrector term). We used an explicit method because they are much easier to propagate derivatives through than implicit methods, requiring no further work than expressing the method using the CasADi symbolics. We found this custom method, although lower order and far less sophisticated than the IDA and CVODE algorithms available through the `integrator` interface, to be perfectly acceptable in terms of stability and precision.

The key difference between the two optimization problems outside of time, would be that the total time optimization problem has a variable time, which allows us to plan to exactly the final state. The other optimization problem however, has a fixed time, and tries to get close to the final state in the given time, but is minimizing the distance between final state instead of constraining it to be zero.

The setup for the problem is shown below:

$$J = \sum_{i=1}^{N} \Delta t_i \tag{3}$$

$$\begin{aligned}
\text{s.t} \quad t_i &= t_j, \quad \forall i, j \leq N && (4)\\
q_{i+1} &= F(q_i, u_i), \quad \forall i \leq N && (5)\\
u_i &\geq u_{\min}, \quad \forall i && (6)\\
u_i &\leq u_{\max}, \quad \forall i && (7)\\
q_i &\geq q_{\min}, \quad \forall i && (8)\\
q_i &\leq q_{\max}, \quad \forall i && (9)\\
(x_i - obs_{j_x})^2 + (y_i - obs_{j_y})^2 &\geq obs_{j_r}^2, \quad \forall i, j && (10)\\
q_N - q_{goal} &= \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} && (11)
\end{aligned}$$

C. Sinusoidal Steering

In this section we discuss our approach towards our implementation of sinsusoidal steering based off of the paper by Sastry and Murray [3].
Following the algorithm defined, there are individual parts that are controlled (x, ϕ, α, y) in order and the motions are chained together to get our final path. Where the steering for ϕ and α are already implemented, so this will focus on the implementation of steering x and y.

1) Steer x:
We are given a start state and goal state, where states are defined as $[x, y, \theta, \phi]$, and we want to drive the starting x_0 to the goal x_g.
We know that $d = vt$, so we can get $v = \frac{d}{t}$, as v_x (Linear Velocity) is one of our control inputs.
So to steer x, we can apply linear velocity control input $\frac{x_g - x_o}{t_g - t_o}$ for every Δt.

2) Steer y: Based off of the paper, to steer y, we apply control inputs

$$v_1 = a_1 sin(\omega t)$$
$$v_2 = a_2 cos(2\omega t)$$

Where $\omega = \frac{2\pi}{\Delta t}$, we want to find an a_1 and β_1 that satisfies

$$y(\frac{2\pi}{\omega}) = y(0) + \frac{\pi a_1 \beta_1}{\omega}$$

$$\beta_1 = \tfrac{\omega}{\pi} \int_0^{\frac{2\pi}{\omega}} g\left(\int_0^t f\left(\tfrac{a_2}{2\omega}\sin(2\omega\tau) + \phi(0)\right) a_1 \sin(\omega\tau) d\tau + \alpha(0)\right) \sin(\omega t) dt$$

With these two equations in mind, if we fix an a_2, and we know that β_1 is a monotonic function of a_1, we can binary search over values of a_1 to get the a_1, β_1 that satisfy our requirements.

To deal with constraints on y, we understand that y depends on the system dynamics. Which we can tune by changing parameters $[a_1, a_2, \beta_1]$, as we solve for the optimal a_1 and β_1 from the binary search, we can iterate over values of a_2 to deal with these constraints. To check for constraint violations, we check over the minimum and maximum value that we search over for a_1, where we evaluate the $\beta_{1,1}$ and $\beta_{1,2}$ and check for a change in signs to show that there is a root that exists in the function β over our bounds. If there isn't a root, we change a_2 and continue checking until we have a possible solution.

Listing 1. root finding function

```
def bisection(f, a0, b0, tol):
    a, b = a0, b0
    while True:
        p = (a + b) / 2
        val = f(p)
        if abs(val) < tol or (b - a)
            < tol:
            return p
        if val * f(a) < 0.0:
            b = p
        else:
            a = p
```

With this root-finding method, we pass in the rearranged equation to find the roots for

$$f(a_1) = y(0) - y(\frac{2\pi}{\omega}) + \frac{\pi a_1 \beta_1}{\omega}$$

Where β_1 is defined by the function previously.
Once we find the best (a_1, a_2, β_1), we plug them back in to find v_1, v_2, and use a similar method as steer x to get the control inputs for each Δt.

II. Experimental Results

A. *Manipulation Tasks*

1) RRT: Figures [1-3] The performance of the RRT based paths were not the best paths, due to the randomness of RRT and our motion primitives not always being ideal for the task. Furthermore for the point turn, our tolerance of ± 0.1 allows the point turn to have a path where it doesn't move in place or only turn in place.

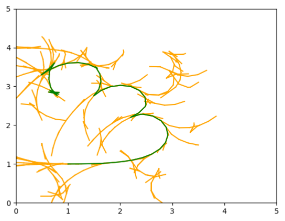


Fig. 1. Parallel Park: $[1, 1, 0, 0]$ to $[1, 3, 0, 0]$

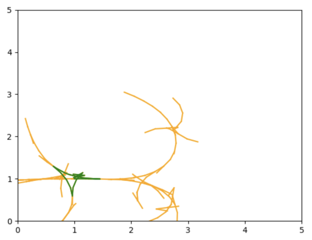


Fig. 2. Point Turn: $[1, 1, 0, 0]$ to $[1, 1, \pi, 0]$

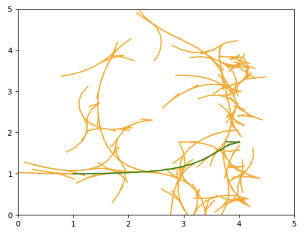


Fig. 3. Simple Motion: $[1, 1, 0, 0]$ to $[3.8, 2, 0, 0]$

2) Optimization: Figures [4-6] The optimization based planners yielded the best results with a smooth and feasible path that we chose for the turtlebot to follow to test our closed-loop controller
3) Sinusoidal Steering: Figures [7-8] Our sinusoidal based path planner worked as expected being able to successfully plan the paths, however for simple motions, the sinusoidal based path planner outputs unnecessary motions, as well as not being able to perform a point turn due to a singularity.

B. *Navigation Tasks*

1) Map 1: Figures [9-10]
2) Map 2: Figures [11-12]

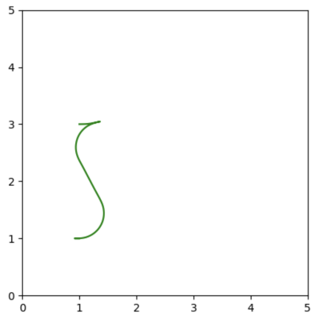


Fig. 4. Parallel Park: $[1, 1, 0, 0]$ to $[1, 3, 0, 0]$

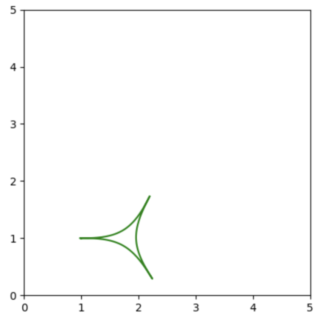


Fig. 5. Point Turn: $[1, 1, 0, 0]$ to $[1, 1, \pi, 0]$

C. *Results of Controller (Fig 13)*

In this section we can see the results of implementing the MPC controller on the turtlebot, where due to the weighting on θ we can notice oscillation in the θ and ϕ errors, however there is still very good tracking overall with position errors $\pm 3.0cm$. Resulting video can be seen in the media section attached.

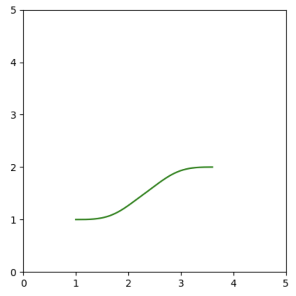


Fig. 6. Simple Motion: $[1, 1, 0, 0]$ to $[3.8, 2, 0, 0]$

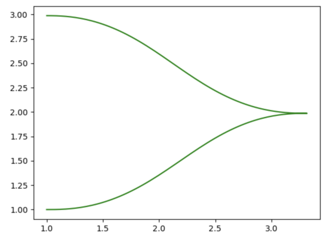


Fig. 7. Parallel Park: $[1, 1, 0, 0]$ to $[1, 3, 0, 0]$

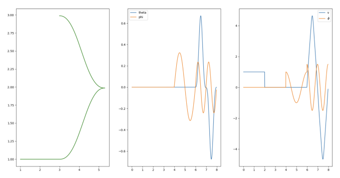


Fig. 8. Simple Motion: $[1, 1, 0, 0]$ to $[3, 3, 0, 0]$

III. Discussion

A. Optimization

The time based optimization controller was far faster than the optimization controller used in the project spec. If we were to utilize CasADI's ability to autogenerate C++ files, it could likely run in real-time on a car. Given that it also creates the most time optimal path, it's far and away the best planner that can be used. Even when adding in a much higher number of obstacles, the planner was still able to run in less than 5 seconds, when using FATROP. A large part of this performance is due to the banded structure we are able to create by carefully adding the constraints to the constraint stack. A more in-depth explanation on how FATROP uses this banded structure to efficiently solve the problem is given in [1]. A visual of the overall banded structure we created in our time-based optimization problem is shown below.

The difference between solve times is also shown below, for various N

B. RRT

RRT's random factor makes it very difficult to compete with Optimization, but with well-tuned motion primitives, it can be useful for exploring completely unknown spaces, where constant updating would be needed, as the algorithm can be made to run much faster, depending on the convergent distance, and the dt.

We had good success with creating motion primitives based on different possible inputs to the vehicle over a given dt. This approach was also very intuitive, as it allowed us to look at the plots created by RRT, and analyze with the perspective as if we were driving a car. One can look at a plot, and say that the driver is turning too much, or not turning hard enough. It also allows us to create paths with extra restrictions. For example, a path that only uses inputs below a certain threshold, or a path that only uses inputs above a certain threshold.

C. Sinusoid Planner

The sinusoid planner would work well for parallel parking, but it did create very inefficent paths, as it's primarily designed for parallel parking. In this aspect however, it did perform very well, and created nice and aesthetically pleasing graphs. But the limited use-case, combined with the lack of ability

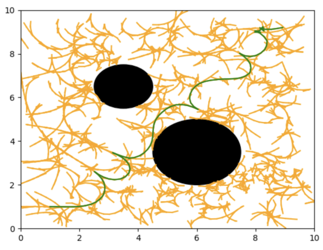


Fig. 9. RRT $[1, 1, 0, 0]$ to $[9, 9, 0, 0]$ 2 Obstacles

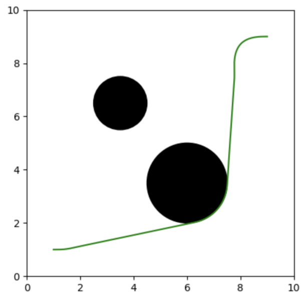


Fig. 10. Optimization $[1, 1, 0, 0]$ to $[9, 9, 0, 0]$ 2 Obstacles

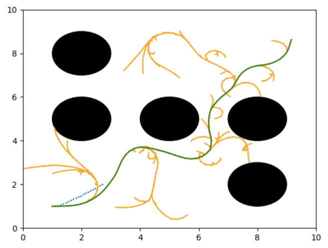


Fig. 11. RRT $[1, 1, 0, 0]$ to $[9, 9, 0, 0]$ 5 Obstacles

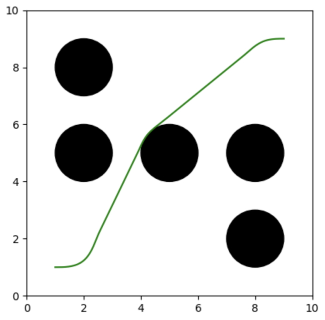
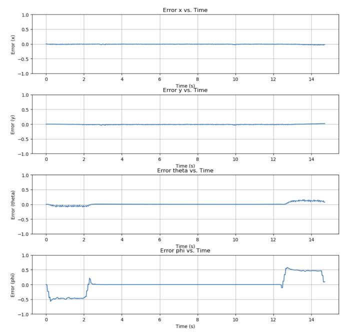
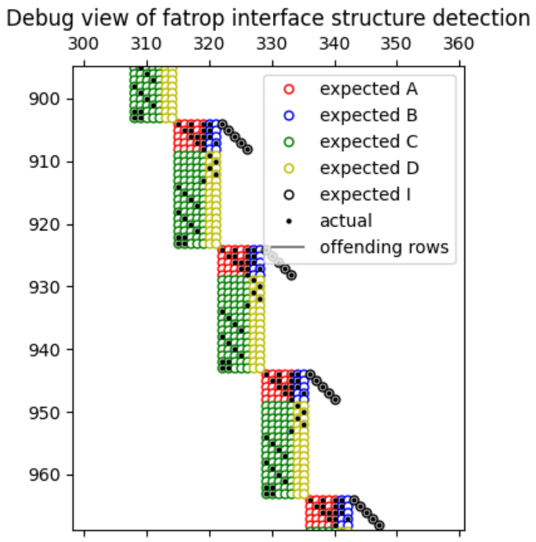


Fig. 12. Optimization $[1, 1, 0, 0]$ to $[9, 9, 0, 0]$ 5 Obstacles



[htb]

Fig. 13. Error vs Time for x, y, θ, ϕ



to model obstacles, makes the planner unlikely to be useful in most real world situations. One upside was that with an efficient root-finding algorithm, the sinusoid was able to generate a path incredibly quickly. One improvement that can be made with the sinusoid planner is allowing it to attempt multiple different fixed a2's before solving, and using a faster and more robust rootfinding algorithm that might be able to increase the searchable region and improve the rate of convergence. Hybrid methods which include the safeguards of the bisection method but achieve higher convergence rates of Newton's method or IQI or the like might be applicable here.

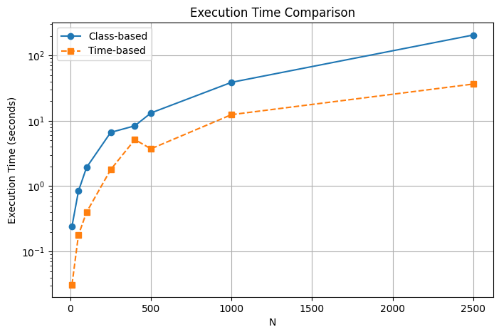


Fig. 14. Comparison between optimization planners

Beyond the poor numerical stability, the main weakness of the sinusoid planner is its inability to account for state and input constraints. More control over the behavior of the system could be achieved by chaining multiple shorter paths generated by the sinusoid method, but that would require increasing the number of direction changes. Perhaps systems where the input derivatives are not constrained might benefit from this, but for systems like ours where the velocity cannot realistically be changed instantaneously, there is little use. There might be some use cases related to sliding mode control, where the system is constrained to within a very small distance of some manifold, where the sinusoidal planner could allow movement in some new directions thanks to very small, high-frequency controls, but that would be very niche.

The abilities of the sinusoid planner are further inhibited by the singularities at $\theta \in \{-90°, 90°\}$, which prevent the planner from turning a significant amount. We developed a procedure to work around this, but did not have time to implement it. In theory, we can perform a change of variables by mirroring our coordinate system according to the following rules:

$$\bar{x} = y \tag{12}$$
$$\bar{y} = x \tag{13}$$
$$\bar{\theta} = \sin(\arccos(\theta)) \tag{14}$$

This transformation moves the singularity by $90°$, so we can keep alternating it and its inverse as we progress our rotation in θ. Conveniently, being a mirror, this transformation is its own inverse, so we simply have to rotate until we get close to a singularity, apply the transformation, then repeat this process until we get to our desired value. Then, if we have mirrored an odd number of times, we mirror once more and rotate once more. Further, if we always mirror at the midpoint between singularities, $\theta = \pm 45°$, then the transform does not change the value of theta, only the direction in which its cosine increases. We thus do not need to compute the comparatively costly $\arccos$ and $\sin$ functions.

D. Control Law

The open loop controller worked well when dt was very small, though it did seem to struggle in the parallel park and point turn where more turning was involved. It also performed better in achieving a goal xy position over achieving a goal orientation. It also struggled heavily as dt got larger, due to there being more time in between recieving inputs, which allowed it to drift farther off path. When we switched over to hardware, the open loop controller really fell apart, as it could not handle the noise in the sensors, and input delay that came with the turtlebots. To remedy this, we developed two different control strategies.

1) Control-Lyapunov Based Design
   We implemented a controller based on the control-lyapunov function [2] where the control law is:

$$v_r = v_{ref}\cos(\theta_e) + k_1 x_e \tag{15}$$
$$\omega = \omega_{ref} + v_{ref}(k_2 y_e + k_3 \sin(\theta_e) \tag{16}$$

   This allowed the simulator to sucessfully track the sinusoidal based planner's inputs much better than open loop control.
2) Model Predictive Control We implemented a nonlinear model-predictive controller with $N = 20$ and $\Delta t = 0.1\text{s}$. We used the same FATROP solver as for the optimization planner, and the same 3-step, 2nd-order explicit Runge-Kutta method for discretization. We used the positive definite cost matrices $Q = \text{diag}(10, 10, 0.1, 0.1)$ and $R = \text{diag}(0.001, 3)$. We attempted to solve the algebraic ricatti equation to find a terminal cost that would be a lyapunov function for the system under LQR control, but the nonholonomic nature meant that the ARE was not solvable. Instead, we used the lyapunov function from the Control-Lyapunov design as our terminal cost:

$$V = \frac{1}{2}(x^2 + y^2) + \frac{1 - \cos\theta}{k_2}$$

   Which is known to be a valid lyapunov function for the closed-loop system under the corresponding control, at least locally. Because we have positive definite state and input costs, a terminal constraint that is a lyapunov function for some simplified (but locally feasible) controller, we can guarantee (local) stability. We cannot necessarily guarantee recursive feasibility, since this is a very nonlinear system and computing the appropriate terminal constraint would be computationally infeasible, but we make the assumption that the path planner will generate a feasible and safe solution to the full CFTOCP, and that we only need to follow it (not bother ourselves with state constraints).

IV. Appendix

Code: https://github.com/danielmunicio/106b/tree/pp/project2
Media: https://tinyurl.com/106bvideos

References

[1] L. Vanroye, A. Sathya, J. De Schutter, and W. Decré, "FATROP: A Fast Constrained Optimal Control Problem Solver for Robot Trajectory Optimization and Control," in Proceedings of the 2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Detroit, MI, USA, Oct. 2023, pp. 10036–10043.

[2] Paden, Brian et al. "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles." IEEE Transactions on Intelligent Vehicles 1 (2016): 33-55.

[3] R. M. Murray and S. S. Sastry. "Nonholonomic motion planning: steering using sinusoids". IEEE Transactions on Automatic Control 38.5 (May 1993).