Abdur-Raheem, Bhunitavel, Jashwanth

Developing a Minimum-Error Trajectory Generation Framework for Autonomous Robotic Navigation with Limited Sensing Capability

Queen Elizabeth's School, Barnet

Abstract

A framework for Bézier-based trajectory generation was developed, using a deep Q-learning algorithm (DQN) to produce splines and accompanying motion profiles. Each trajectory is a continuous sequence of splines passing through prescribed waypoints, in a manner that reduces spatially-dependent jerk and curvature, which are key causes of positional error. We used a replayed memory buffer to train our DQN model, using an *ε-greedy* algorithm for concurrent motion profile optimisation. The framework continues to be iterated upon, with potential for implementation of dynamic obstacle avoidance functionality and, in the long term, more sophisticated multi-agent simulations.

Research Aims

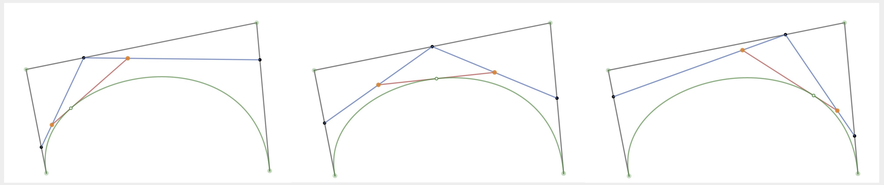
- Develop a framework for automatically generating trajectories that abide to a minimum-jerk model
- Develop a framework for paths that minimise accumulated positional error
- Proof of concept for development of accurate, autonomous trajectory generation for non-holonomic robotic drives with limited sensing capabilities (only internal state feedback)

Background information

Bézier splines

Various splines are used to generate paths which robots can follow, produced using control points and other information given. We chose to use Bézier curves, which can be conceptualised as stacking linear interpolations between given control points from t=0 to t=1 where t represents the proportion which the linearly interpolated points have moved between the two control points.

Below is a cubic Bézier curve with 4 control points, shown at t=0.25, t=0.5, t=0.75. The green curve shows the path that the hollow point takes from t=0 to t=1 and this is the Bézier curve for the control points. This process of stacked linear interpolations is implemented in code, known as De Casteljau's Algorithm.



Another way of expressing Bézier curves that is more useful to manipulate algebraically is called the Bernstein polynomial form, as shown below for a cubic Bézier:

For points P_1, P_2, P_3 and P_4 with coordinates (x_1,y_1), (x_2,y_2), (x_3,y_3) and (x_4,y_4) respectively,

$S_x(t) = x_1(t\text{-}1)^3 + x_2\,3t(1\text{-}t)^2 + x_3\,3t^2(1\text{-}t) + x_4t^3$
$S_y(t) = y_1(t\text{-}1)^3 + y_2\,3t(1\text{-}t)^2 + y_3\,3t^2(1\text{-}t) + y_4t^3$

This polynomial form can us work out the velocity, acceleration, jerk (if t values are taken directly as time values) and curvature (1/radius of the circle that would have the same curvature) of the curve at given t values, along with a bounding box which can be very useful for later application where only certain regions of terrain are traversable.

Bézier curves face two main disadvantages; firstly, the control points do not lie on the curve generated, so for two waypoints that the curve has to pass through, control points have to be generated to form one Bézier curve through them. Additionally, there is no way of always calculating the exact arc length or how far a point has travelled along the Bézier curve for a given t value, so arc length parametrization is required to be able to control the velocity of the point along the curve.

Minimum-Jerk Model for Mechanical Systems

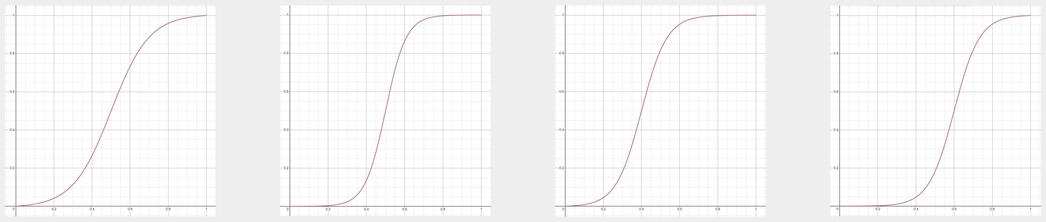
Smoothness of kinematic transitions can be quantified through a jerk cost function (time derivative of acceleration). For holonomic drives, smoothness of driving reduces wheel slippage and skidding, improving the reliability of any internal state models. The jerk cost function is a scalar, defined as

$$\int_{t_1}^{t_2} \dddot{x}(t)^2$$

Where t_1 and t_2 indicate the start and end time and x'''(t) the third derivative of displacement. By implementing this model, we essentially develop a numerical approximation to the Euler-Lagrange equation, minimising our trajectory space against the jerk functional - the distinction is that other parameters are also used to select our chosen trajectory.

Motion Profiling

The motion profile describes the position and velocity of the robot at any given time. We use the Bézier curves for position, and an adjustable sigmoid curve for our distance-time function. Use of a sigmoid curve is prevalent in many motion profiles as it smoothes a robot's speed profile, which tends to minimise errors accumulated. We emphasise however, that this only works for uniaxial trajectories, failing to account for the spatially variant jerk acting on a robot moving in 2D space. The form of sigmoid we use allows us to control the "steepness" of the sigmoid curve (i.e. the maximum velocity it reaches) shown in the left two images, and how early/late in time acceleration occurs show in the right two images - the latter feature will become more useful when incorporating dynamic obstacle avoidance, so for now, it is preset at a fixed value that makes profiles symmetrical.



Sigmoid curves generated that pass through (0,0) and (1,1)

Reinforcement Learning

Reinforcement learning (RL) can improve the algorithm's efficiency of function through repeated simulations or trials. It is perfect for constructing a model for a particular problem (like our use case). We aim to implement our algorithm alongside a Deep-Q Network (DQN). A typical machine learning algorithm will be able to produce an optimal curve for a specific set of waypoints, however DQN (which brings the advantages of Deep Learning to RL) provide more diverse and de-correlated training data as it stores all of our agent's experiences and randomly samples and replays them. This means our algorithm is generalised to any set of waypoints that it is assigned.

Method

Tools Used

We chose to use Python 3 for the development of our model and framework. It facilitates the development of object-oriented programming features, which make troubleshooting and code reusability much easier. In particular, classes allow us to keep data members together in our code - improving organisation. Additionally, the team's mutual familiarity with python allows for simultaneous development of the code.

Architecture and Testing Framework

For a given start point, end point, and set of waypoints, defined within 2D space, we wanted to be able to generate a sequence of Bézier based trajectories, minimised against our cost function, that allowed the robot to pass through each way point and start/end point

We developed classes for splines, motion profiles and agents. Generating usable motion profiling required arc-length parametrization of Bézier curves - n-degree Bézier curves cannot be parameterised exactly, so numerical methods are required. For a Bézier curve B(t) (0<=t<=1), where t comprises the linear continuum of points between 0 and 1, t does not correspond with T (global time) unless B(t) produces a straight line with regularly spaced control points. We overcame this through using a cubic-interpolation mapping function (Madi, 2004), facilitating $T \rightarrow t$ mapping to generate reference points that were regularly-spaced on the curve. With regularly-spaced reference points, we were able to create motion profiles, using sigmoids alterable by parameters (allowing our DQN algorithm to alter motion profiles). Developing a mapping function required an initial estimation of total arc-length; after experimenting with Runge Kutta and Simpson integration approximation methods, we decided to use Simpson's rule, achieving convergence (to 10s.f.) after 8 iterations. We developed custom spline splicing, control line, degree elevation and differentiation functionality to reduce cyclomatic complexity when producing simulations. Simulation and training of the DQN is entirely data-based, not visualised.

To allow the model to alter motion profiles, we developed a parameterised sigmoid model, as described on the right

We placed bounds on m and k through experimentation - k indicates the position of the sigmoid's halfway point (allowing us to accelerate earlier or later), and m indicates the steepness of the speed spike. For a fixed value of k, the graph of maximum velocity against m converges to a linear relationship, so m can be bounded to avoid surpassing a maximum speed using m - we envisage that this will become a useful property when developing this for real-world applications (e.g. m could be correlated with a motor's threshold speed). For now, we intend to set k equal to 0.5, ensuring the sigmoid is symmetric - for future iterations, we intend to make k alterable, allowing for 'stopping and starting' behaviour. Determining a value of m is an action available to the DQN model.

$$s(x) = \frac{a}{1 + e^{-max + c}} + b$$

Where $a = -b - be^{c}$

$$b = \frac{1 + e^{c-m}}{e^{c-m} - e^{c}}$$

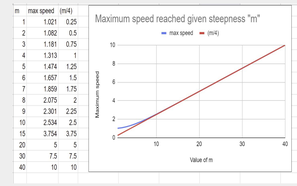
$$c = km$$

$$0.25 \le k \le 0.75$$

$$0 \le m \le 100$$

Such that $s(0) = 0$ and $s(1) = 1$

m	max speed	(m/4)
1	1.021	0.25
2	1.082	0.5
3	1.181	0.75
4	1.313	1
5	1.474	1.25
6	1.657	1.5
7	1.859	1.75
8	2.075	2
9	2.301	2.25
10	2.534	2.5
15	3.754	3.75
20	5	5
30	7.5	7.5
40	10	10



Model and Optimisation functions

Solving for jerk analytically was impossible, as you cannot ensure continuity of two Bézier curves beyond 2nd order derivatives. We thus used a forward Euler method to approximate jerk, calculating the tangential velocity at regularly-spaced T intervals, and using 2nd order changes to find an approximate jerk cost scalar for any given path.
We were able to analytically solve for the curvature at any point on a given path, by using the curvature formula below.

$$K(t) = \frac{|x'(t)y''(t) - y'(t)x''(t)|}{[(x'(t))^2 + (y'(t))^2]^{\frac{3}{2}}}$$

To generate a scalar associated with curvature for each path, we took samples of curvature at each T reference point, and summated the modulus of curvature at each of these reference points - a path with a low sum has a less eccentric path (desirable for the same reasons elucidated in the minimum jerk model). The key benefit to using curvature alongside jerk is to avoid favouring linear paths with small sharp turns at the beginning and end of paths.

Initial Model Approach - Q-Learning

$$r(s(m,\mathbf{j_a},\mathbf{j_b}),a(m,\mathbf{j_a},\mathbf{j_b})) = \left(\sum_{t=t_1}^{t_2} \ddot{x}(t)^2 + \frac{|x'(t)y''(t) - y'(t)x''(t)|}{[(x'(t))^2 + (y'(t))^2]^{\frac{3}{2}}}\right)^{-1}$$

Where s represents the combination of current control points and m values for a spline in a sequence of splines
a represents the set of new possible combinations of m, $\mathbf{j_a}$ and $\mathbf{j_b}$
m is the set of integers from 0 to 100 that the model could possible pick
$\mathbf{j_a}$ is the set of tuples of discrete potential coordinates for the first control point within the 1x1 operating space, separated by intervals of 0.001 (producing a 1000 by 1000 grid)
$\mathbf{j_b}$ is the set of tuples of discrete potential coordinates for the first control point within the 1x1 operating space, separated by intervals of 0.001 (producing a 1000 by 1000 grid)
Note that to ensure continuity between splines, $(\int_n^{n-1} - \mathbf{w}_{PnPn+1}) \times (\int_n^{n+1} - \mathbf{w}_{PnPn+1}) = 0$, where $\mathbf{w}_{PnPn+1}$ is the waypoint between splines p_n and p_{n+1}. This restricts the values that $\mathbf{j_a}$ can take for all splines other than p_1, but j_b can take any value within the operating space

Bellman equation (Q-learning update rule): $Q(s,a) := r(s,a) + \gamma \max_a Q(s',a)$
Where r(s,a) is the immediate reward received from the current spline, Q(s',a) is the maximum Q value yielded from being at state s' (the state after taking action a from state s), γ is the discount factor (which controls the importance of long term rewards) and Q(s,a) is the Q value yielded from being at state s. To encourage exploration, we used an *ε-greedy* approach for m values, allowing for exploration even if a terminal state is reached, such that for some 0<ε<1, we choose the greedy action (the same-best m value) with probability p = 1 - ε, and a random action with probability p = ε. This prevents motion profile optimisation from occurring too quickly, before spline shape has converged

New Approach - Deep Q-Learning

$$Cost = \left[Q(s,a;\theta) - (r(s,a) + \gamma \max_a Q(s',a;\theta))\right]^2$$

Where θ represents the trainable weights of the neural network. We use a neural network to approximate Q values, more suitable for the large number of potential states. In training we ask the agent to select the best action using the current network, and record the state, action, reward and next state it ended up at. The deep architecture allows us to generalise beyond a single set of waypoints – we can select batches from multiple different waypoint sets, thus producing the desirable behaviour

Outcome, Implications and Future Work

$$F = \begin{bmatrix} A & 0 & 0 \\ 0 & A & 0 \\ 0 & 0 & A \end{bmatrix} \text{ with } A = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \qquad G = \begin{bmatrix} B & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & B \end{bmatrix} \text{ with } B = \begin{bmatrix} \frac{1}{2}s^2 \\ s \end{bmatrix}$$

Outcome

Our framework was able to successfully produce trajectories - we are now in the process of conducting sensitivity analysis to evaluate whether trajectories (specifically motion profiles) lie at a local minimum.

Incorporating Dynamic Obstacle Avoidance into Model

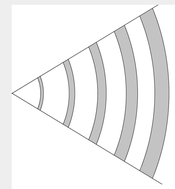
For dynamic obstacle avoidance, the system needs to estimate the motion of each moving obstacle. We intend to use a Kalman filter for tracking obstacles, where every new object is tracked by a separate filter. For every obstacle, both the position and size are estimated, as well as its state relative to the vehicle. the state of the i-th obstacle, the position of the obstacle, its circle radius and linear velocities, are denoted in a vector At. Then, At+1 = FAt + Gwt; where G is the noise gain matrix, and F is defined to the right (where s is the length of a sampling period). A snapshot of the position and the velocity of the moving obstacle is provided at T = 0, with an associated uncertainty in the velocity. At each timestep, we simulate the uncertainty in the robot's estimation of obstacle position; over time, uncertainty in position increases, following a Gaussian distribution (as modelled by the noise gain matrix). In response, the model alters our motion profile, changing the k value within our sigmoid accordingly (achieving the desired starting/stopping behaviour). In this manner, the model now also incorporates certainty/safety into the cost function.

Multi-agent simulation

If the DOA model proves successful, then there is potential to expand the project, to synchronously generate trajectories for multiple robotic agents within an environment. This architecture would bear large similarities to the framework used for DOA, but with centralised control over paths we could begin to observe more intelligent crowd behaviour between agents, an exciting prospect.

Industrial settings

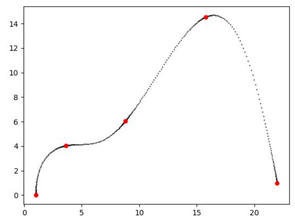
The ability to develop minimum-error paths for robots means that deviations from the path which could cause damage to a work-site or warehouse, or potentially injure people are minimised.



Above: matrices used for Kalman filter/obstacle mapping

Left; Diagram representing increasing uncertainty as time increases using the Kalman filter (each consecutive band represents each increasing timestep)

Below; Example trajectory sequence generated using framework, produced on matplotlib (rescaled for visual clarity)



Madi, M., 2004. 'CLOSED-FORM EXPRESSIONS FOR THE APPROXIMATION OF ARCLENGTH PARAMETERIZATION FOR BÉZIER CURVES', Int. J. Appl. Math. Comput. Sci. [Online]. Accessed (December 15 2021). Available at: http://matwbn.icm.edu.pl/ksiazki/amc/amc14/amc1415.pdf