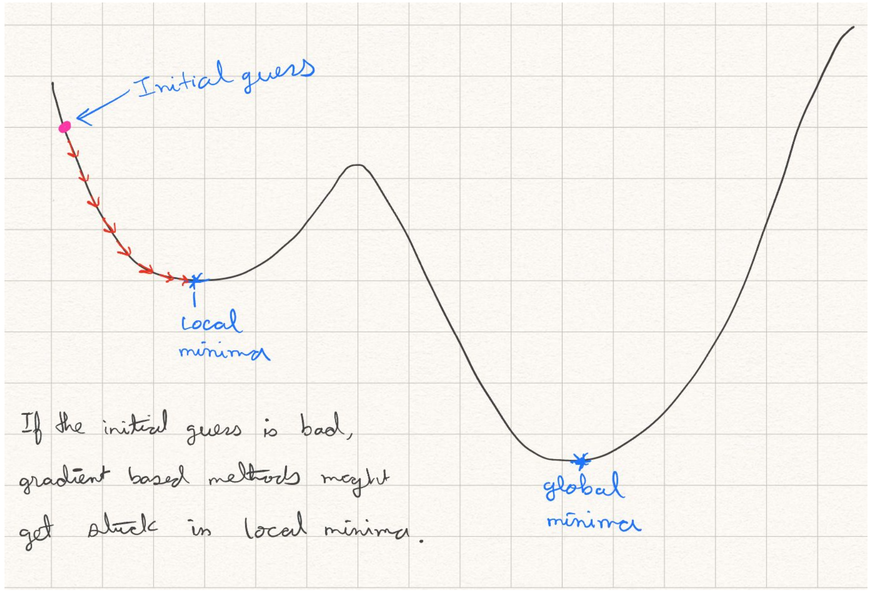
Cross Entropy Method

The motion planning problem in a mobile robot has two main constraints:

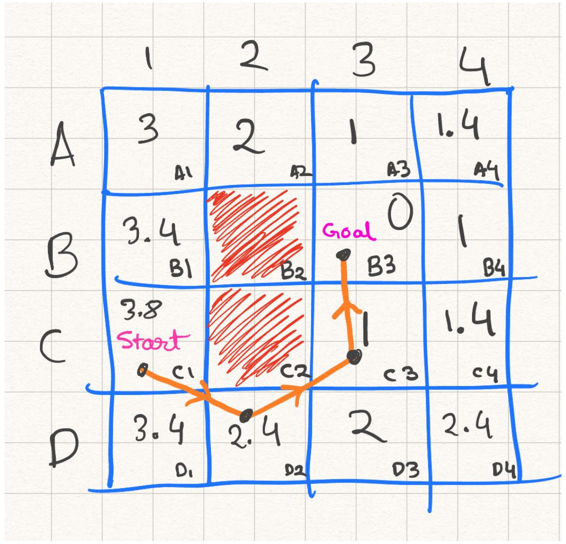
1. Kinematics/Dynamics constraints
2. Obstacle constraints

The goal is to generate a trajectory to reach a goal optimally.

- Gradient based optimization approaches like stochastic gradient descent are not suitable as they need a good starting guess and the obstacle avoidance constraints may be non-smooth. Without a good starting guess the algorithm might get stuck in a local minima.


Initial guess
local minima
global minima
If the initial guess is bad,
gradient based methods might
get stuck in local minima.

- An alternative approach is to discretize the vehicle state space and generate a path by transitioning between adjacent cells. However if the state space is very large, this approach becomes computationally expensive. Also we can't add kinematic constraints in these kinds of approaches.



- To overcome these difficulties, we use sampling based motion planning.
- Here we construct a graph with nodes corresponding to the vehicle states and the edges satisfying the dynamics and constraints.
- Two main approaches here are:
  - **Rapidly Exploring Random Trees:** Quickly explores the state space and gives a solution which may not be optimal.
  - **Probabilistic Road Maps:** Approaches the optimal trajectory but at an exceptionally slow rate.
- **How do we achieve the best of both worlds?**
- For this we use stochastic optimization based methods like **Cross Entropy Method (CEM)** or **Model Predictive Path Integral (MPPI).**

Cross Entropy Method (CEM) based motion planning algorithm

1. Sample controls from a gaussian distribution.

Let’s say we are planning 100 (N=100) trajectories for 10 timesteps into the future (H = 10) and our action space is of 2-dimension (a_dim=2) consisting of velocity and angular velocity.

```
controls = torch.randn(H, N, a_dim)
# Here, H = horizon
# N = number of samples
# a_dim = dimension of the action space
```

Cross Entropy Method (CEM) based motion planning algorithm

1. <u>Sample controls from a gaussian distribution.</u>

The sampled controls are from a normal distribution with 0 mean and 1 standard deviation.

```
controls = a_mu + a_std * controls
```

Here, a_mu is a vector of dimension [10, 1, 2] and is the desired mean.

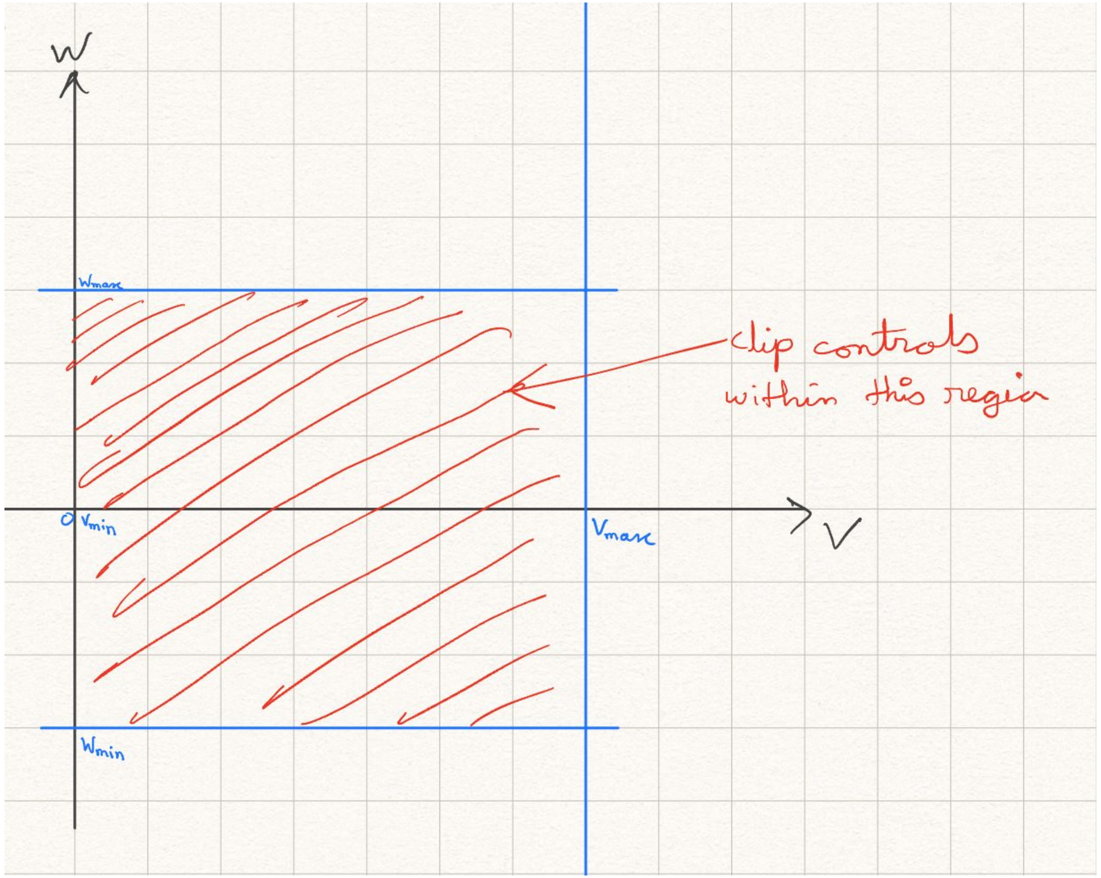
a_std is a vector of dimension [10, 1, 2] and is the desired standard deviation.

Cross Entropy Method (CEM) based motion planning algorithm

2. **<u>Clip the controls within the control bounds of the agent.</u>**

The sampled controls may have values that are beyond the maximum and minimum velocities and angular velocities of the ego-vehicle.

```
u_min = torch.tensor([v_min, w_min)

u_max = torch.tensor([v_max, w_max])

clipped_controls = torch.min(torch.max(controls,u_min),u_max)
```

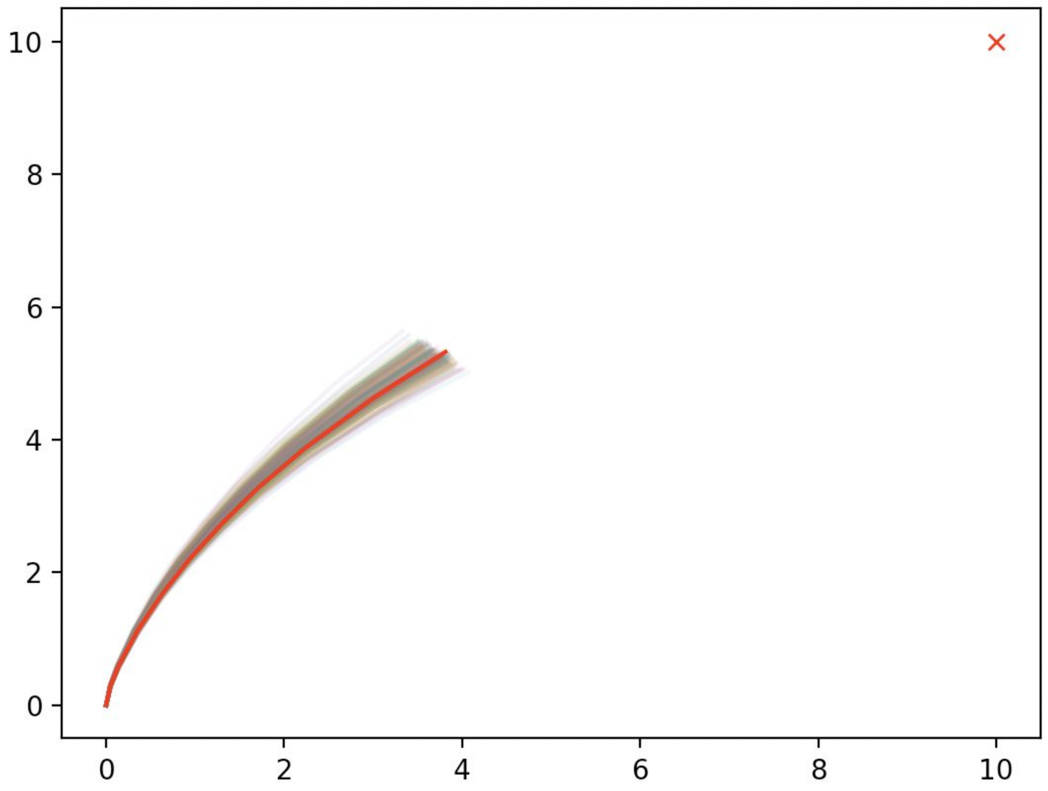

W
W_{max}
clip controls within this region
0
V_{min}
V_{max}
V
W_{min}

Rollout the trajectories

Using a simplified kinematics model of our vehicle we now pass the sampled controls through our kinematics model to get our trajectories.

- theta_t = theta_{t-1} + w*dt
- x_t = x_{t-1} + v*cos(theta_t)*dt
- y_t = y_{t-1} + v*sin(theta_t)*dt

We will have a tensor named `traj` of dimension [H,N,3] since each trajectory has x,y and theta.



- Planning horizon (H):10.
- Number of samples (N): 200
- Control dimension (a_dim): 2
- The resulting tensor is of shape [10,200,2] # [H, N, a_dim]
- After rolling out the controls through a unicycle kinematics model we get the following trajectories.

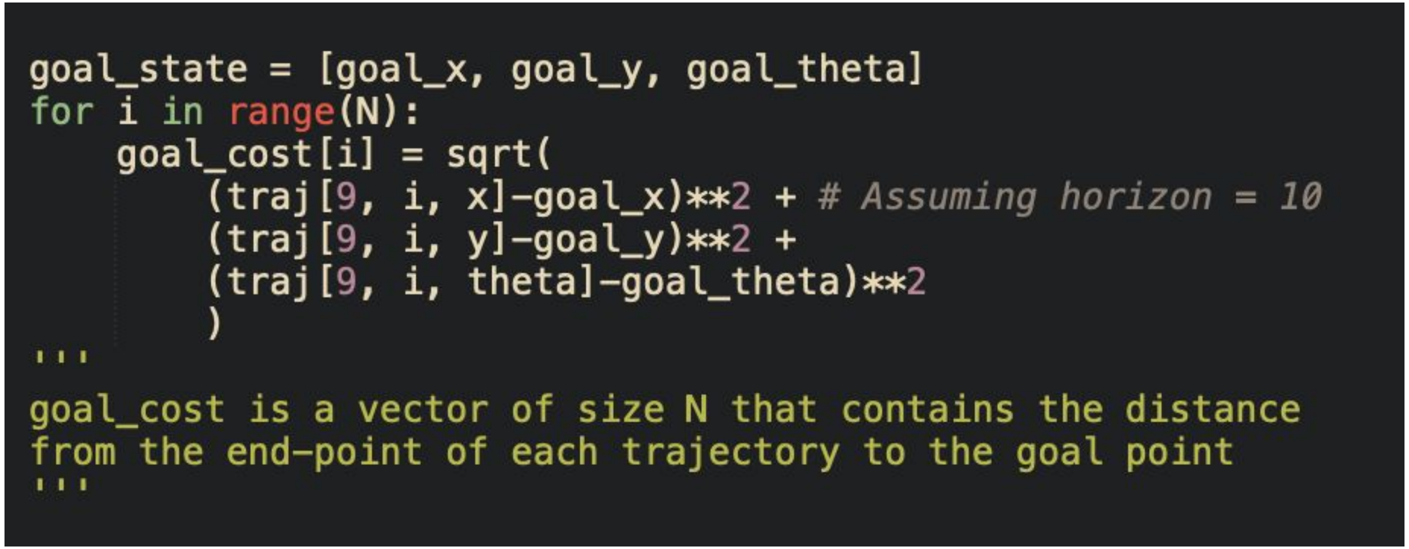
Cross Entropy Method (CEM) based motion planning algorithm

4. **Score the trajectories**

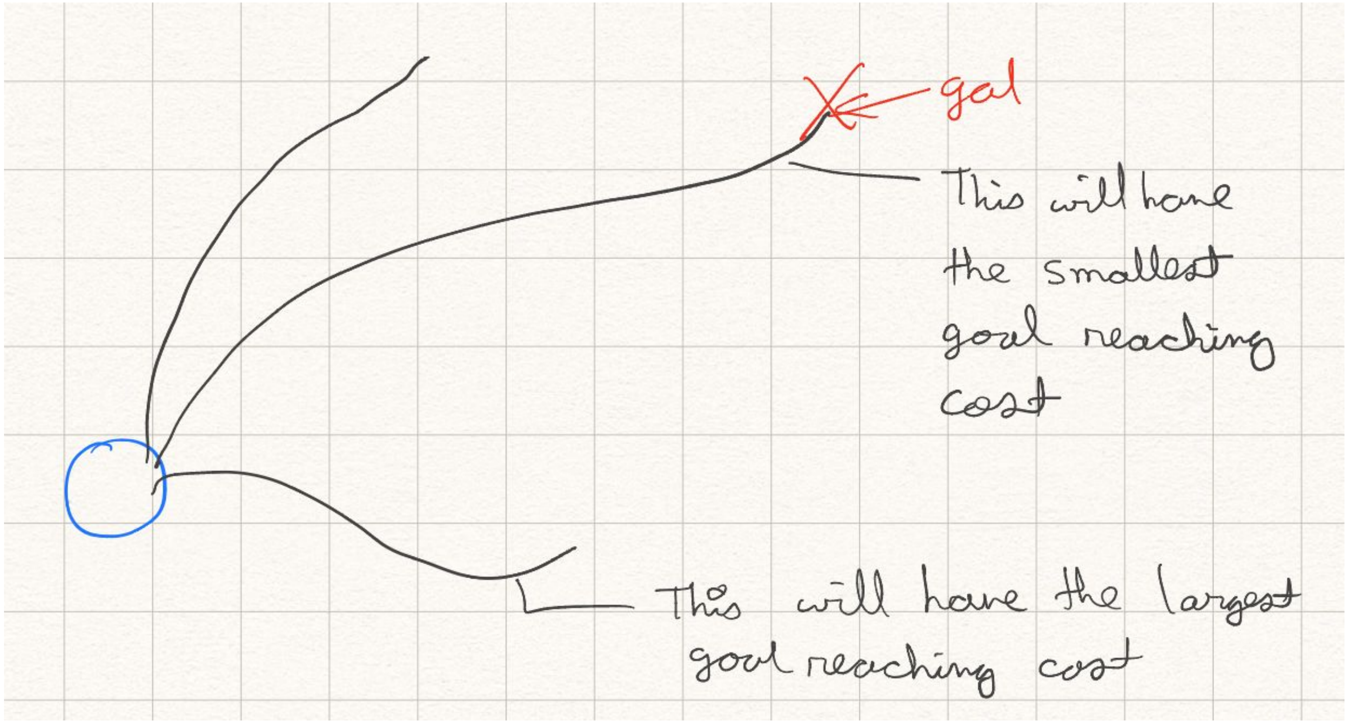
- Score our N trajectories using some cost functions.
- Examples of cost functions:
  1. Goal-reaching cost
  2. Smoothness cost
  3. Obstacle avoidance cost
- For each trajectory the total cost is: goal-reaching cost + smoothness cost + obstacle avoidance cost.
- Select the top k trajectories with the smallest cost. These are the elite trajectories.

4.1 Goal reaching cost

- The trajectory whose endpoint is closest to the goal position gets the smallest cost.

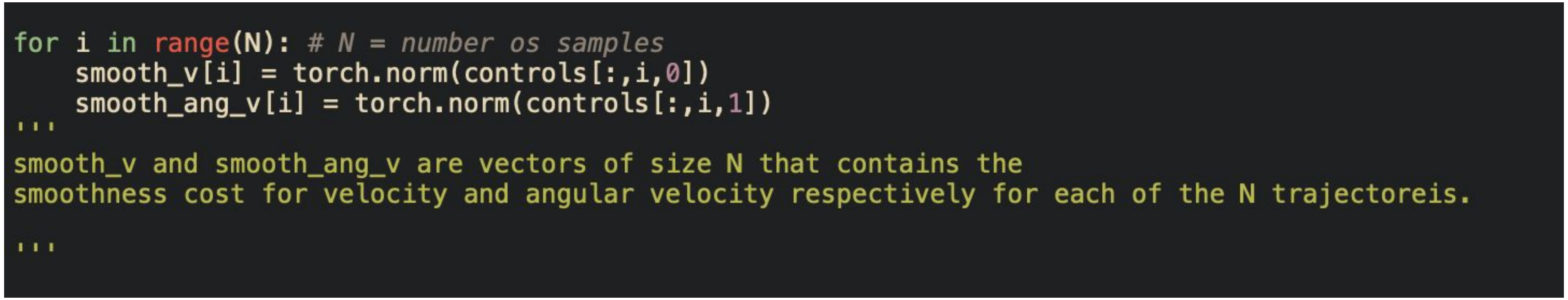


```
goal_state = [goal_x, goal_y, goal_theta]
for i in range(N):
    goal_cost[i] = sqrt(
        (traj[9, i, x]-goal_x)**2 + # Assuming horizon = 10
        (traj[9, i, y]-goal_y)**2 +
        (traj[9, i, theta]-goal_theta)**2
        )
'''
goal_cost is a vector of size N that contains the distance
from the end-point of each trajectory to the goal point
'''
```

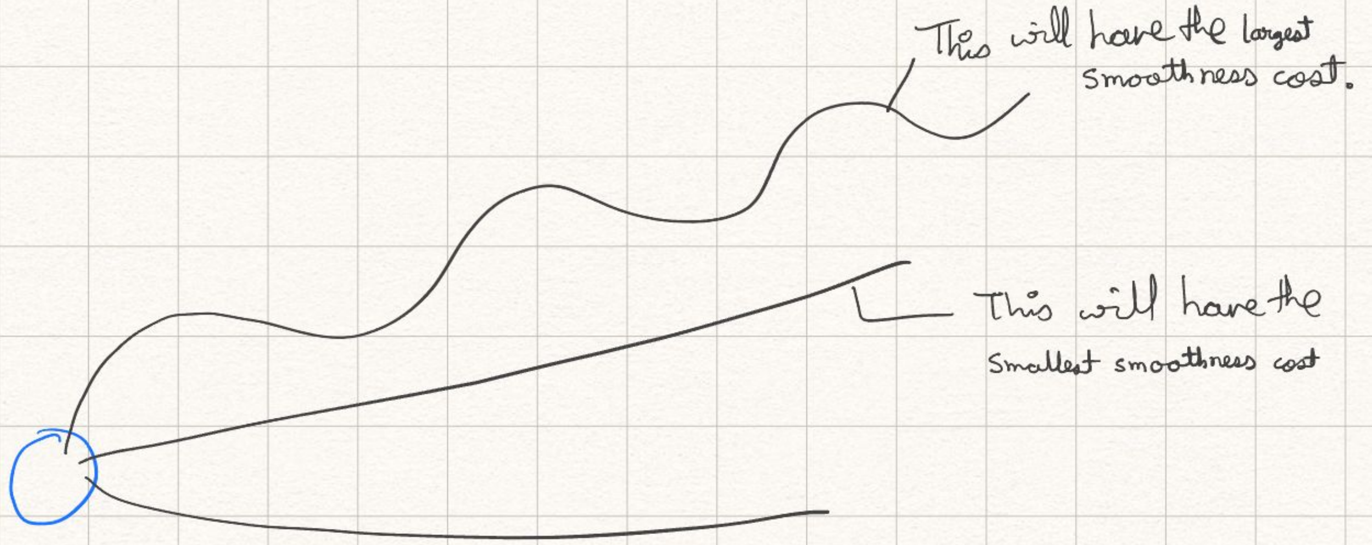

goal
This will have the smallest goal reaching cost
This will have the largest goal reaching cost

4.2 Smoothness cost

- The trajectory where the change in the angular velocity and velocity is minimum is considered smoothest.

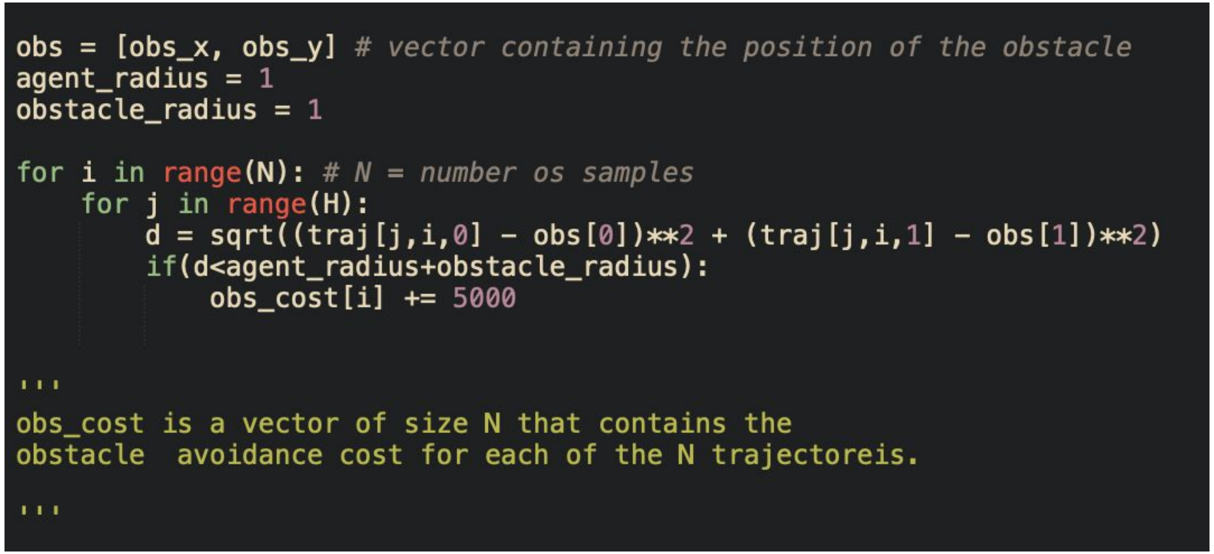


```
for i in range(N): # N = number os samples
    smooth_v[i] = torch.norm(controls[:,i,0])
    smooth_ang_v[i] = torch.norm(controls[:,i,1])
'''
smooth_v and smooth_ang_v are vectors of size N that contains the
smoothness cost for velocity and angular velocity respectively for each of the N trajectoreis.

'''
```

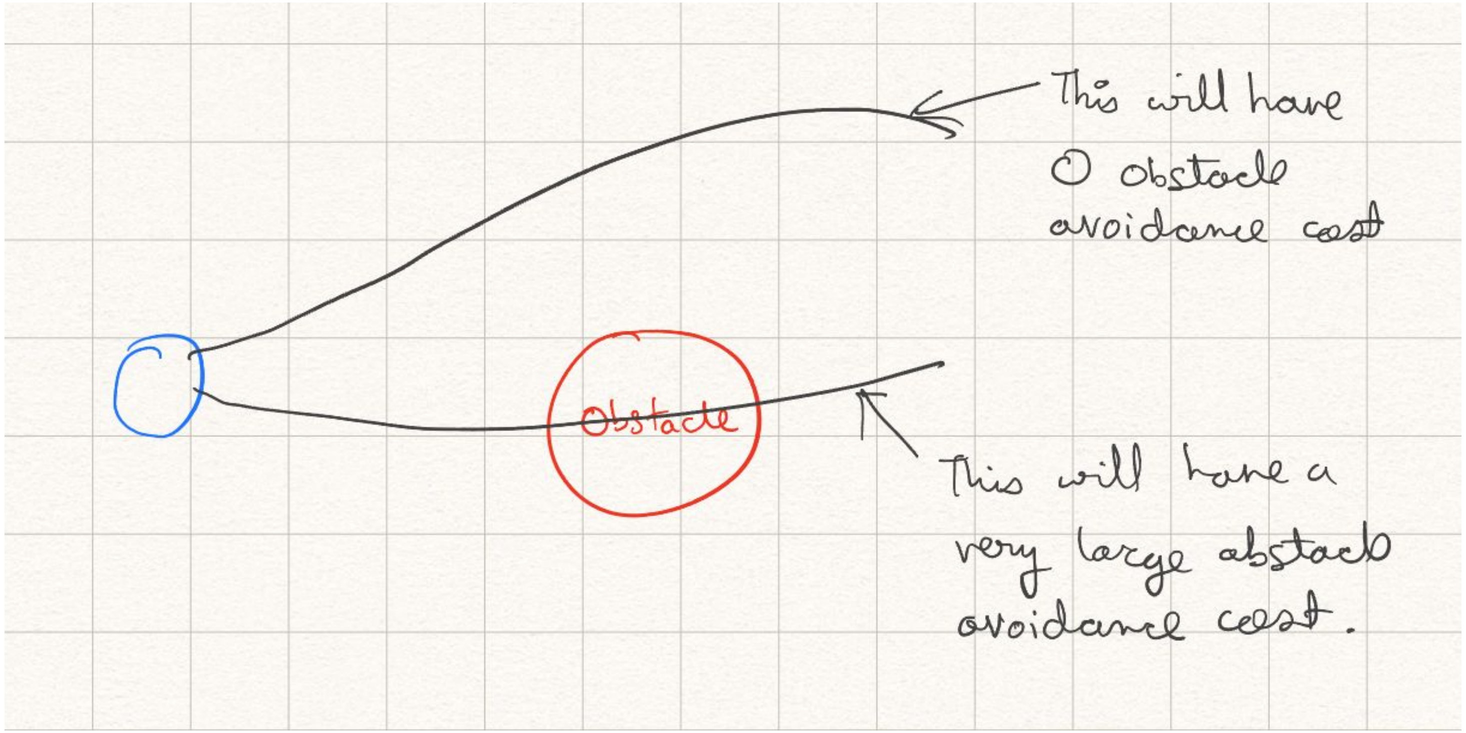

This will have the largest smoothness cost.
This will have the smallest smoothness cost

4.3 Obstacle avoidance cost

- If the x,y of the trajectory lies inside obstacle, assign a very high cost to the trajectory.

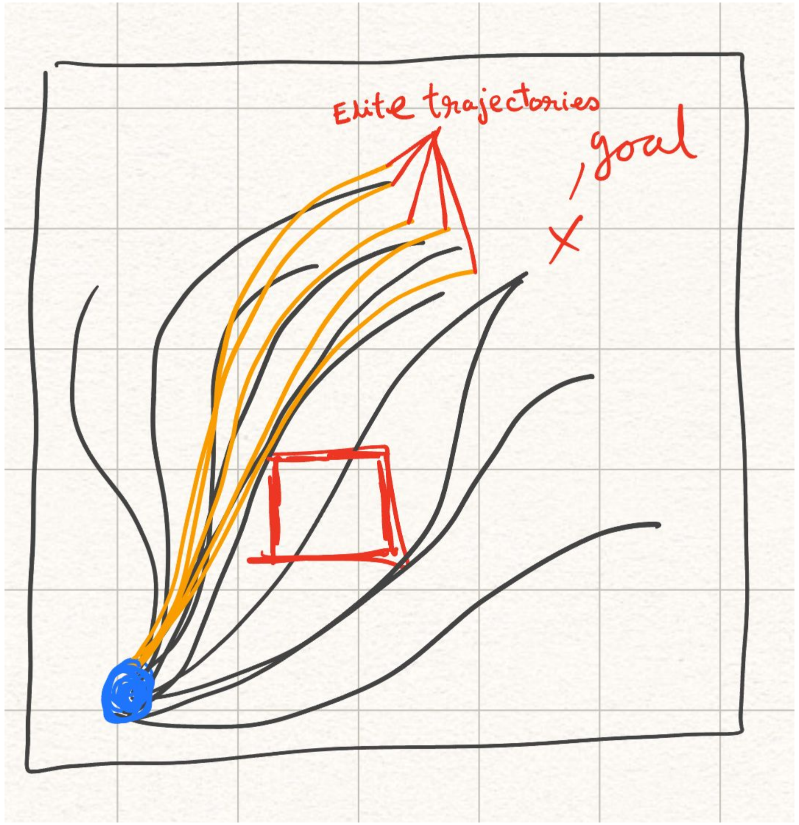


```
obs = [obs_x, obs_y] # vector containing the position of the obstacle
agent_radius = 1
obstacle_radius = 1

for i in range(N): # N = number os samples
    for j in range(H):
        d = sqrt((traj[j,i,0] - obs[0])**2 + (traj[j,i,1] - obs[1])**2)
        if(d<agent_radius+obstacle_radius):
            obs_cost[i] += 5000


'''
obs_cost is a vector of size N that contains the
obstacle  avoidance cost for each of the N trajectoreis.

'''
```


This will have
0 obstacle
avoidance cost
Obstacle
This will have a
very large obstacle
avoidance cost.

5. Select the elite trajectories

- For each trajectory the total cost is: goal-reaching cost + smoothness cost + obstacle avoidance cost.
- Select the top k trajectories with the smallest cost.
    - k<<N. A good rule of thumb is to assign k as 10% of N.
    - This means that out of N trajectories we are only considering the top k trajectories which has the best cost.
    - These k trajectories are called elite trajectories.


Elite trajectories
goal

6. **<u>Set the mean and covariance of the gaussian distribution as the mean and covariance of the elite trajectories</u>**
    a. Find the mean and the std of the elite trajectories using torch.mean and torch.std.
    b. Set the mean and std of the sampling distribution as the mean and std of the elite trajectories.
7. **<u>Goto step 1</u>**
    a. Repeat the above steps until the K-L divergence between the previous distribution and the current distribution is less than a small constant or if the covariances have nearly shrunk to 0.

Initialization of the parameters

In step 1 of the CEM algorithm we sample from a gaussian distribution.

At the very start of the algorithm, we need to have a good initialization for the this gaussian distribution.

For this:

1. We compute the optimal trajectory that reaches the goal ignoring the obstacles and kinematics/dynamics constraints.
2. The initial parameters for the gaussian distribution is centered around this optimal trajectory.

Initialization of the parameters

1. We can use RRT to find the optimal trajectory to the goal.
2. Use the controls for this trajectory to obtain the mean and the covariance.

We can also generate trajectories using bernstein polynomials. The second derivative of the bernstein trajectories gives us the velocities which was can use as the initial guess.

`Note: This step needs to be done only once.`

Once we get a trajectory using the CEM method, we can use that as a guess value for next call to the CEM function for faster computation.

One issue with this approach is that if the environment has very narrow passages, the feasible region in the trajectory space will shrink. As a result a large number of the samples will be rejected and as a result the resultant trajectory may not be close to the optimal trajectory.

Thank You