Dynamic Motion Planning Networks Applied to Dubins Vehicle Dynamics

1st Marshall Vielmetti
Department of Electrical & Computer Engineering
University of Michigan
Ann Arbor, USA
mvielmet@umich.edu

***Abstract*—Online planning for complex, constrained systems is very important as more and more mobile robotics begin to operate in uncontrolled environments. Many standard methods simplify systems, relax constraints, or become infeasible for complex systems. In this paper, we implement the Dynamic Motion Planning Network approach for a vehicle modeled with Dubins dynamics, enabling real-time motion planning in complex environments. In this paper, we discuss expert trajectory generation using RRT* for offline data collection, the model construction and training methods used to create the model, and then evaluate the resulting model on a set of sample environments, using a simulated Dubins vehicle. We furthermore slightly tweak a few aspects of the original paper to better fit this scenario and understand the impact of these changes. Much of this paper is adapted from and borrows directly from [1].** [1]

***Index Terms*—Dynamic Motion Planning, Trajectory Generation, Nonholonomic Constraints**

I. Introduction

Motion planning under dynamics constraints in nontrivial environments remains a core, open problem in robotics. The problem is to find a series of points between given start and goal states which satisfy kinematic and dynamic constraints. When applied to non-holonomic vehicles, such as a car modeled with Dubin's Dynamics [2], the resulting constraints depend on the derivative of the system state. Thus, a trajectory in the systems configuration space may not be actually executable by the system due to these constraints.

Many algorithms exist to solve for dynamically feasible system trajectories offline. Sampling-based algorithms such as *RRT** [3] solve the problem with probabilistic completeness by randomly sampling points in the state space and guarantees asymptotic optimality as the number of samples increases. Due to the asymptotic nature of such algorithms, they often struggle to achieve real-time performance. Additionally, in obstacle-dense environments, sampling algorithms will further struggle to come across feasible trajectories [4].

Many *RRT** variants exist to address these issues. Variants like *RT-RRT** (Real-Time *RRT**) optimizes *RRT** by retaining the tree between iterations, so it does not have to be regrown at every iteration. Other approaches like *NRRT** (Neural *RRT**) use a non-uniform sampling distribution generated by a CNN to return more promising samples, increasing the convergence rate [5].

Reinforcement learning methods have also gained traction for constrained motion planning [6]. However, navigation is an inherently difficult task, and these algorithms suffer from many of the issues common with reinforcement learning based approaches, like the sim-to-real gap, and also suffer from a lack of rigorous dynamic feasibility and safety guarantees.

In this paper, we implement Dynamic-MPNet [1], a variation of the motion planning network approach [7]. Motion-planning networks were originally introduced to bridge the gap between neural and classical motion-planning algorithms. The underlying approach is to pass a neural network an encoded representation of its environment, as well as the current and desired configurations, and the planner will produce a dynamically feasible path between two. A neural network approach promises a flat computation time when compared to a sampling-based approach, and the authors of the original paper achieved significantly faster computations relative to the state of the art in a planning problem for a 7DOF robotic arm. This approach has the benefit of learning from expert-general trajectories and thus does not require a complicated reward function. These expert trajectories can be generated in bulk, offline, and then used to train the model. The Dynamic-MPNet paper extended this approach to a system modeled with non-holonomic, dynamic constraints The implementation details of such an approach are the focus of this paper, as well as an ablation-study type approach to validating the decisions taken by the authors of the original paper.

II. Problem Definition

Let $X \subset R^d$ be a d-dimensional state space, where $d \in \mathbb{N}$. Let $X_{\text{obs}}, X_{\text{free}} \subset X$ where $X \setminus X_{\text{obs}} = X_{\text{free}}$.

Dubins vehicle dynamics are given by the nonlinear model:

$$\dot{x} = f(x, u),$$
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ u \end{bmatrix} \tag{1}$$

Where V is some constant maximum velocity, and $|u|$ is bounded by some maximum turning rate κ. Let U denote the set of valid control inputs.

[1] Code is available online at http://github.com/marshallvielmetti/dynamic_mpnet

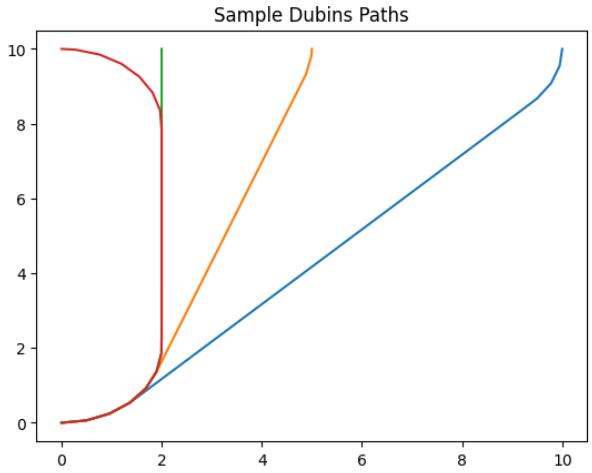

Fig. 1. Examples of Dubin's shortest paths originating from the pose $x = [0, 0, 0]^T$ to $[0, 10, -\pi]^T, [2, 10, \frac{\pi}{2}]^T, [5, 10, \frac{\pi}{2}]^T, [10, 10, \frac{\pi}{2}]^T$

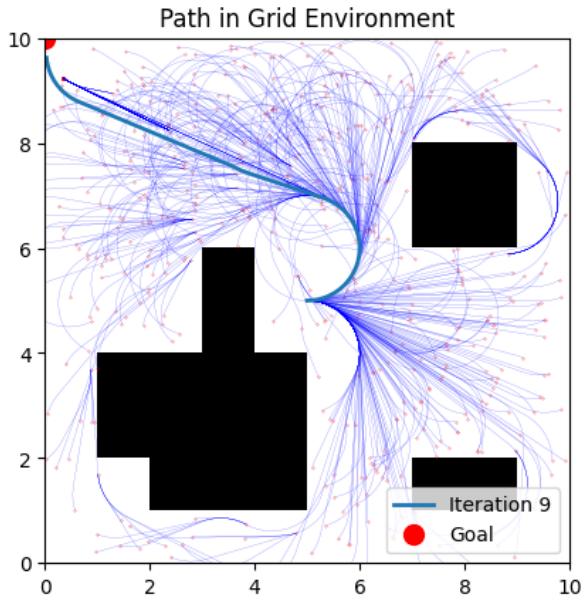

Fig. 2. Sample *RRT** trajectory generated in a 10×10 grid environment. 0.5 unit grid cells were used for the maps. Obstacles were placed randomly, and selected from a set of pre-determined obstacle sizes.

The goal of a trajectory planner is to, given some starting and goal configurations $s_0, s_T \in X_{\text{free}}$, to find a series of points $s_1, ..., s_{T-1}$ such that there exists some control inputs $u_0, \ldots, u_{T-1} \in U$ which drive the system through the series of points. That is to say, the trajectory is dynamically feasible subject to the system constraints.

A 2D Dubins shortest path is defined as the path of minimum length that connects two poses $x \in \mathbb{R}^3$ with arcs of some maximum curvature, and straight lines. As proven in [2], this path consists of either two curve segments and a straight line, or three curves. Some examples of what these paths look like are shown in Fig. 1. These paths can be computed analytically, and are dynamically feasible by vehicles with Dubins dynamics.

III. Implementation

In this paper, we implement the Dynamic MPNet approach for a vehicle modeled by Dubins Dynamics, as given in (1). We first generate reference trajectories using an offline approach. These reference trajectories are then used to train the motion planning network.

A. Data Generation

To train the model, we generated approximately 1200 sample trajectories using an offline RRT* implementation using Dubins path primitives [3]. We generate these trajectories for random pairs of points in generated 12×12 grid environments. Following the recommendations in [1], we containerize the data generation process using Docker allowing us to run many instances in parallel. An example of a generated path and prototypical environment are shown in Fig. 2.

B. Environmental Encoding

The environment is always transformed to be ego-centric about the current robot pose at each time step. This significantly reduces the dimensionality of the input space and also significantly reduces the amount of required training data. Furthermore, we encode the environment into a latent representation, rather than using the raw occupancy grid, to reduce the dimensionality of the network.

To encode the environment, we use a deep convolutional neural network (CNN). We evaluated multiple different sizes for the latent map representation, as shown in figure [**?**], and ultimately settled on a 32-dimensional latent space. The encoder network consists of three convolutional layers, with kernel sizes $[5, 5]$, $[3, 3]$, and $[3, 3]$ with 8, 16, and 32 output channels. After each convolutional layer is a PReLu ([8]) activation function, and after the first two is a MaxPool2d layer with kernel size 2. Following the convolutional layers is a fully connected layer, which reduces the output to a 1D latent representation.

In order to train the encoder model, we also created a convolutional decoder, using transposed convolution layers and upsampling to create an inverse of the original encoder architecture. This allows us to apply a reconstruction loss in model training, ensuring that the encoder accurately captures the environment.

C. Dynamics Modeling Network

The dynamics model is a fully connected deep neural network. There are a total of six layers, with PReLu activation and dropout layers. The dropout layers are used to prevent overfitting. The final layer is passed through a $\tanh$ activation function.

The dynamics model attempts to predict the "next" state s_{t+1} from the current orientation θ_t, a goal pose s_T, and the latent representation of the ego-centric cost map produced by the encoder model. Note that the current state information s_t is not required in its entirety, only the orientation, because the

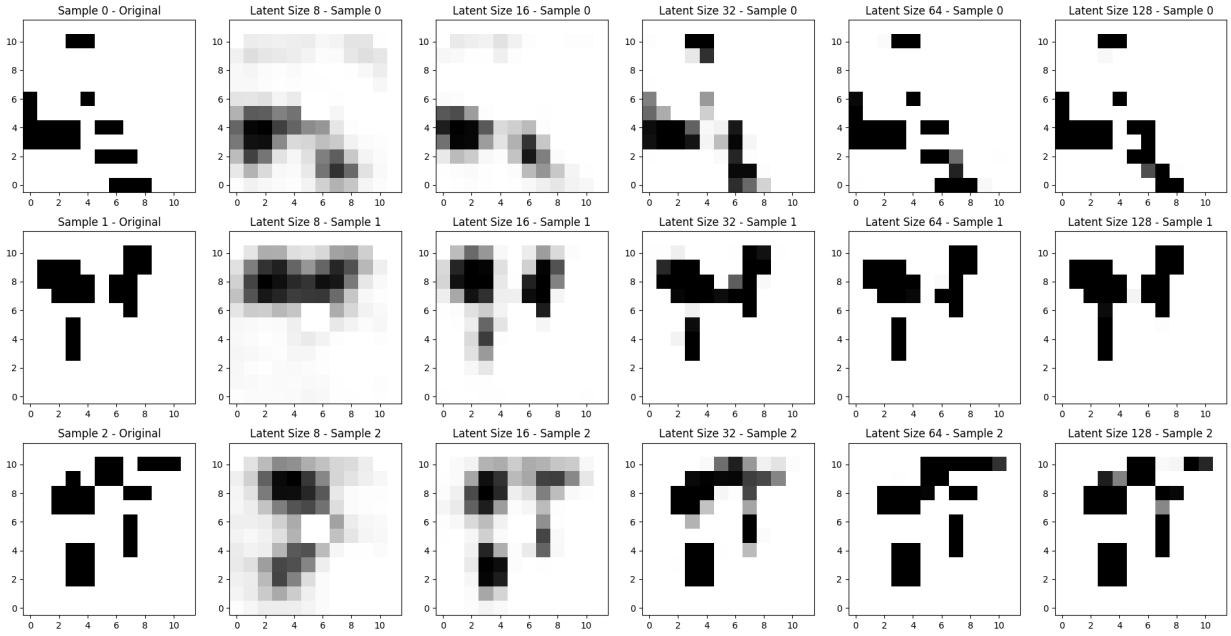

Fig. 3. This figure shows the original map, and the reconstructed map produced by encoding and decoding the map, using a trained model, with different latent representations. As the size of the latent representation increased, so did the reconstruction performance, Based on the results of this comparison, a latent size of 32 was chosen, to trade off reconstruction performance and dimensionality.

goal pose and cost map are translated to be relative to the current position at each iteration.

An overview of the network structure is given by figure 4.

D. Training

The model is trained in an end-to-end fashion. The original dataset consists of arbitrary length expert trajectories. For training, short samples are taken from these trajectories along with the occupancy grid.

This expert trajectory $\{s_0, s_1, \ldots, s_T\}$ consists of the position of the robot at time t, as well as the goal pose s_T. Also provided is the occupancy grid C. Training tuples consist of the terms $(\theta_t, s_t, c_t, s_{t+1})$

Three loss terms are used in the training of the model. First, a position loss is calculated for each pose in the trajectory by evaluating the model at that pose, and taking the MSE with the next pose in the trajectory, resulting in poses $\{\hat{s}_1, \ldots, \hat{s}_T\}$ Second, a rollout loss is calculated by taking the model and attempting to "roll out" the model dynamics from the initial pose, and taking the MSE of the resulting trajectory and the original. This rolled out trajectory is given by $\{x_1, \ldots, x_T\}$. Finally, a reconstruction loss term is calculated by decoding the latent map representation for the points in the original expert trajectory, and calculating the MSE loss compared to the original map. The reconstructed maps around each point in the expert trajectory are given by $\hat{c}_t$

Thus, the loss function about trajectory n is given by:

$$L_n(\theta) = \frac{1}{T}\sum_{i=1}^{T} ||\hat{s}_i - s_i||^2 + ||x_i - s_i||^2 + ||\hat{c}_i - c_i||^2 \quad (2)$$

This differs significantly from the methods used in the original paper, which did not include a rollout loss term nor a reconstruction loss term. The rollout term was included to encourage the model to more closely mimic the expert trajectories and encourage the model to output dynamically feasible trajectories.

IV. Experiment

In order to evaluate the algorithm on real problems, we use Algorithm 1, which includes the necessary additional steps to make the network work within a planning framework. We then construct a nonlinear model predictive control formulation to track the resulting trajectory. Finally, we compare the implementation runtime and performance to the original *RRT** used to generate sample trajectories.

A. Planning

The planner takes as input the current position and a "sub-goal" within the model's range, and the local costmap, and uses this information to generate a dynamically feasible path between the current position and sub-goal.

1) Costmap Padding: The `Pad` function prepares the cost map for use by the planner. Given the pose s_t and global cost map C, it first takes a $l \times l$ subsection of the costmap, centered at the current pose of the robot. This ego-centric map is then padded to $2l \times 2l$ with obstacles. This is necessary so that as the state used by the planner in the rollout shifts away from the center, it still plans only inside the original $l \times l$ grid, as it will perceive the padded areas as obstacles.

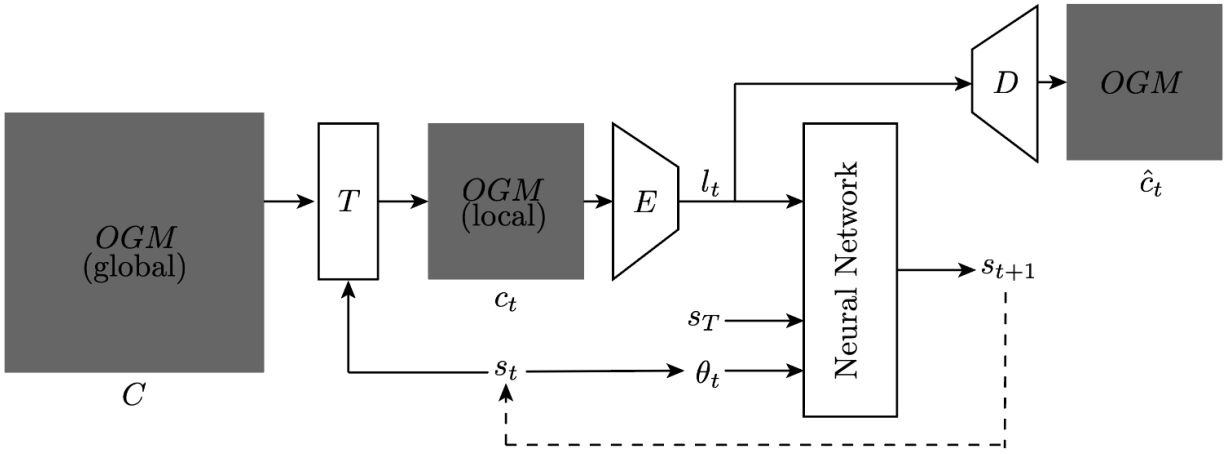

Fig. 4. The structure of the Dynamic MPNet approach. C is the global cost map. T represents the transformation to an egocentric cost map c_t about the current pose, s_t. l_t is the latent representation of the occupancy grid output by the encoder module. $\hat{c}_t$ is the reconstructed occupancy grid created by the decoder. s_T is the goal pose. θ_t is the current pose theta. s_{t+1} is the next predicted state, as output by the network.

2) Steering: The `Steer` function attempts to connect the previous pose to the pose output by the neural network at every iteration, using a Dubins shortest path. This path is then sub-sampled, and checked for collisions. This is to ensure that there exists a dynamically feasible, collision-free path between two poses.
3) Neural Net: The `Network` function represents the forward pass of the network. Given the inputs θ_t, s_T and c_t, representing the current orientation, goal pose, and ego-centric costmap, it generates s_{t+1}, the next step on the path.
4) Transform: The `Transform` function, which represents the block T in Fig. 4, transforms an arbitrary costmap C to be ego-centric to position s_i.

Algorithm 1 $\tau \leftarrow$ Dynamic MPNet $(x_{\text{from}}, x_{goal}, C)$

```
τ ← x_from
x_curr ← x_from
for i ∈ 1...N do
  ĉ ← Transform(C, x_curr)
  x_temp ← Net(x_from, x_goal, ĉ)
  τ_temp ← Steer(x_curr, x_temp)
  if NotEmpty(τ_temp) then
    τ ← Add(τ, τ_temp)
    τ_goal ← Steer(x_temp, x_goal)
    if NotEmpty(τ_goal) then
      τ ← Add(τ, τ_goal)
      return τ
    end if
    x_curr ← x_temp
  end if
end for
return ∅
```

B. Trajectory Tracking

Given the path produced by Alg. 1, we first fit Dubins curves between all the points, which can then be sampled at equally spaced intervals. Following the recommendations set forward in [9], and similar to the implementation in [1], we implement NPMC using the following objective function for the dynamics described by Eq. 1, and add a terminal cost:

$$\begin{aligned} \min_{u(t)} \ & \tilde{s}(N)^T Q_t \tilde{s}(N) + \sum_{i=0}^{N-1} \tilde{s}(t)^T Q_s \tilde{s}(t) + u(t)^T R u(t) \\ \text{subject to} \quad & s(t+1) = s(t) + \begin{bmatrix} v_s \cos\theta \\ v_s \sin\theta \\ u \end{bmatrix} \Delta t, \\ & -u_{\min} \leq u(t) \leq u_{\max} \end{aligned} \tag{3}$$

This is a NMPC formulation with state update constraints and control input constraints, where: N is the prediction horizon, $u(t)$ is the applied steering angle at time t, Δt is discretized timestep used for the discrete kinematic state constraints. Q_s is the state penalty matrix, R is the control effort penalty and Q_t is the terminal state penalty. v_s is a constant velocity. $\tilde{s}(t) = s(t) - \hat{s}(t)$, where s is the predicted state at time t, and $\hat{s}$ is the reference state. The solver is implemented in Python using CASAdi [10] and Interior Point Optimizer (IPOpt, [11]).

V. Results

Using the methods described above, the network was able to achieve promising results in generalized, previously unseen environments. The planner is able to run in real time, even without any GPU acceleration, on consumer hardware.

The model is able to understand the dynamics constraints, and plan convoluted paths even in the presence of obstacles. See, for instance, Fig. 5.

The planner is also able to plan well near obstacles. The trajectory tracking performance is also solid. Fig. 6 shows a

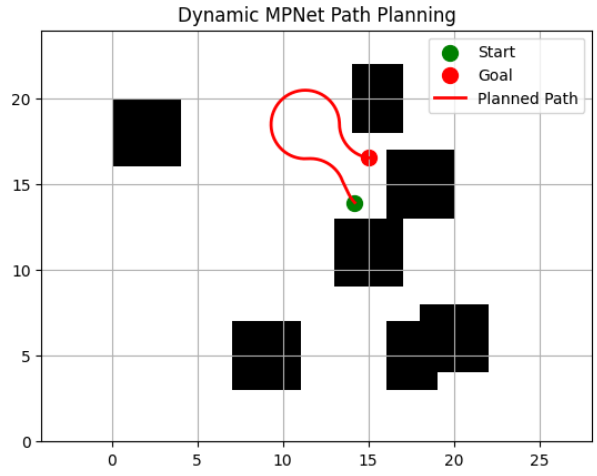

Fig. 5. A path planned using the DynamicMPNet planner.

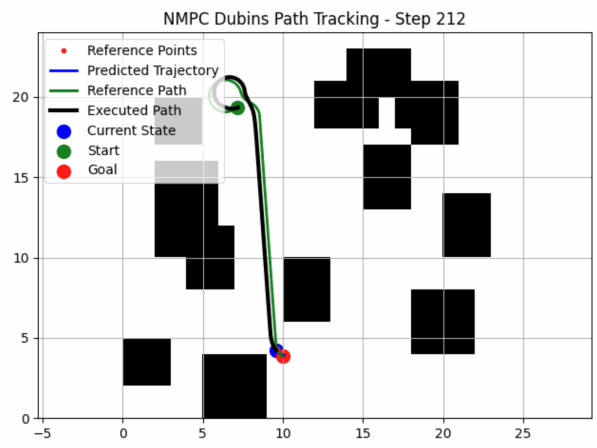

Fig. 6. The planner narrowly evades nearby obstacles in order to plan a safe route between the two points. The NMPC tracking performance is good. Planning such a path using the neural network is almost instantaneous.

simulated path planning and trajectory tracking between two distant points.

Overall, we were pleased with the results the model was able to obtain. While there is room for improvement, through further training and fine tuning model architecture, we were able to replicate the results of the original paper with good success.

VI. Conclusion

In this paper, we have implemented the approach put forward in [1] for a vehicle modeled by Dubins dynamics to generate dynamically feasible trajectories in real time. Compared to the offline RRT* trajectory planner, this approach achieves significantly faster speeds while still successfully generating trajectories. There are, however, limitations to this approach. Due to the neural nature of this approach, the generated trajectories are not guaranteed to be dynamically feasible, and thus it must be used in conjunction with a fallback planner. Additionally, to plan trajectories beyond the local region, a global, nominal planner must be used to generate intermediate goal poses.

Other papers, e.g. [12], have applied a transformer architecture for trajectory generation of non-holonomic systems. Applying a transformer architecture would be particularly interesting for this type of problem, as the waypoints of the trajectory could be predicted as a sequence with their interdependencies more accurately captured by the model.

Overall, we believe applying neural methods to trajectory generation problems will allow capturing complex system dynamics with real time performance capabilities in a way classical methods will likely fail to do.

References

[1] J. J. Johnson, L. Li, F. Liu, A. H. Qureshi, and M. C. Yip, "Dynamically Constrained Motion Planning Networks for Non-Holonomic Robots," 2020.

[2] L. E. Dubins, "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *American Journal of Mathematics*, vol. 79, no. 3, p. 497, Jul. 1957.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, Jun. 2011.

[4] A. Orthey, C. Chamzas, and L. E. Kavraki, "Sampling-Based Motion Planning: A Comparative Review," 2023.

[5] J. Wang, W. Chi, C. Li, C. Wang, and M. Q.-H. Meng, "Neural RRT*: Learning-Based Optimal Path Planning," *IEEE Transactions on Automation Science and Engineering*, vol. 17, no. 4, pp. 1748–1758, Oct. 2020.

[6] H. Jiang, H. Wang, W.-Y. Yau, and K.-W. Wan, "A Brief Survey: Deep Reinforcement Learning in Mobile Robot Navigation," in *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. Kristiansand, Norway: IEEE, Nov. 2020, pp. 592–597.

[7] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion Planning Networks," in *2019 International Conference on Robotics and Automation (ICRA)*. Montreal, QC, Canada: IEEE, May 2019, pp. 2118–2124.

[8] L. Trottier, P. Giguère, and B. Chaib-draa, "Parametric Exponential Linear Unit for Deep Convolutional Neural Networks," Jan. 2018.

[9] L. Grüne and J. Pannek, *Nonlinear Model Predictive Control*. London: Springer London, 2011, pp. 43–66.

[10] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi: A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, Mar. 2019.

[11] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, Mar. 2006.

[12] H. Wang, A. H. Tan, and G. Nejat, "NavFormer: A Transformer Architecture for Robot Target-Driven Navigation in Unknown and Dynamic Environments," *IEEE Robotics and Automation Letters*, vol. 9, no. 8, pp. 6808–6815, Aug. 2024.