ME 5824: Homework-4

Shaunak Mehta, Ananth Jonavittula

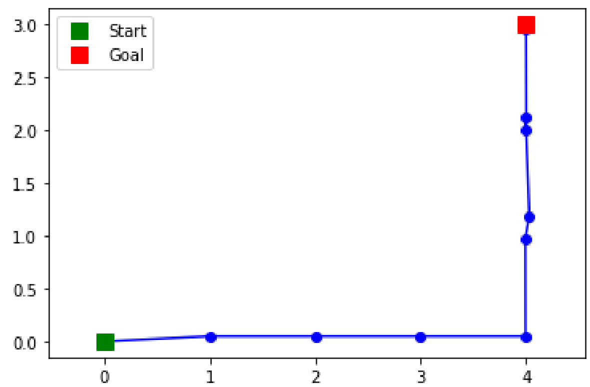
Q 1

The cost function used for the task is:

$$cost = \sum_{s} \left[||g - s|| + s_y - (s_y | s_y < 0.05) \right]$$

Here $\sum_s$ sums over the entire trajectory, g is the goal, s is the current state in the trajectory, s_y is the height of the quadcopter, and $(s_y | s_y < 0.05)$ penalizes the quadcopter if it is too close to the ground.

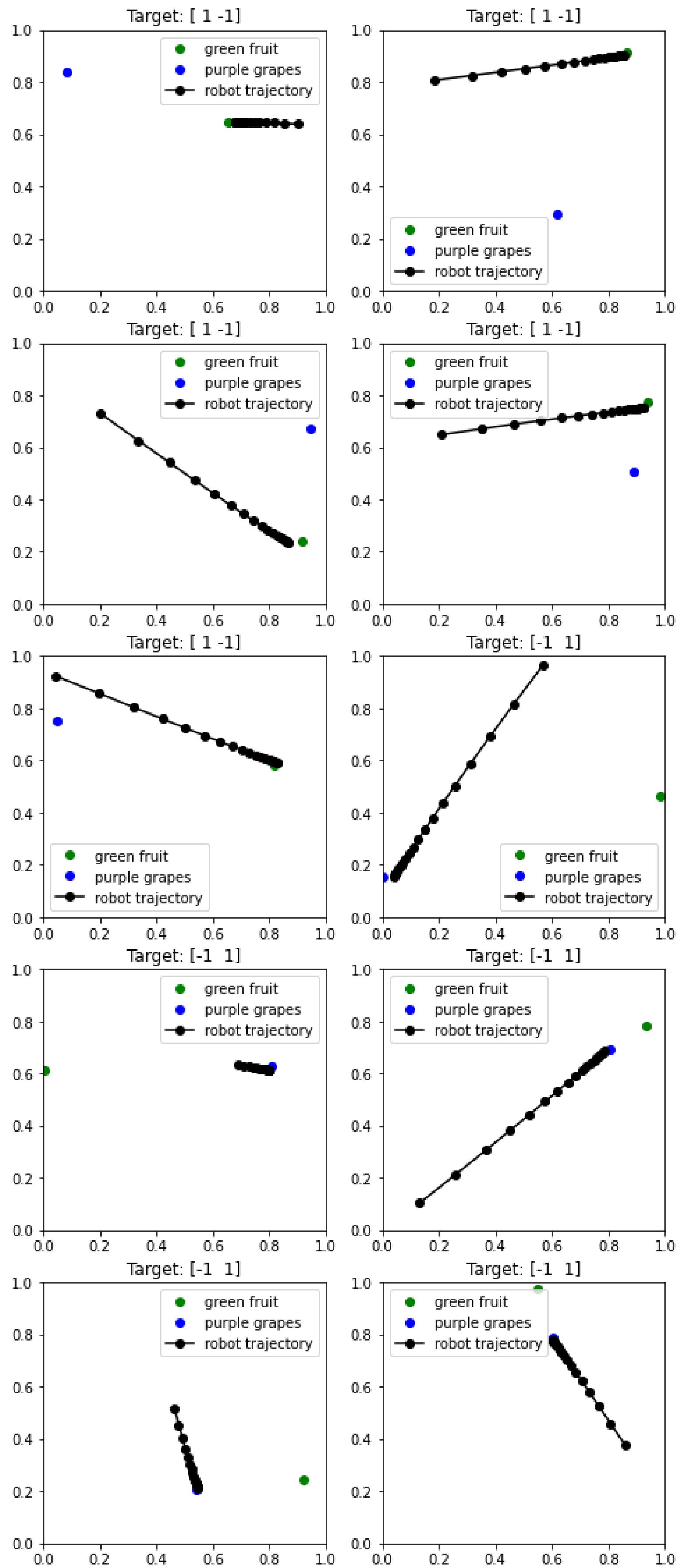
The optimized trajectory obtained at the end of simulation is shown below:



The python script for the same is attached at the end of the file.

Q 2

Plots can be found on the next page.


Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
0.0
0.2
0.4
0.6
0.8
1.0

The python script for the same is attached at the end of the file.

Q 3

$$\begin{aligned}
Regret_i &= R(\zeta^*, \theta^*) - R(\zeta^i, \theta^*) \\
Regret_{i+1} &= R(\zeta^*, \theta^*) - R(\zeta^{i+1}, \theta^*) \\
Regret_{i+1} - Regret_i &= R(\zeta^i, \theta^*) - R(\zeta^{i+1}, \theta^*) \\
&= \theta^* F(\zeta^i) - \theta^* F(\zeta^{i+1}) \\
&= \theta^* [F(\zeta^i) - F(\zeta^{i+1})]
\end{aligned}$$

From the hint (assuming that $F(\zeta)\, and\, \theta$ are unit vectors), we know that $F(\zeta^i) = \theta^i$

$$\begin{aligned}
Regret_{i+1} - Regret_i &= \theta^* [\theta^i - \theta^{i+1}] \\
&= \theta^* [\theta^i - (\theta^i + \alpha(F(\zeta^*) - F(\zeta^i))] \\
&= \theta^* [\alpha(F(\zeta^i) - F(\zeta^*))]
\end{aligned}$$

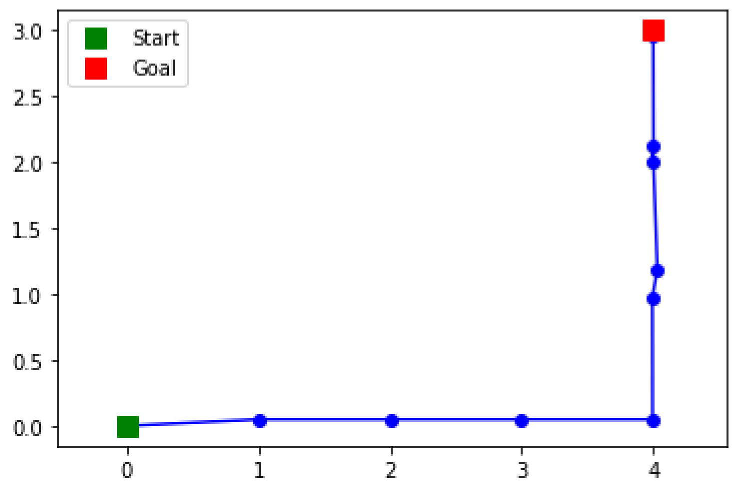
We know that $\alpha, \theta^* \in [0, \infty)$ and that $R(\zeta^*, \theta^*) > R(\zeta, \theta^*)$
Therefore, $F(\zeta^*) > F(\zeta^i)$
Therefore, $F(\zeta^i) - F(\zeta^*) < 0$
Therefore, $\theta^* [F(\zeta^i) - F(\zeta^*) < 0]$
Therefore,

$$\begin{aligned}
Regret_{i+1} - Regret_i &= \theta^* [\alpha(F(\zeta^i) - F(\zeta^*))] \\
Regret_{i+1} - Regret_i &<= 0 \\
Regret_{i+1} &<= Regret_i
\end{aligned}$$

```
### Link to Github Repo with the same code: ###
### https://github.com/dylan-losey/me5824.git ###

import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize, LinearConstraint, NonlinearConstraint


### Create a class to perform the trajectory optimization ###

class TrajOpt(object):

    def __init__(self):

        # initialize trajectory
        self.n_waypoints = 10
        self.n_dof = 2
        self.home = np.array([0., 0.])
        self.xi0 = np.zeros((self.n_waypoints, self.n_dof))
        self.xi0 = self.xi0.reshape(-1)

        # create start constraint and action constraint
        self.B = np.zeros((self.n_dof, self.n_dof * self.n_waypoints))
        for idx in range(self.n_dof):
            self.B[idx,idx] = 1
        self.lincon = LinearConstraint(self.B, self.home, self.home)
        self.nonlincon = NonlinearConstraint(self.nl_function, -1.0, 1.0)

    # each action cannot move more than 1 unit
    def nl_function(self, xi):
        xi = xi.reshape(self.n_waypoints, self.n_dof)
        actions = xi[1:, :] - xi[:-1, :]
        return np.linalg.norm(actions, axis=1)

    # trajectory cost function
    def trajcost(self, xi):
        xi = xi.reshape(self.n_waypoints, self.n_dof)
        cost = 0
        ### define your cost function here ###
        ### here is an example encouraging the robot to reach [5, 2] ###
        for idx in range(self.n_waypoints):
            cost += np.linalg.norm(np.array([4., 3.]) - xi[idx, :])
            cost += 1*abs(xi[idx, 1])
            if xi[idx, 1] < 0.05:
              cost -= 1*xi[idx, 1]
        return cost

    # run the optimizer
    def optimize(self):
        res = minimize(self.trajcost, self.xi0, method='SLSQP', constraints={self.lincon, self
        xi = res.x.reshape(self.n_waypoints, self.n_dof)
        return xi, res


### Run the trajectory optimizer ###
```

```
trajopt = TrajOpt()
xi, res = trajopt.optimize()
print(xi)
plt.plot(xi[:,0], xi[:,1], 'bo-')
plt.plot(0, 0,  'gs', markersize=10, label='Start')
plt.plot(4, 3,  'rs', markersize=10, label='Goal')
plt.legend()
plt.axis("equal")
plt.show()
```

```
[[0.         0.        ]
 [0.99879898 0.0489958 ]
 [1.99879891 0.04901323]
 [2.99879884 0.04899491]
 [3.99387065 0.04898458]
 [3.99188275 0.97197579]
 [4.03034504 1.18648881]
 [3.99522503 2.13092249]
 [4.00148332 1.99750678]
 [3.99855789 2.96269541]]
```



behavior_cloning

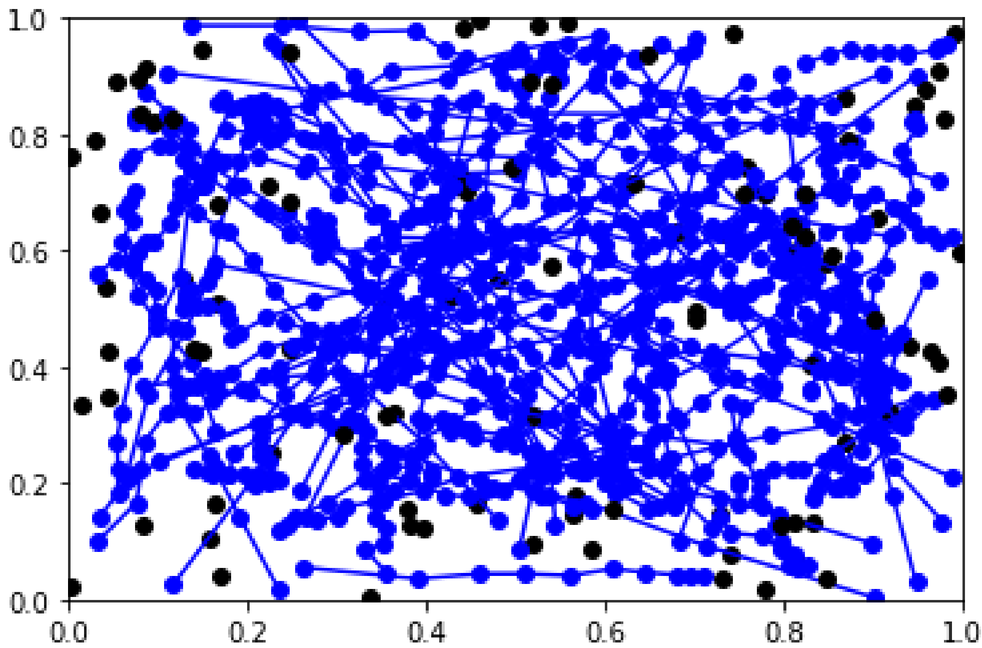
February 26, 2022

[ ]:
```python
### Link to Github Repo with the same code: ###
### https://github.com/dylan-losey/me5824.git ###

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import numpy as np
import random
```

[ ]:
```python
### Create our BC Model ###

class HumanData(Dataset):

    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return torch.FloatTensor(self.data[idx])


class BC(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim):
        super(BC, self).__init__()

        self.state_dim = state_dim
        self.action_dim = action_dim

        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, action_dim)

        self.loss_func = nn.MSELoss()
```

```python
    def encoder(self, state):
        h1 = torch.tanh(self.linear1(state))
        h2 = torch.tanh(self.linear2(h1))
        return self.linear3(h2)

    def forward(self, x):
        state = x[:, :self.state_dim]
        a_target = x[:, -self.action_dim:]
        a_predicted = self.encoder(state)
        loss = self.loss(a_predicted, a_target)
        return loss

    def loss(self, a_predicted, a_target):
        return self.loss_func(a_predicted, a_target)
```

[ ]:
```python
### Collect the human demonstrations ###

N = 100                 # number of demonstrations
sigma_h = 0.01           # amount of noise in the demonstration
T = 10                  # each demonstration has T timesteps
D = []                  # dataset of state-action pairs

for iter in range(N):
    xi = np.zeros((T, 2))
    p_robot = np.random.rand(2)
    p_goal_1 = np.random.rand(2)
    p_goal_2 = np.random.rand(2)
    choice = random.randint(0,1)
    if choice == 0:
      p_goal = p_goal_1
      target = [1, -1]
    else:
      p_goal = p_goal_2
      target = [-1, 1]

    for timestep in range(T):
        a = np.random.normal((p_goal - p_robot) / 5.0, sigma_h)
        xi[timestep, :] = np.copy(p_robot)
        D.append(p_robot.tolist() + p_goal_1.tolist() + p_goal_2.tolist() +␣
 ↪target + a.tolist())
        p_robot += a
    plt.plot(p_goal[0], p_goal[1], 'ko')
    plt.plot(xi[:,0], xi[:,1], 'bo-')
plt.axis([0, 1, 0, 1])
plt.show()
```

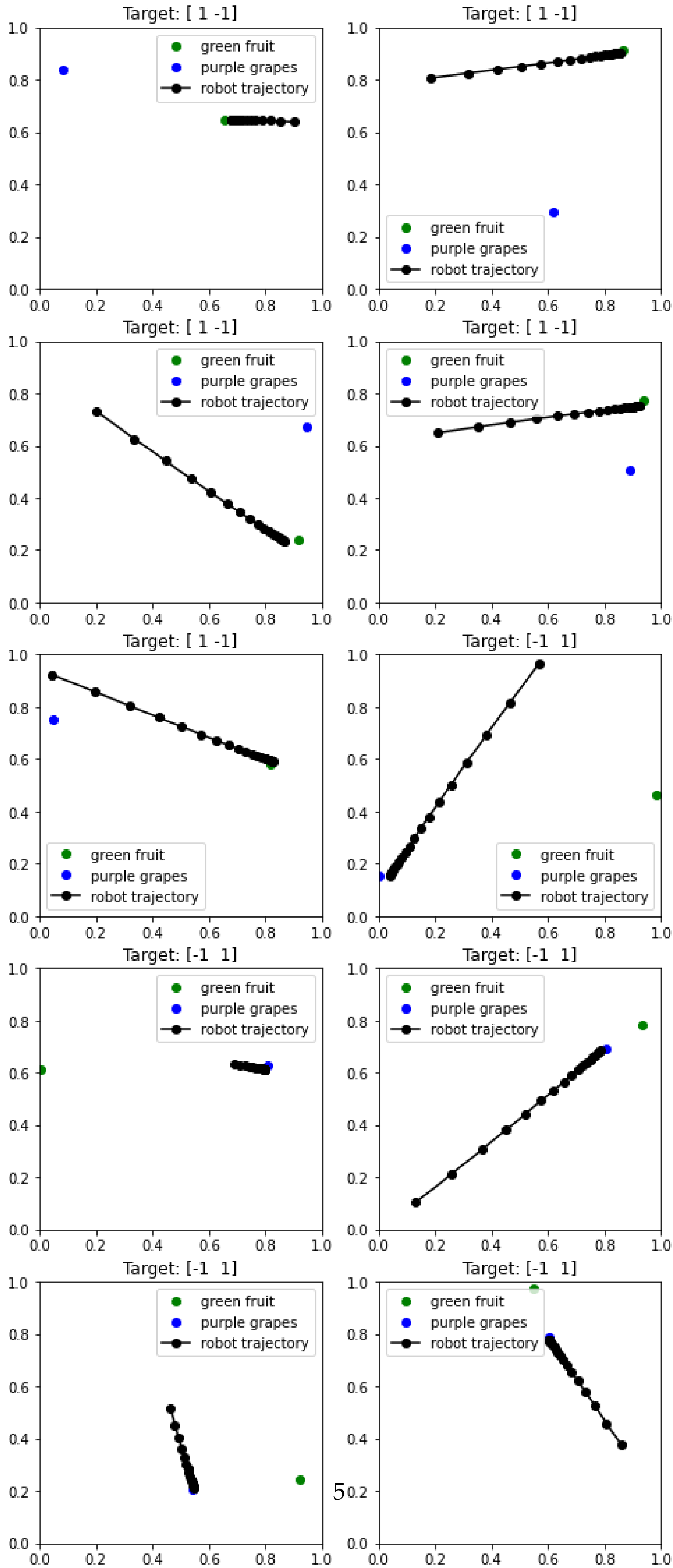


[ ]:
```
### Train the BC Model ###

# arguments: state dimension, action dimension, hidden size
model = BC(8, 2, 32)

EPOCH = 1001
BATCH_SIZE_TRAIN = 100
LR = 0.01
LR_STEP_SIZE = 360
LR_GAMMA = 0.1

train_data = HumanData(D)
train_set = DataLoader(dataset=train_data, batch_size=BATCH_SIZE_TRAIN,␣
 ↪shuffle=True)

optimizer = optim.Adam(model.parameters(), lr=LR)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=LR_STEP_SIZE,␣
 ↪gamma=LR_GAMMA)

for epoch in range(EPOCH):
  for batch, x in enumerate(train_set):
      optimizer.zero_grad()
      loss = model(x)
      loss.backward()
      optimizer.step()
  scheduler.step()
```

```
  if epoch % 100 == 0:
    print(epoch, loss.item())
torch.save(model.state_dict(), "bc_weights")
```

```
0 0.005890688393265009
100 0.00011825011461041868
200 0.0001473638549214229
300 0.00014695516438223422
400 8.811458974378183e-05
500 0.0001227668981300667
600 9.010592475533485e-05
700 7.98568144091405e-05
800 9.984023199649528e-05
900 0.00010542211384745315
1000 9.910707012750208e-05
```

[ ]:
```
### Rollout the trained model ###

model = BC(8, 2, 32)
model.load_state_dict(torch.load("bc_weights"))



N = 10          # number of rollouts
T = 20          # each one has T timesteps
fig, ax2d = plt.subplots(5,2, figsize=(8, 20), squeeze=False)
axli = ax2d.flatten()
for iter, ax in enumerate(ax2d.flat):
    xi = np.zeros((T, 2))
    p_robot = np.random.rand(2)
    p_goal_1 = np.random.rand(2)
    p_goal_2 = np.random.rand(2)
    if iter < N/2:
      target = np.array([1, -1])
    else:
      target = np.array([-1, 1])
    for timestep in range(T):
        context = np.concatenate((p_robot, p_goal_1, p_goal_2, target))
        a = model.encoder(torch.Tensor(context)).detach().numpy()
        xi[timestep, :] = np.copy(p_robot)
        p_robot += a
    ax.plot(p_goal_1[0], p_goal_1[1], 'go', label='green fruit')
    ax.plot(p_goal_2[0], p_goal_2[1], 'bo', label='purple grapes')
    ax.plot(xi[:,0], xi[:,1], 'ko-', label='robot trajectory')
    ax.axis([0, 1, 0, 1])
    ax.legend()
    ax.set_title('Target: ' + str(target))
plt.show()
```


Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [1 -1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
Target: [-1 1]
green fruit
purple grapes
robot trajectory
0.0
0.2
0.4
0.6
0.8
1.0