Semestral Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Path Planning with Diffusion Models

Leoš Drahotský

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
January 2025

Contents

Chapter 1

Introduction

Path planning (also known as motion planning) is a problem that involves finding a safe (meaning collision-free) route for a robot in an environment with obstacles. As one can imagine, it is a common problem in robotics, but planning algorithms are also used in other fields, such as computational biology, architectural design, or video games [1].

This semestral project is concerned with motion planning using diffusion models, an approach only recently introduced to the field. The aim of the project was to implement programs for training diffusion models and generating paths using these models. Furthermore, we propose how this technique can be combined with more traditional methods like Rapidly Exploring Random Trees to achieve collision-free paths.

The first two chapters provide the theoretical background behind path planning and diffusion models. In the third chapter, we describe our implementation and evaluate the performance on two simple test cases. Finally, we summarize our findings in the concluding chapter.

Chapter 2

Path Planning

The goal of the path planning problem is to find a feasible path from some starting state to a desired goal state. For example, consider a living room filled with furniture and a robotic vacuum cleaner tasked with moving from one side of the room to the other. Suppose the robot has access to a complete map of the room, including the locations of the various pieces of furniture. It can then utilize a path planning algorithm to obtain a sequence of steps that it must take to navigate safely through the space.

2.1 Configuration Space

The planning problem is best defined by the so-called "configuration space" abstraction.

For a robot that can move with n degrees of freedom, a configuration is usually an n-dimensional vector that fully describes the robot's position in the environment. The configuration space C (or C-space) is then defined as a set of all possible configurations [1].

In the previous example with the vacuum robot, the C-space is three-dimensional and consists of two space coordinates, x and y, which define the location of the robot in the living room (from a bird's eye view) and an angle that describes the rotation of the vacuum cleaner. In general, a three-dimensional rigid body moving in three-dimensional space has a six-dimensional configuration space [1]. Three dimensions account for the x, y, and z coordinates in the three-dimensional space, and the other three come from rotation, sometimes referred to as yaw, pitch, and roll.

From the configuration space C, we derive the obstacle region, $C_{obs} \subseteq C$, a set of all configurations in which the robot collides with obstacles in the environment. A traditional planning algorithm then searches for a feasible path in the free space, $C_{free} = C \setminus C_{obs}$.

This project will mainly consider a simplified model with a point robot (a body with zero volume, occupying only a single point in the environment) moving in the XY plane. This yields a two-dimensional configuration space on its own, but we will also work in a three-dimensional C-space in the car-like model, which we will describe shortly. Nevertheless, the described algorithms should be easily extendable to even more dimensional spaces.

2.2 Sampling-based Path Planning

The exact construction of the obstacle region C_{obs} is a challenging task in many planning problems, because of this, many algorithms avoid it. A popular approach to path planning is to only randomly sample configurations from the configuration space C and connect them into a graph or a tree. This way, we are effectively discretizing the free space C_{free} into a graph structure, and we can use standard graph algorithms such as Breadth-first Search or A* to extract the path. This has proven to be effective even for large problems and high-dimensional configuration spaces [2].

2.2.1 Rapidly Exploring Random Tree

One of the representatives of these randomized strategies to planning is Rapidly Exploring Random Tree (or RRT for short), which works by incrementally growing a search tree in the configuration space.

A simple version of this algorithm is depicted in Algorithm 1. In each iteration, we select a random configuration q_{rand} from C and find the nearest vertex $q_{nearest}$ to q_{rand} in the RRT tree T. We then construct a new vertex q_{new} by moving an incremental distance Δl from $q_{nearest}$ in the direction of q_{rand}. If the edge from $q_{nearest}$ to q_{new} does not collide with obstacles (checked inside the $OBSTACLE\_FREE()$ function in Algorithm 1), we add q_{new} to the RRT tree. The search ends if the tree is close enough to the goal configuration q_{goal} (controlled by the tolerance d_{goal}) or if the maximum number of iterations is reached.

Algorithm 1: Rapidly Exploring Random Tree

Input: initial configuration q_{init}, goal configuration q_{goal}, maximum number of iterations I_{max}, incremental distance Δl, tolerance from goal d_{goal}

Output: RRT tree T

$T.add\_vertex(q_{init})$
for $i = 1$ **to** I_{max} **do**
  $q_{rand} \leftarrow RANDOM\_CONF()$
  $q_{nearest} \leftarrow NEAREST\_NEIGHBOR(q_{rand}, T)$
  $q_{new} \leftarrow NEW\_CONF(q_{nearest}, q_{rand}, \Delta l)$
  if $OBSTACLE\_FREE(q_{nearest}, q_{new})$ **then**
    $T.add\_vertex(q_{new})$
    $T.add\_edge(q_{nearest}, q_{new})$
    if $DISTANCE(q_{new}, q_{goal}) < d_{goal}$ **then**
      return T

2.2.2 RRT*

The RRT algorithm efficiently searches the C-space for a feasible path to the goal configuration. However, the paths generated by RRT are often suboptimal, with many "sharp turns". To address this issue, Sertac Karaman and Emilio Frazzoli introduced an improved variant called RRT* [3].

Algorithm 2: RRT*

Input: initial configuration q_{init}, maximum number of iterations I_{max}, dimension of the configuration space d, incremental distance Δl, multiplication factor γ[1]

Output: RRT* tree T

$T.add\_vertex(q_{init})$
for $i = 1$ **to** I_{max} **do**
  $q_{rand} \leftarrow RANDOM\_CONF()$
  $q_{nearest} \leftarrow NEAREST\_NEIGHBOR(q_{rand}, T)$
  $q_{new} \leftarrow NEW\_CONF(q_{nearest}, q_{rand}, \Delta l)$
  if $OBSTACLE\_FREE(q_{nearest}, q_{new})$ **then**
    $T.add\_vertex(q_{new})$
    $r \leftarrow MIN(\gamma(\frac{log(T.vertex\_count)}{T.vertex\_count})^{\frac{1}{d}}, \Delta l)$
    $Q_{near} \leftarrow NEAREST\_NEIGHBORS(q_{new}, T, r)$
    $q_{min} \leftarrow q_{nearest}$
    $c_{min} \leftarrow COST(q_{nearest}) + DISTANCE(q_{nearest}, q_{new})$
    foreach $q_{near} \in Q_{near}$ **do**
      if $OBSTACLE\_FREE(q_{near}, q_{new}) \wedge COST(q_{near}) +$
       $DISTANCE(q_{near}, q_{new}) < c_{min}$ **then**
        $q_{min} \leftarrow q_{near}$
        $c_{min} \leftarrow COST(q_{near}) + DISTANCE(q_{near}, q_{new})$
    $T.add\_edge(q_{min}, q_{new})$ `// Connect to minimum-cost path`
    foreach $q_{near} \in Q_{near}$ **do**
      if $OBSTACLE\_FREE(q_{new}, q_{near}) \wedge COST(q_{new}) +$
       $DISTANCE(q_{new}, q_{near}) < COST(q_{near})$ **then**
        $q_{parent} \leftarrow PARENT(q_{near})$
        $T.remove\_edge(q_{parent}, q_{near})$
        $T.add\_edge(q_{new}, q_{near})$ `// Rewire`
return T

[1]The multiplication factor γ in the RRT* algorithm can be computed as

$$\gamma > \gamma* = 2\left(1 + \frac{1}{d}\right)^{\frac{1}{d}} \left(\frac{\mu(C_{free})}{\zeta_d}\right)^{\frac{1}{d}}$$

where d is dimension of the configuration space, $\mu(C_{free})$ is Lebesgue measure of the free space, and ζ_d is the volume of the unit ball in d-dimensional space [3]. However, it is often easier to estimate γ empirically.

RRT* adds two new steps to the RRT algorithm.

When connecting a new vertex q_{new} to the tree, RRT* considers all nearby vertices (in radius r - see Algorithm 2 for more details) rather than just the nearest vertex $q_{nearest}$. It then connects q_{new} to the vertex, through which the shortest path to the starting configuration q_{init} is achieved. To facilitate this, the RRT* tree must also store information about the length of the path from each vertex to q_{init}. In the original paper, this distance is referred to as the cost of the vertex [3].

The second improvement lies in its additional rewiring step. After q_{new} is connected to the tree, the neighboring vertices are examined again and "reconnected" to the newly added node q_{new} if it yields a better cost of the vertex.

The complete algorithm is presented in Algorithm 2.

In contrast to RRT, RRT* does not terminate upon finding a feasible solution. Instead, it continues with the rewering process and thus is able to produce shorter and often smoother paths. Karaman and Franzolli have shown that the RRT* algorithm converges to the optimal solution [3].

2.3 Planning Under Differential Constraints

The RRT algorithm described in Algorithm 1 assumes that two configurations in the free C-space can be connected by a straight line. However, this assumption often does not hold in real-world scenarios.

Consider an autonomous car that attempts to park in a narrow parking spot without crashing into adjacent cars. A simple RRT algorithm would not give us a satisfactory solution since the car would have to move sideways or rotate in place to follow these "straight paths". Of course, this is impossible for most cars. To solve these kinds of problems, we need to incorporate the car's motion constraints into the planning algorithm itself.

In general, to describe the motion capabilities of the robot, we use differential equations

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \tag{2.1}$$

where $\boldsymbol{x} \in X$ is a state vector from state space X, $\dot{\boldsymbol{x}}$ denotes derivative of $\boldsymbol{x}$ with respect to time, $\boldsymbol{u} \in U$ is an action vector from an action space U and the vector function $\boldsymbol{f}$ is called the state transition function.

We have mentioned two new "spaces" in the Equation 2.1.

The state space X represents a generalization of the configuration space C. For a simple kinematic model (a model that is described using only first-order differential equations), the state space is the same as a configuration space, $X = C$. However, suppose that we use higher-order differential equations (dynamics of the robot are also considered) to describe the robot's motion. In that case, it is often beneficial to first convert the higher-order derivatives to a new set of constraints that contains only first-order derivatives. We can achieve this by adding extra dimensions to the configuration space, which represent the higher-order derivatives. This extra-dimensional C-space is

called a "phase space", and in this scenario, phase space is also the state space X [1].

The second new space, the action space U, is a set of all actions the robot can take in a certain state [1]. For a robot, the actions typically represent its control inputs (speed, acceleration, steering angle, ...).

2.3.1 RRT Under Constraints

Planning under differential constraints, which is the name of this concept in literature, is still an open problem in many aspects, mainly when it comes to optimal planning [4]. Fortunately, a simple modification of the RRT from Algorithm 1 exists that can incorporate the motion characteristics of robots.

Algorithm 3: RRT Under Differential Constraints

Input: initial state x_{init}, goal state x_{goal}, maximal number of iterations I_{max}, tolerance from goal d_{goal}, time step Δt, state transition function $f(x, u)$, discretized action space U

Output: RRT tree T

$T.add\_vertex(q_{init})$
for $i = 1$ **to** I_{max} **do**
  $x_{rand} \leftarrow RANDOM\_STATE()$
  $x_{nearest} \leftarrow NEAREST\_NEIGHBOR(x_{rand}, T)$
  $d_{min} = \infty$
  $x_{min} \leftarrow \emptyset$
  foreach $u \in U$ **do**
    $x_{new} \leftarrow x_{nearest} + \int_0^{\Delta t} f(x(t), u)\, dt$
    if $OBSTACLE\_FREE(x_{nearest}, x_{new}) \wedge$ $DISTANCE(x_{new}, x_{rand}) < d_{min}$ **then**
      $x_{min} \leftarrow x_{new}$
      $d_{min} \leftarrow DISTANCE(x_{new}, x_{rand})$
  if $x_{min} \neq \emptyset$ **then**
    $T.add\_vertex(x_{min})$
    $T.add\_edge(x_{nearest}, x_{min})$
    if $DISTANCE(x_{min}, x_{goal}) < d_{goal}$ **then**
      return T

Instead of moving an incremental distance Δl (as inside the NEW_CONF function in Algorithm 1), we can discretize the actions space U into a few selected actions, which are applied during the tree expansion. Each action is applied over a small time step Δt, and the new state is computed by integrating the state transition function

$$\boldsymbol{x}(\Delta t) = \boldsymbol{x}(0) + \int_0^{\Delta t} \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u})\, dt \tag{2.2}$$

where $\boldsymbol{x}(0)$ is the initial state at the start of the movement (corresponds to $x_{nearest}$ in Algorithm 3) and action $\boldsymbol{u}$ is constant over the time interval Δt.

In each iteration of the RRT algorithm, all actions from the discretized action set U are applied, and the action that yields a new state closest to the previously sampled random state x_{rand} is selected. This way, the random state x_{rand} effectively guides the direction of the search.

2.3.2 Car-like Model

To simulate a car's motion, we can use the so-called car-like model

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \frac{v}{L}\tan\phi \end{aligned} \tag{2.3}$$

where x and y represent coordinates of the model in XY plane, θ is the rotation of the car model, v is the speed of the car, ϕ is the steering angle of the front wheels, and L is the length of the car (more precisely distance between the front and rear axles) [5].

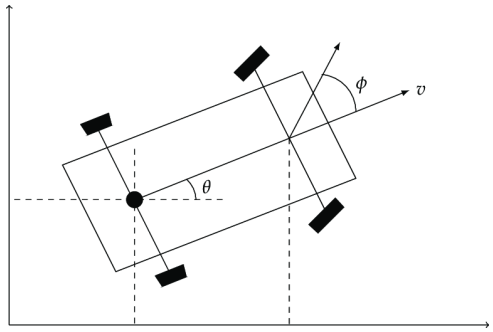


Figure 2.1: Car-like model. (Figure taken from [5]. Original authors: Ana Gavina, Jose Matos and Paulo Vasconcelos. Licensed under Creative Commons.)

In this simplified model, a car has three degrees of freedom, x, y, and θ, and thus the configuration space is three-dimensional. Since we do not consider the car's dynamics, the configuration space is also the state space. The action space has two components: the speed of the car v and the steering angle ϕ.

To simplify the path planning algorithm further, we can assume that the car is moving with a constant speed $v = 1$ and a constant steering angle ϕ is applied over a time step Δt. In this scenario, a simple analytical solution to the Equations 2.3 exists

$$\begin{aligned} x(\Delta t) &= x(0) + \frac{L}{\tan\phi}\Big(\sin\big(\theta(\Delta t)\big) - \sin\big(\theta(0)\big)\Big) \\ y(\Delta t) &= y(0) + \frac{L}{\tan\phi}\Big(\cos\big(\theta(0)\big) - \cos\big(\theta(\Delta t)\big)\Big) \\ \theta(\Delta t) &= \theta(0) + \frac{\Delta t}{L}\tan(\phi) \end{aligned} \tag{2.4}$$

This result can be used together with Algorithm 3 to generate collision-free paths using the car-like model.

We will use both the "car-like RRT" and RRT* algorithms to generate datasets for our diffusion models.

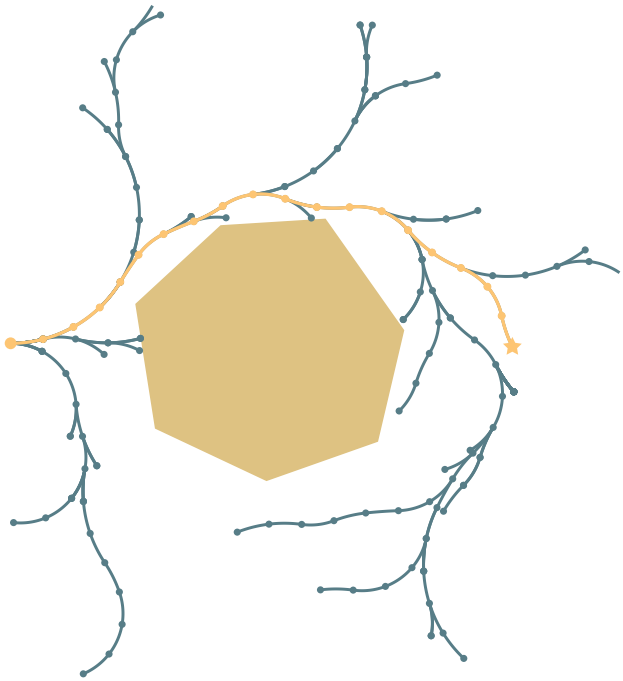


Figure 2.2: Example of a search tree (blue) generated by Algorithm 3 with a point robot and car-like differential constraints. There is one brown obstacle in the middle and the result path is orange.

Chapter 3

Diffusion Models

Diffusion models, first introduced in 2015 [6], are a class of deep learning generative models that have demonstrated state-of-the-art performance in image generation, as seen in models like DALL-E or Stable Diffusion. In a nutshell, diffusion models (or denoising diffusion probabilistic models) learn to generate new images by first iteratively adding a small amount of noise to the training images until the images resemble only pure noise. The goal of the diffusion model is to learn the opposite, the reverse diffusion process, which takes a noisy image as an input and progressively denoises it. By applying this reverse process to random noise, the model can generate entirely new images not seen during training.

However, diffusion models are not limited to image generation. Through the forward diffusion process, they can learn complex distributions and then, by applying the reverse diffusion process to random noise, sample from these distributions.

3.1 Forward Diffusion Process

Let $\boldsymbol{x}_0 \sim q(\boldsymbol{x_0})$ be a training sample from some distribution $q(\boldsymbol{x_0})$, which we want to "learn" using the diffusion model. The forward diffusion process is a Markov chain $q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)$, which progressively adds noise to the sample $\boldsymbol{x}_0$ over series of timesteps t (indicated by the subscript) as follows:

$$q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) \tag{3.1}$$

$$q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I}) \tag{3.2}$$

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a normal distribution with mean $\boldsymbol{\mu}$ and covarience $\boldsymbol{\Sigma}$ and $\beta_t \in (0, 1)$ is variance at time t. The end goal of the forward process is to transform the sample $\boldsymbol{x}_0$ into a distribution that approximates the standard Gaussian $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ at the final timestep T. The rate at which the data distribution transitions into Gaussian noise is controlled by the so-called variance schedule $\beta_1, ..., \beta_T$ [7].

By introducing $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, we can express $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)$ in

closed form

$$q(\boldsymbol{x}_t \mid \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}) \tag{3.3}$$

This means that we can sample $\boldsymbol{x}_t$ at arbitrary timestep t directly, without needing to perform t subsequent diffusion steps, as

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon} \tag{3.4}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ [7].

3.2 Reverse Denoising Process

To generate new samples from the data distributions, we need to construct the reverse process. This process starts with pure noise $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and gradually denoises it over T steps to obtain $\boldsymbol{x}_0$, which should resemble the data distribution $q(\boldsymbol{x_0})$. This is most commonly achieved by training a neural network $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ parametrizied by θ to predict the noise $\boldsymbol{\epsilon}$ added to the sample $\boldsymbol{x}_0$ during the forward process (as in Equation 3.4) [7, 8]. It can be shown, that the exact reverse process $q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$ (which is intractable as it depends on the entire data distribution [8]) can be then approximated as follows:

$$\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sqrt{1-\alpha_t}\boldsymbol{z} \tag{3.5}$$

where $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.

The architecture of the neural network $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ usually follows the structure of the U-Net [9], often with added residual and attention layers. The diffusion timestep (the second input to the network) is first embedded into a higher-dimensional space (for example using sinusoidal position embedding [10]) and then typically added to the residual blocks [7]. We can train the model using the standard mean squared error (MSE) loss function. The complete training procedure is outlined in Algorithm 4, while the sampling process is described in Algorithm 5.

Algorithm 4: Diffusion Model Training

Input: diffusion model $\boldsymbol{\epsilon}_\theta$, training set $\mathcal{T}$, diffusion length T, variance schedule $\bar{\alpha}_1, ..., \bar{\alpha}_T$
Output: trained diffusion model $\boldsymbol{\epsilon}_\theta$

repeat
  $\boldsymbol{x}_0 \sim \mathcal{T}$ `// Sample from training set`
  $t \sim UNIFORM(\{1, ..., T\})$ `// Sample random diffusion step`
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
  $\boldsymbol{x}_t \leftarrow \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$ `// Add random noise`
  Take a gradient descent step on $\nabla_\theta \left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right\|^2$
until *convergence*
return $\boldsymbol{\epsilon}_\theta$

Algorithm 5: Sampling From Diffusion Model

Input: diffusion model $\boldsymbol{\epsilon}_\theta$, diffusion length T, variance schedule $\alpha_1, ..., \alpha_T$ and $\bar{\alpha}_1, ..., \bar{\alpha}_T$

Output: sample $\boldsymbol{x}_0$ from the learned data distribution

$\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ `// Sample input noise`

for $t = T$ **to** 1 **do**

  if $t > 1$ **then**

    $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$

  else

    $\boldsymbol{z} \leftarrow \mathbf{0}$

  $\boldsymbol{x}_{t-1} \leftarrow \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t) \right) + \sqrt{1-\alpha_t}\boldsymbol{z}$ `// Denoise`

return $\boldsymbol{x}_0$

3.3 Planning with Diffusion Models

The idea behind path planning using diffusion models is to train the model on a dataset of paths from a specific environment and then use it to generate new paths. Furthermore, to generate paths from a given starting configuration to a desired goal state, we can utilize the so-called inpainting process.

Inpainting, initially introduced in the context of image generation, focuses on reconstructing missing parts of an image. This is achieved by fixing certain regions of the image and allowing the diffusion model to generate the missing sections. For path planning, the concept is adapted by fixing the start and goal configurations of the path, while the diffusion model "inpaints" the intermediate configurations to produce a complete path.

Another significant advantage of diffusion models lies in their incremental sampling process. At each denoising step, the process can be further guided toward the desired output by applying a differentiable value function [8]. In this work, we employ a technique known as steering, as proposed by Petr Zahradník [11].

In general, a steering function is a differentiable function $d(\boldsymbol{x})$ that evaluates some desired quality of the generated samples. During the sampling process, gradient descent with step size γ can be applied after each denoising step

$$\hat{\boldsymbol{x}}_t = \boldsymbol{x}_t - \gamma \frac{\partial d}{\partial \boldsymbol{x}}(\boldsymbol{x}_t) \tag{3.6}$$

to guide the diffusion process toward a desired output.

In this work, we use two types of steering functions, both introduced by Zahradník [11]. The first, referred to as length steering, measures the pairwise squared distances between neighboring configurations in a path

$$d_{length}(\boldsymbol{x}) = \sum_{i=1}^{N-1} \|\boldsymbol{x}[i] - \boldsymbol{x}[i+1]\|^2 \tag{3.7}$$

where $\boldsymbol{x}[i]$ denotes the i-th configuration in the path $\boldsymbol{x}$ and N is the length of the path. Applying this steering function results in smoother and more equidistant paths (see Figure 4.3).

The second steering function is called obstacle steering and its aim is to generate collision-free paths. This function first finds the nearest collision-free configuration $\boldsymbol{p}$ for each configuration $\boldsymbol{c}$ colliding with obstacles and computes the difference $\Delta(\boldsymbol{c})$ as

$$\Delta(\boldsymbol{c}) = \begin{cases} \boldsymbol{c} - \text{argmin}_{\boldsymbol{p} \in C_{free}} \|\boldsymbol{p} - \boldsymbol{c}\| & \text{if } \boldsymbol{c} \in C_{obs} \\ \boldsymbol{0} & \text{if } \boldsymbol{c} \in C_{free} \end{cases} \tag{3.8}$$

By applying $\Delta(\boldsymbol{c})$ to every configuration in the path and summing the squared norms, we define the obstacle steering function as:

$$d_{obstacle}(\boldsymbol{x}) = \sum_{i=1}^{N} \|\Delta(\boldsymbol{x}[i])\|^2 \tag{3.9}$$

The complete sampling algorithm, including the inpainting and steering processes, is presented in Algorithm 6.

Algorithm 6: Sampling From Diffusion Model for Path Planning

Input: initial state x_{init}, goal state x_{goal}, diffusion model $\boldsymbol{\epsilon}_\theta$, diffusion length T, variance schedule $\alpha_1, ..., \alpha_T$ and $\bar{\alpha}_1, ..., \bar{\alpha}_T$

Output: path sample $\boldsymbol{x}_0$

$\boldsymbol{x}_T \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ `// Sample input noise`

for $t = T$ **to** 1 **do**

  if $t > 1$ **then**

    $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$

  else

    $\boldsymbol{z} = \boldsymbol{0}$

  $\boldsymbol{x}_{t-1} \leftarrow \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t) \right) + \sqrt{1-\alpha_t}\boldsymbol{z}$ `// Denoise`

  $\boldsymbol{x}_{t-1} \leftarrow STEER(\boldsymbol{x}_{t-1})$

  $\boldsymbol{x}_{t-1}[1] \leftarrow x_{init}$; $\boldsymbol{x}_{t-1}[N] \leftarrow x_{goal}$ `// Inpaint`

return $\boldsymbol{x}_0$

Chapter 4

Our Implementation

Our implementation of the diffusion model builds upon the work of Peter Zahradník [11]. In his master's thesis, Zahradník was also concerned with motion planning using diffusion models and developed support for a one-dimensional (1-D) U-Net architecture within Hugging Face's Diffusers library.

The Diffusers library is a collection of popular diffusion models and provides a unified framework for training these models. It is built on PyTorch, a widely used deep learning framework. Prior to Peter's contributions, proper support for U-Nets that convolve only along one dimension of the input signal (as opposed to the two-dimensional convolution used in image generation) was missing. Peter refactored problematic parts of the library and demonstrated that his architecture could be applied to path planning problems. In this work, we will use his U-Net architecture, which adheres to the standard U-Net layout, incorporates ResNet blocks, and is conditioned on the diffusion timestep. For a detailed description, refer to [11].

We trained our models for 50 epochs with a learning rate of 0.001. We used the Adam optimizer (with default PyTorch settings) and a cosine learning rate schedule with 500 warm-up steps.

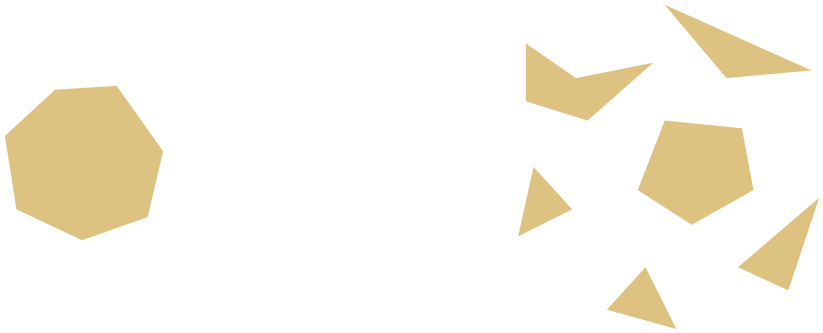
For testing purposes, we designed two environments for a two-dimensional point robot. The easy environment contains a single large obstacle in the center, while the hard environment features several smaller obstacles scattered across the 2D space - see Figure 4.1.

4.1 Cirles Test

To showcase that our diffusion model is learning properly, we first trained it on a dataset of 2000 circular paths, all centered at the same origin, oriented in the same direction, and with randomly sampled radii. The results are shown in Figure 4.2.

4.2 RRT* Dataset

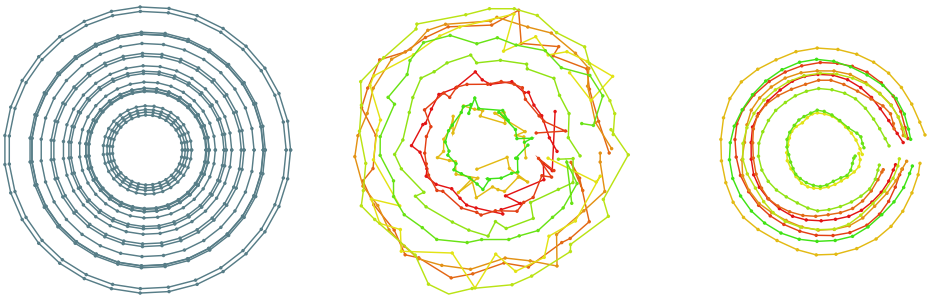
Next, we constructed datasets containing 10,000 paths for both the easy and hard environments. To generate these paths, start and goal configurations



(a) : Easy environment

(b) : Hard environment

Figure 4.1: Testing environments considered in this work.



(a) : Samples from the dataset.

(b) : Inference without any steering.

(c) : Inference with length steering.

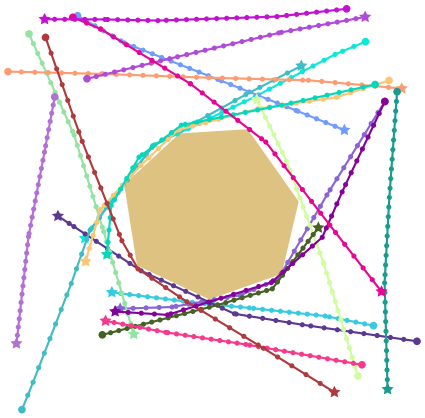
Figure 4.2: Results of the "Circles test".

were randomly sampled from opposite sides of the environment and connected using the RRT* algorithm. The resulting paths were then resampled to ensure that each contained exactly 32 configurations [1].

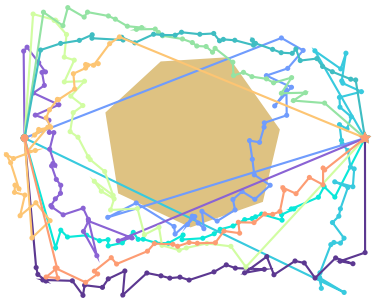
The performance of the diffusion models trained using these datasets is presented in Figures 4.3 and 4.4.

In the easy environment, the diffusion model successfully learned to avoid the obstacle in the center, and many of the generated paths were completely collision-free. In the more challenging test case, no collision-free paths were generated. However, the model was still able to identify patterns in the data, and some of the generated paths showed potential. It is also evident that employing steering functions plays a crucial role in the quality of the resulting path.

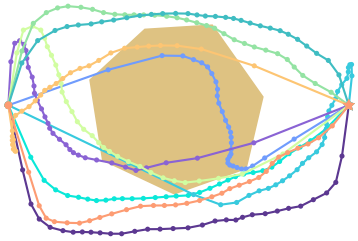
[1] All paths in the dataset must have the same length to make it possible to train the diffusion model in small batches, represented as tensors (multi-dimensional arrays).



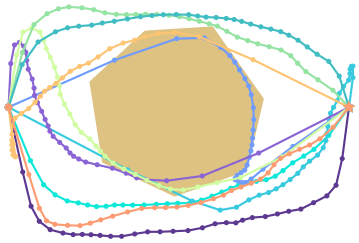
(a) : Twenty samples from the training dataset.



(b) : Inference without any steering enabled.



(c) : Inference with length steering enabled.



(d) : Inference with length and obstacle steering enabled.

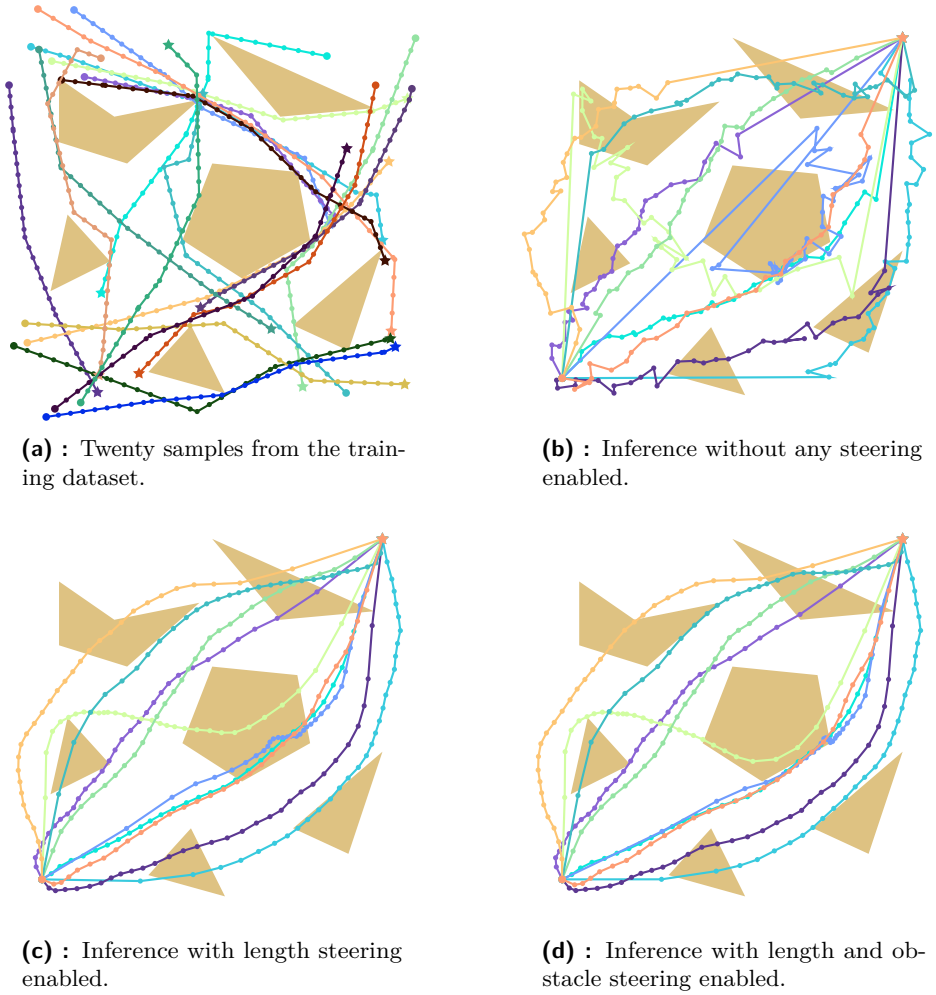
Figure 4.3: Performance of the diffusion model trained on 10,000 RRT* paths in the easy environment.

4.3 Comparison with Other Implementations

We believe that our results are similar to those achieved by Zahradník in his master's thesis [11]. While he was able to generate collision-free paths even in environments similar to our hard environment, the majority of the generated paths still collided with obstacles.

Other authors have also leveraged diffusion models for motion planning [12, 13, 14], and most of their models significantly outperform our implementation. We closely examined the work of Janner et al. [12], who were the first to introduce diffusion models to planning problems. Although many implementation details are missing in the original paper, we identified several differences compared to our approach upon closer inspection of the source code.

In their Maze2D environments (which are similar to our tests), Janner et



(a) : Twenty samples from the training dataset.

(b) : Inference without any steering enabled.

(c) : Inference with length steering enabled.

(d) : Inference with length and obstacle steering enabled.

Figure 4.4: Performance of the diffusion model trained on 10,000 RRT* paths in the hard environment.

al. have trained their models on a significantly larger dataset from the D4RL library [15]. Additionally, their training paths featured a longer planning horizon, meaning their paths were ”longer“ and contained more configurations. Perhaps the most notable difference we observed was that the waypoints (configurations) in their training paths were not limited to two dimensions (i.e., x and y coordinates) but instead were six-dimensional. Rather than providing the model with only spatial coordinates, their paths included additional information, passed into the U-Net model through six separate channels.

We hypothesized that providing the network with this additional information might enable the diffusion model to better capture the specific characteristics of the individual paths, thereby allowing it to sample higher-quality plans. To test this hypothesis, we conducted experiments using the car-like model described in Section 2.3.2.

4.4 Car-like Dataset

First, we generated 50,000 paths in the easy environment using the car-like model with fixed velocity $v = 1$ and discretized action space. We used the modified version of the RRT planner described in Algorithm 3 to accomplish this. All paths were subsequently resampled to contain exactly 32 waypoints. The state space of the car-like model is three-dimensional, and with a fixed car speed, the action space (steering angle) is one-dimensional. After combining all this information into four channels, the resulting PyTorch tensors had a shape of 4×32.

The resulting diffusion model, trained on this dataset, showed better performance compared to the dataset generated by the RRT* algorithm, even when no steering functions were applied (see Figure 4.5). We did not conduct enough tests to reliably determine whether this improvement is due to the additional information or is the result of using a larger dataset. However, it is evident that the model struggled with the inpainting process, as the start and goal states are connected to the paths in a somewhat forced manner.

Secondly, we generated a dataset of 10,000 paths in the hard environment using the same process, but with an increased planning horizon of 64 waypoints[2]. With this dataset, we were able to generate one completely collision-free path, but the failing inpainting process is even more evident - see Figure 4.6.

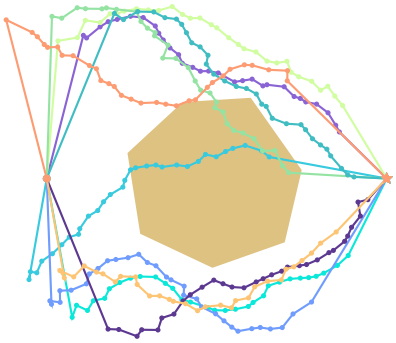
4.5 Incorporating Sampling-based Planners

Although many of the generated paths are not perfect, some still provide a reasonable "guess" for a collision-free path, as they often collide with obstacles in only a small portion of the trajectory. These small problematic segments could be corrected using methods such as Rapidly Exploring Random Tree to achieve fully collision-free paths. By focusing the RRT algorithm on only a small fraction of the environment, we could potentially achieve faster runtime, mainly in problems where multiple feasible paths need to be generated simultaneously, as the diffusion process can be easily parallelized.

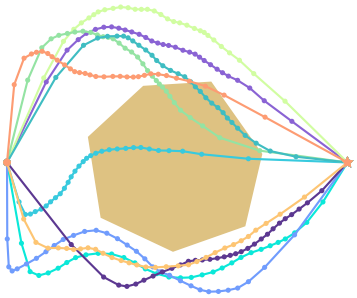
The diffusion model would serve as a form of intuition for the robot, while the RRT algorithm would act as a local planner to refine the trajectory and provide the exact path to the goal configuration. Unfortunately, due to time constraints, we could not thoroughly investigate this idea in practice, and therefore, we propose it as a direction for future research.

[2]Only 10,000 paths were generated in the hard environment (as opposed to the 50,000 in the easy environment) due to limited computational resources.

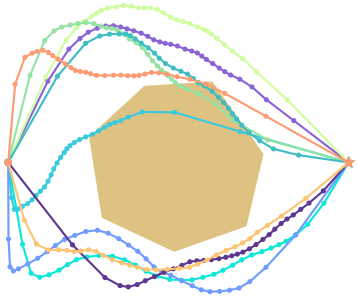
(a) : Twenty samples from the training dataset.



(b) : Inference without any steering enabled.

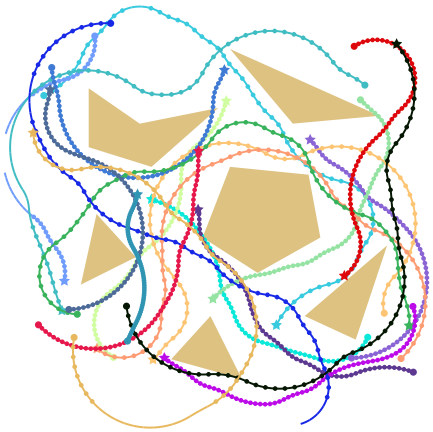


(c) : Inference with length steering enabled.

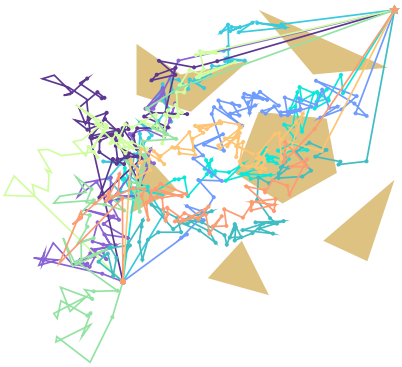


(d) : Inference with length and obstacle steering enabled.

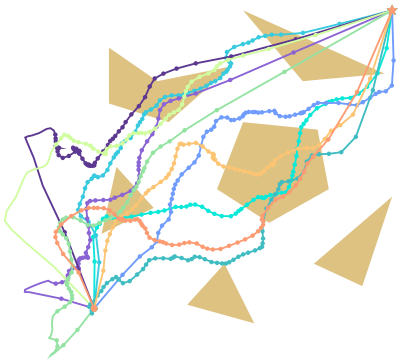
Figure 4.5: Performance of the diffusion model trained on 50,000 Car-like paths in the easy environment.



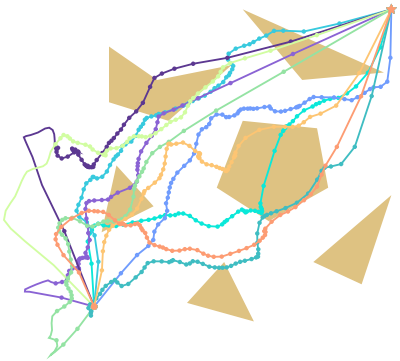
(a) : Twenty samples from the training dataset.



(b) : Inference without any steering enabled.



(c) : Inference with length steering enabled.



(d) : Inference with length and obstacle steering enabled.

Figure 4.6: Performance of the diffusion model trained on 10,000 Car-like paths in the hard environment.

Chapter 5

Conclusion

In this semestral project, we developed a framework for generating datasets, training diffusion models, and sampling paths for simple path planning problems. We demonstrated the performance of our implementation on two simple planning tests for a 2-dimensional point robot. Additionally, we investigated whether incorporating more information about the planning problem (in the form of action variables) leads to improved sample quality. Although this approach allowed us to generate more collision-free paths on average, other issues arose, particularly with the inpainting process. These shortcomings remain unclear, and further research is needed to understand them better.

Due to time constraints, we were unable to fully address the last point in the project assignment, which involved using the diffusion models in conjunction with traditional sampling-based methods to produce collision-free paths. Instead, we decided to allocate the available time to improving the quality of the samples generated by the diffusion models. Nonetheless, we believe this topic holds great potential for future research.

Appendix A

Bibliography

1. LAVALLE, Steven M. *Planning algorithms*. New York: Cambridge University Press, 2006. ISBN 978-0-521-86205-9. Available also from: `https://lavalle.pl/planning/`.
2. LAVALLE, Steven M. *Rapidly-exploring random trees: A new tool for path planning*. Iowa State University, 1998. Available also from: `https://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf`.
3. KARAMAN, Sertac; FRAZZOLI, Emilio. *Sampling-based Algorithms for Optimal Motion Planning* [online]. arXiv, 2011 [visited on 2025-01-04]. No. arXiv:1105.1186. Available from DOI: `10.48550/arXiv.1105.1186`.
4. SCHMERLING, Edward; JANSON, Lucas; PAVONE, Marco. Optimal sampling-based motion planning under differential constraints: The driftless case. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)* [online]. Seattle, WA, USA: IEEE, 2015, pp. 2368–2375 [visited on 2025-01-08]. ISBN 978-1-4799-6923-4. Available from DOI: `10.1109/ICRA.2015.7139514`.
5. GAVINA, Alexandra; MATOS, José M. A.; VASCONCELOS, Paulo B. Solving Nonholonomic Systems with the Tau Method. *Mathematical and Computational Applications* [online]. 2019, vol. 24, no. 4, p. 91 [visited on 2025-01-10]. ISSN 2297-8747. Available from DOI: `10.3390/mca24040091`.
6. SOHL-DICKSTEIN, Jascha; WEISS, Eric A.; MAHESWARANATHAN, Niru; GANGULI, Surya. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics* [online]. arXiv, 2015 [visited on 2025-01-17]. No. arXiv:1503.03585. Available from DOI: `10.48550/arXiv.1503.03585`.
7. HO, Jonathan; JAIN, Ajay; ABBEEL, Pieter. *Denoising Diffusion Probabilistic Models* [online]. 2020. [visited on 2024-09-10]. No. arXiv:2006.11239. Available from arXiv: `2006.11239[cs,stat]`.
8. NICHOL, Alex; DHARIWAL, Prafulla. *Improved Denoising Diffusion Probabilistic Models* [online]. arXiv, 2021 [visited on 2025-01-18]. No. arXiv:2102.09672. Available from DOI: `10.48550/arXiv.2102.09672`.

9. RONNEBERGER, Olaf; FISCHER, Philipp; BROX, Thomas. *U-Net: Convolutional Networks for Biomedical Image Segmentation* [online]. arXiv, 2015 [visited on 2024-11-30]. No. arXiv:1505.04597. Available from DOI: 10.48550/arXiv.1505.04597.

10. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. *Attention Is All You Need* [online]. arXiv, 2023 [visited on 2025-01-19]. No. arXiv:1706.03762. Available from DOI: 10.48550/arXiv.1706.03762.

11. ZAHRADNÍK, Petr. *Diffusion models for path planning*. 2024. Available also from: https://dspace.cvut.cz/handle/10467/114914. master thesis. Czech Technical University in Prague.

12. JANNER, Michael; DU, Yilun; TENENBAUM, Joshua B.; LEVINE, Sergey. *Planning with Diffusion for Flexible Behavior Synthesis* [online]. arXiv, 2022 [visited on 2024-09-10]. No. arXiv:2205.09991. Available from arXiv: 2205.09991[cs].

13. CARVALHO, Joao; LE, An T.; BAIERL, Mark; KOERT, Dorothea; PETERS, Jan. *Motion Planning Diffusion: Learning and Planning of Robot Motions with Diffusion Models* [online]. arXiv, 2024 [visited on 2025-01-10]. No. arXiv:2308.01557. Available from DOI: 10.48550/arXiv.2308.01557.

14. LIANG, Zhixuan; MU, Yao; DING, Mingyu; NI, Fei; TOMIZUKA, Masayoshi; LUO, Ping. *AdaptDiffuser: Diffusion Models as Adaptive Self-evolving Planners* [online]. arXiv, 2023 [visited on 2024-12-01]. No. arXiv:2302.01877. Available from DOI: 10.48550/arXiv.2302.01877.

15. FU, Justin; KUMAR, Aviral; NACHUM, Ofir; TUCKER, George; LEVINE, Sergey. *D4RL: Datasets for Deep Data-Driven Reinforcement Learning* [online]. arXiv, 2021 [visited on 2024-11-11]. No. arXiv:2004.07219. Available from arXiv: 2004.07219[cs].